View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Científico da Universidade de Évora

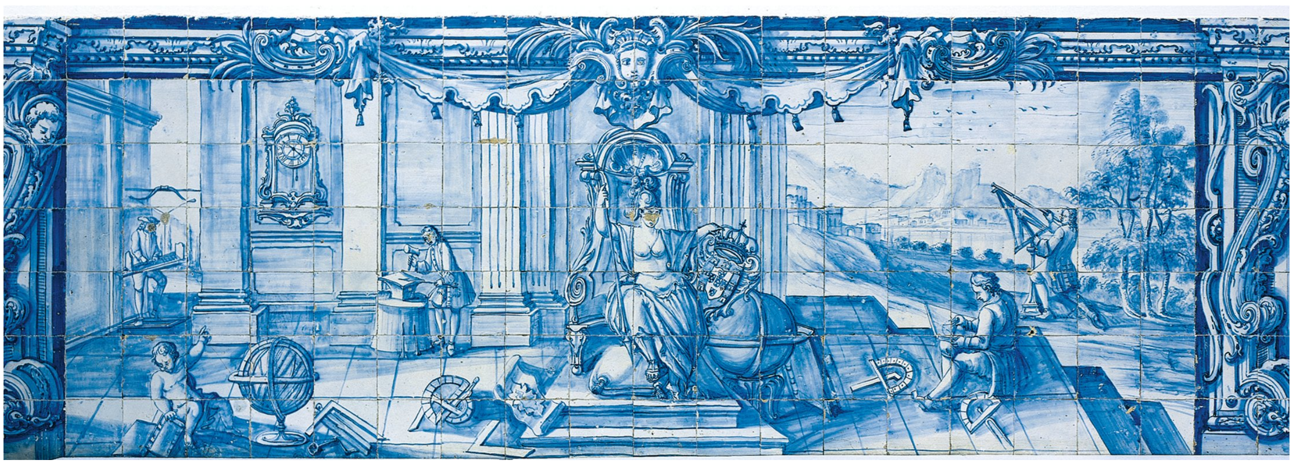
MASSIVELY PARALLEL DECLARATIVE COMPUTATIONAL MODELS

Rui Mário da Silva Machado

Tese apresentada à Universidade de Évora
para obtenção do Grau de Doutor em Informática

ORIENTADOR: *Salvador Pinto Abreu*

ÉVORA, Julho de 2013



INSTITUTO DE INVESTIGAÇÃO E FORMAÇÃO AVANÇADA

UNIVERSIDADE DE ÉVORA

Massively Parallel Declarative Computational Models

Rui Mário da Silva Machado

Tese apresentada à Universidade de Évora para obtenção

do grau de Doutor em Informática

Orientador: *Salvador Pinto Abreu*

Évora, Julho de 2013

It is a journey, not a destination...

Acknowledgments

It is interesting to look back, remembering the circumstances and persons that crossed our path. When I started my PhD, I had no clue where it would take me. Now, looking back, it is my sincere conviction that it is the people that either leave a mark on you in some way or are present, sharing the roller coaster of the journey, what counts the most. Here I would like to thank them all, humbly apologising those I might overlook.

The first person I must thank to is Werner Kriechbaum. Werner was the person who shed the light to an option which I never considered before I met him. I will always be thankful for his mentor-ship, support and friendship.

I would like to thank to Salvador Abreu, my adviser. First of all, for his patience and readiness of advice when things were not looking so bright to me. But also for his guidance and discussions that shaped this work and allowed me to actually finish it.

My journey took place at the Fraunhofer ITWM. There I made friends, got introduced to the HPC community and learned a lot thanks to the great professionals of the HPC department. I would like to thank Franz-Josef Pfreundt for giving me the opportunity to join the department and making me feel welcome. To my office mate, Carsten Lojewski, who has become a good friend, from whom I learned a great deal of things and spent amazing moments. To all other colleagues and friends Mirko, Martin, Matthias, Ely, Jens, Valentin, Daniel for the interesting discussions, support and great Wednesday nights.

To those outside my working place but who also contributed for the realisation of this work. I would like to thank Vasco Pedro for sharing PaCCS and his knowledge with me which allowed me to progress quickly. And to Daniel Diaz and Philip Codognet for the discussions, ideas and help that made this work a better one.

Finally, simple words for those without whom life would be meaningless and insignificant. To all my friends for remembering that there is more to life than just work. To my parents whose hard work and support allowed me to get a good education; for simply being present, always, whenever I needed them, for the good or for the bad. And my bro, who is a constant presence and source of strength and inspiration in my mind. Finally to my treasure, Tania, the person who felt more closely my moments of enthusiasm but also those of frustration. Her love and support, allowed me to relax on hard moments and to focus and avoid distractions when I needed to. I dedicate this work to you.

Abstract

Current computer archictectures are parallel, with an increasing number of processors. Parallel programming is an error-prone task and declarative models such as those based on constraints relieve the programmer from some of its difficult aspects, because they abstract control away.

In this work we study and develop techniques for declarative computational models based on constraints using GPI, aiming at large scale parallel execution.

The main contributions of this work are:

- A GPI implementation of a scalable dynamic load balancing scheme based on work stealing, suitable for tree shaped computations and effective for systems with thousands of threads.
- A parallel constraint solver, MaCS, implemented to take advantage of the GPI programming model. Experimental evaluation shows very good scalability results on systems with hundreds of cores.
- A GPI parallel version of the Adaptive Search algorithm, including different variants. The study on different problems advances the understanding of scalability issues known to exist with large numbers of cores.

Keywords: UTS, GPI, Constraint Programming, Local Search, Adaptive Search, Parallel Programming, Work Stealing, Dynamic Load Balancing

Modelos de Computação Declarativos Massivamente Paralelos

Sumário

Actualmente as arquitecturas de computadores são paralelas, com um crescente número de processadores. A programação paralela é uma tarefa propensa a erros e modelos declarativos baseados em restrições aliviam o programador de aspectos difíceis, dado que abstraem o controlo.

Neste trabalho estudamos e desenvolvemos técnicas para modelos de computação declarativos baseados em restrições usando o GPI, uma ferramenta e modelo de programação recente. O objectivo é a execução paralela em larga escala.

As contribuições deste trabalho são as seguintes:

- A implementação de um esquema dinâmico para balanceamento da computação baseado no GPI. O esquema é adequado para computações em árvore e efectiva em sistemas compostos por milhares de unidades de computação.
- Uma abordagem à resolução paralela de restrições denominada de MaCS, que tira partido do modelo de programção do GPI. A avaliação experimental revelou boa escalabilidade num sistema com centenas de processadores.
- Uma versão paralela do algoritmo Adaptive Search baseada no GPI, que inclui diferentes variantes. O estudo de diversos problemas aumenta a compreensão de aspectos relacionados com a escalabilidade e presentes na execução deste tipo de algoritmos num grande número de processadores.

List of Publications

On the Scalability of Constraint Programming on Hierarchical Multiprocessor Systems Machado, Rui and Pedro, Vasco and Abreu, Salvador In *Proc.* 42^{nd} *International Conference on Parallel Processing (ICPP-2013), 1-4 Oct. 2013, Lyon, France*

Parallel Performance of Declarative Programming using a PGAS Model Machado, Rui and Abreu, Salvador and Diaz, Daniel In *Practical Aspects of Declarative Languages (PADL13), 21 - 22 Jan. 2013, Rome, Italy*

Parallel Local Search: Experiments with a PGAS-based programming model. Machado, Rui and Abreu, Salvador and Diaz, Daniel In *12th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS 2012), 4. Sep. 2012, Budapest, Hungary*

A PGAS-based Implementation for the Unstructured CFD Solver TAU. Simmendinger, Christian and Jägersküpper, Jens and Machado, Rui and Lojewski, Carsten. In *5th Conference on Partitioned Global Address Space Programming Models, 15. - 18. Oct. 2011, Galveston Island, Texas, USA.*

Unbalanced Tree Search on a Manycore System using the GPI Programming Model. Machado, Rui and Lojewski, Carsten and Abreu, Salvador and Pfreundt, Franz-Josef In *Computer Science - Research and Development, Vol. 26, Issue 3-4, pp. 229-236, 2011*

The Fraunhofer Virtual Machine: a communication library and runtime system based on the RDMA model. Machado, Rui and Lojewski, Carsten In *Computer Science - Research and Development, Vol. 23, Issue 3-4, pp. 125-132, 2009*

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

AS Adaptive Search

CBLS Constraint-Based Local Search

CSP Constraint Satisfaction Problem

COP Constraint Optimisation Problem

GPI Global address space Programming Interface

MaCS Massively parallel Constraint Solver

MPI Message Passing Interface

PaCCS Parallel Complete Constraint Solver

PGAS Partitioned Global Address Space

RDMA Remote Direct Memory Access

UTS Unbalanced Tree Search

Chapter 1

Introduction

1.1 Motivation

Ever since it was stated, Moore's Law has adequately described the progress of newly developed processors. Before the turn of the millennium, the performance improvements of a processor were mostly driven by frequency scaling of a single processor. Still, multiprocessor chips were already seen as a valid approach to increase performance. The focus on multiprocessor designs became clear with the diminishing returns of frequency scaling and emerging physical limitations.

The emergence of single-chip multiprocessors is a consequence of a number of limitations in today's microprocessor design: deep pipe-lining performance exhaustion, reduced benefits of technology scaling for higher frequency operation, power dissipation approaching limits and memory latency. To address these limitations and continue to increase performance, many chip manufactures started to research and develop multicore processors. Nowadays, these are dominant and used in all kinds of platforms, from mobile devices to supercomputers.

Under this shift, the software becomes the problem since it has to be modified for parallelism and to take advantage of a thousand or even of a million-way parallelism. And this affects all layers of software, from operating systems to applications and from programming models and run-time systems to algorithms. However, parallel programming is difficult and requires real programming expertise together with a great knowledge of the underlying hardware.

In the field of high performance computing (HPC), writing parallel software is a requirement to take advantage of supercomputers. Thus, observing the research developments in this area can be an advantage. Moreover, a large body of knowledge on parallel computing and programming is therein concentrated.

In HPC, the most popular approach for writing parallel programs is the Message Passing Interface (MPI). MPI has been very successful but recently, researchers have started looking for and developing alternatives.

GPI was created and continues to be developed at the Fraunhofer ITWM as an alternative to MPI. It has already proved its applicability in industrial applications provided by the Fraunhofer ITWM and its partners, in different domains such as seismic imaging, computational fluid dynamics or visualisation techniques such as ray tracing. It is a PGAS [1] API for parallel applications running on cluster systems. GPI was created to take advantage of modern features of recent interconnects and incur in low CPU utilisation for communication. Its programming model focuses on asynchronous one-sided communication, moving away from a bulk-synchronous two-sided communication pattern, an often observed pattern in parallel applications.

Notwithstanding the success and effectiveness of GPI in solving real world problems, parallel programming is still hard specially for scientists with less background on Computer Science but with a large expertise in their domain. Moreover, applications possess different characteristics and kinds of parallelism, from which it might be straightforward or not to obtain benefit.

Declarative computational models can simplify the modelling task and are thus attractive for parallel programming, as they concentrate on the logic of the problem, reducing the programming effort and increasing the programmer productivity. The issue of how the programmer expresses its problem is orthogonal to the underlying implementation which should target and exploit the available parallelism.

Among the different declarative models, those based on constraint programming have been successfully applied to hard problems, which usually involve exploring large search spaces, a computationally intensive task, but one with significant potential for parallelisation.

It is thus promising to match such declarative models with recent technologies such as GPI and study the suitability and scalability at large scale of such engagement.

1.2 Constraint Programming

Constraint Programming (CP) is one approach to declarative programming based on constraints. In CP, a problem is modelled as a set of variables where each of them has a domain. The domain of a variable is the set of possible values it can take. Moreover, a set of relations between (some) variables is defined. This set is called the constraints of the problem. Solving a problem is thus, the assignment of a value of the domain to each variable such that no constraint is violated.

There exist different methods to solve a problem. One can distinguish between *complete* and *incomplete* methods. *Complete* methods always find a (best) solution or prove that no solution exists, whereas *incomplete* methods cannot give such guarantee. Nevertheless, *incomplete* methods are often very fast and return good quality solutions.

[1]PGAS - Partitioned Global Address Space

One of the important techniques used in *complete* methods is *search*.

Walking through a large search space looking for a solution state has a good potential for parallelisation by simply splitting the search space among different processing units, one can expect to get good speedups.

PaCCS [109]is a recent and state-of-art parallel complete constraint solver which has shown scalable performance on various problems. PaCCS explores the idea of search space splitting among different workers and teams, implementing techniques to ensure that all units are kept busy at all times.

1.3 Constraint-Based Local Search

Local Search is an effective technique to tackle optimisation problems and constraint satisfaction problems. The general idea of local search algorithms is simple: starting from an initial "candidate solution", the algorithm performs a series of improving modifications (*moves*) to the current configuration until a stop criterion is met, which usually means that a solution was found.

Local Search algorithms are not guaranteed to find a (optimal) solution because they are incomplete and do not perform an exhaustive search. But they are very fast, often return good quality solutions and are able to tackle large problem instances.

Constraint-based Local Search builds on the idea of using constraints to describe and control local search. Problems are modelled using constraints plus heuristics and solutions are searched for using Local Search.

The Adaptive Search algorithm is one of many different local search methods which proved to be very efficient in the types of problems on which it was tested. Adaptive search [20, 21] is, by definition, a local search method for solving Constraint Satisfaction Problems (CSP). The key idea of the approach is to take into account the structure of the problem given by the CSP description, and to use in particular variable-based information to design general meta-heuristics.

1.4 Goals and Contributions

The main goal of this thesis is the study of declarative computational models based on constraints, with the execution on large parallel systems in mind. More specifically, we are interested in the scalability of complete constraint solving and of a constraint-based local search algorithm (Adaptive Search) when implemented to take advantage of the GPI programming model. On the other hand, we want to evaluate the suitability of GPI to such problems and the requirements of their parallel execution. To that end, different analysis and techniques for both models were implemented and are here presented.

In summary, the main contributions of this thesis are the following:

- a GPI implementation of a scalable dynamic load balancing scheme based on work stealing, suitable for the requirements of parallel tree search and tree shaped computations. The implementation was evaluated on a recent system with up to 3072 cores and displayed good results and scalability.
- A parallel constraint solver, MaCS, based on the ideas of PaCCS but implemented to take advantage of the GPI programming model and leveraging the knowledge obtained with general parallel tree search. The experimental evaluation on different problems showed encouraging scalability results on a system with up to 512 cores.
- A new parallel version of the Adaptive Search algorithm with GPI, including different variants and the study of its behaviour on different problems. The study provides a deeper understanding of some scalability problems exhibited by the algorithm when executed at large scale.

Ultimately, the contributions of this thesis aim at spreading the use of declarative models based on constraints given their natural simplification of the parallel programming task and claim the establishment of GPI as a valid alternative for the implementation of constraint-based systems.

1.5 Thesis Organisation

This document is organised as follows. Chapter 2 provides the necessary context for the work presented in this thesis, with respect to parallel programming and presents GPI and its programming model.

The rest of the thesis is organised in two major parts which correspond to the two different declarative computational models we are interested in: Part I deals with the development of a parallel complete constraint solver whereas in Part II we focus on Constraint-based Local Search by exploring the Adaptive Search algorithm.

Part I includes Chapters 3 to 5. Chapter 3 introduces constraint programming and constraint solving (including its parallelisation), providing the required background for this declarative model. Chapter 4 focuses on Parallel Search as one important aspect of parallel constraint solving and load balancing as the major obstacle to overcome. The work with the UTS benchmark is presented and the dynamic load balancing scheme based on work stealing with GPI is analysed. Chapter 5 presents MaCS, the Massively parallel Constraint Solver, based on GPI and an architecture based on that of PaCCS. It leverages the work with UTS to implement a complete constraint solver for parallel systems. Finally, Chapter 6 wraps the first part of this thesis, summarising the work presented and providing directions for future work on the topic.

Part II includes Chapters 7 and 8. In Chapter 7, we present Constraint-based Local Search and other Local Search methods with focus on the parallelisation of Local Search algorithms. Chapter 8 presents the Adaptive Search algorithm and work developed towards its parallelisation with GPI, with the analysis and performance evaluation on different problems. Part II is finalised with Chapter 9 where again a summary of the work of this part is presented and different directions of research are listed.

This thesis concludes with Chapter 10 where both of its parts are put into a common perspective.

Chapter 2

Background

Modern computer architectures continue to move to increasingly parallel systems. On the software side, this trend to parallel systems implies a change towards parallel programming. This means that parallel programming is no longer a topic specific to the HPC area or only for scientists working on supercomputers.

Fortunately, there is a large amount of work on parallel computing and parallel programming. In this chapter we introduce some important concepts and give the necessary context for the development of the work in this thesis where GPI is a central and important point. Its features, the programming model and functionality are extensively presented.

2.1 Parallel Programming

Parallel programming is often seen as a challenging task. The programmer has the added responsibility of dealing with aspects not present when the goal is sequential execution. These include, among others, non-deterministic behaviour, load balancing and a good match to the underlying parallel computer.

When developing a parallel algorithm, the first step to be considered is to identify components that could be processed in parallel, in other words, to find concurrency in a problem in order to decompose it into sub-problems that can be independently processed in parallel.

Parallelism can take different forms: with respect to the data that needs to be processed or to the different operations to be performed. *Data parallelism* is exhibited when the same operation is performed on the same or different elements of a data set. *Task parallelism* (also know as *functional parallelism*) is exhibited when multiple operations can be performed concurrently. These include independent parts (*tasks*) of a program such as function calls, basic blocks or single statements that can be run in parallel by different processors.

Most large problems contain enough concurrency to be exploited by parallel execution. Notwithstanding, their parallelisation has different requirements. Some problems require little effort to decompose them, with no dependencies and communication needs (*embarrassingly parallel*). Other problems can be partitioned statically, using domain decomposition techniques, with careful design to deal with data dependencies and communication. And finally, some problems depend on dynamic information where the amount of parallelism is initially unknown. Such problems require different and more dynamic techniques.

2.1.1 Irregular and Dynamic Problems

The development of parallel applications requires different optimisation and programming techniques to match the different types of problems. Within the whole range of problems, some possess characteristics which lead us to classify them as irregular and dynamic. Examples include optimisation problems or heuristic search problems and are important in different domains such as SAT solving and machine learning.

The characteristics of such applications include unpredictable communication, unpredictable synchronisation and/or a dynamic work granularity. One common requirement is an asynchronous and dynamic load balancing scheme because the inherent unpredictability of the computation does not permit a static partitioning of work across the computing resources.

Unpredictable communication corresponds to an execution of a program where the occurrence of communication changes frequently. An unpredictable communication schedule raises problems to the parallel execution since sender and receiver may need to hand-shake the communication which implies a synchronisation step and possibly blocking one of the parties. Unpredictable synchronisation pattern refers to synchronisation depending on the state of the computation. It is a similar problem to the unpredictable communication. Irregular and asynchronous applications suffer from these characteristics resulting in load imbalance and waste of CPU cycles. Dynamic work granularity, where the computation depends on the input, also creates load imbalance. For example, in a search problem, each node might generate a different number of child nodes to be processed and consequently a work imbalance on each worker.

2.1.2 Performance

When solving a problem in parallel, one of the fundamental objectives is its improvement in terms of performance. One way to consider performance is as a reduction of the required execution time. Moreover, it is reasonable to expect that this execution time reduces as more processing power is employed. Scalability of a parallel algorithm is the measure of its capacity to deal with an increasing number of processing units. It can be used to predict the performance of a parallel algorithm on a large number of processors or, for a fixed problem size, determine the optimal number of processors needed and the maximum speedup that can be obtained.

For scalability analysis and understanding parallel performance in general some common definitions are required:

Speedup is the ratio of the execution time of the fastest sequential algorithm (T_S) to the execution time of the parallel version (T_P).

Serial fraction is the ratio of the serial part of an algorithm to the sequential execution time. The serial part is that part of the algorithm which cannot be parallelised.

Parallel overhead is the sum of the total overhead incurred due to parallel processing. This includes communication and synchronisation costs, idle times or extra or redundant work performed by the parallel implementation.

Efficiency is the ratio of speedup to the number of processors (p) that is, $E = T_S/(pT_P)$.

Degree of concurrency is the maximum number of tasks that can be executed simultaneously in a parallel algorithm.

When discussing the scalability of a parallel algorithm, the speedup is the often used metric. In the ideal case, the speedup should be *linear* with $S_P = p$ that is, the speedup on p processors should equal p. Some problems can even achieve *super linear* speedup where $S_P \geq p$. Super linear speedups happen often due to hardware related effects such as cache effects but may also be due to the nature of the algorithm.

It is often the case, that the speedup does not continue to increase with increasing number of processors, for a **fixed problem size**. Amdahl [2] observed that the speedup of a parallel algorithm is bounded by $1/s$ where s is the serial fraction of the algorithm. This popular statement is know as **Amdahl's Law**. In fact, besides the serial fraction, other factors contribute to a levelling-off or even a decrease of the speedup such as the degree of concurrency or all sorts of parallel overhead.

The problem size is important because speedup may saturate due to overhead growth or to exceeded degree of concurrency. For a fixed problem size, the parallel overhead increases with increasing number of processors, reducing the efficiency. The efficiency can be maintained if, for an increasing number of processors p, the problem size W is also increased. The rate with which W should be increased with respect to p in order to keep a fixed efficiency determines the scalability. **Gustafson's Law** [58] provides a counterpoint to Amdahl's Law in which a new metric called *scaled speedup* is introduced. Scaled speedup is the speedup obtained when the problem size is increased linearly with the number of processors [80].

2.2 Parallel Programming Models

A parallel programming model is an abstract view of the parallel computer, how processing takes place from the standpoint of the programmer. This view is influenced by the architectural design and the language, compiler or runtime libraries and thus, there exist many programming models for the same architecture.

A programming model defines how units of computation are executed and how these are mapped to the available architectural resources. This is called the execution model. The most popular execution model is the SPMD (Single Program, Multiple Data) where the same program is run by different processes which act on different (multiple) data.

A programming model also defines how data is shared. There are models with a shared, distributed address space or with a hierarchical view of it.

The most popular parallel programming models are message passing, shared memory and data parallel models. All these models have their strenghts and weaknesses which will be highlighted in this section.

In the message-passing model, several processes execute concurrently, each of them executing, sequentially, their instructions. The processes cooperate through the exchange of messages since each process only has access to its own private memory space. This message exchange is two-sided that is, there is a sender and a corresponding receiver of the message.

As disadvantages, there is a considerable overhead associated with the communication, specially with small messages. And because each process has a private view of data, for large problem instances which do not fit in that private space, some extra concern with data layout and decomposition has to be introduced. This can reduce the productivity of the programmer.

In the shared memory model, multiple independent threads work on and cooperate through a common, shared address space. There is no difference between a local or a remote access. This results in greater ease of use due to its similarity to the sequential execution. The downside is that there is no control over the locality of data which could generate unnecessary remote memory accesses, degrading performance. Moreover, if the programming environment does not handle it, access to shared data must be synchronised to avoid inconsistencies.

The data parallel model concentrates, as the name suggests, on the data. Concurrency is achieved by processing multiple data items, simultaneously. A process operates, in one operation, on multiple data items. For example, the elements of an array to be processed in a loop can be partitioned and concurrently operated by different threads or by the use of vector instructions of the CPU. This is the case of for instance, many image processing algorithms. For applications rich in data parallelism, this model is very powerful. On the other hand, for problems with more task parallelism, this model might be less effective.

2.2.1 MPI

MPI [98] has become the *de facto* standard for communication among processes that model a parallel program running on a distributed memory system. It primarily addresses the *message passing* programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.

MPI is a specification, not an implementation; there are multiple implementations of MPI (OpenMPI [44], MVAPICH2 [67], MPICH [56] and others).

Notwithstanding its achievable performance and portability, two of its major advantages, the recent developments in architectural design made researchers start to question the scalability of the programming model in the pre-Exascale era.

Since its version 2.0, the MPI standard includes support for one-sided communication to take advantage of hardware with RDMA [1] capabilities. Still, there are some restrictions which limit the applicability of MPI. These restrictions are basically related to memory access (in [11] an extensive review of these restrictions is done for the case of Partitioned Global Address Space (PGAS) Languages).

Another limitation is the implicit synchronisation of the 2-sided communication scheme, which requires both peers to participate in the information exchange even if only one of them actually needs it. This is particularly important for applications with irregular data access patterns. Also, because a sender cannot usually write directly into a receiver's address space, some intermediate buffer is likely to be used [16]. If such buffer is small compared to the data volume, additional synchronisation occurs whenever the sender must wait for the receiver to free up the buffer.

Lastly, with the transition to multicore and manycore system designs, the process-based view of MPI maps poorly to the hierarchical structure of available hardware. Recent research results using hybrid approaches (MPI+OpenMP) are a reflex of that [72, 113].

These are of course issues of which the MPI community is well aware. Indeed, the future version of the MPI standard [22, 99] aims at addressing some of them as well as do new releases of MPI implementations.

2.2.2 PGAS

The Partitioned Global Address Space (PGAS) is another, more recent, parallel programming model which has been seen as a good alternative to the established MPI one. The PGAS approach offers the programmer an abstract shared address space model which simplifies the programming task and at the same time promotes data-locality, thread-based programming and asynchronous communication. One of its main arguments is an increased programming productivity with competing performance.

With the PGAS programming model, each process has access to two address spaces: a global and a local (private) address space. The global address space is partitioned and each partition is mapped to the memory of each node. The programmer is aware of this layout allowing her a higher control of data locality: local or in global address space; in own partition or in a remote partition.

[1]RDMA - Remote Direct Memory Access

Data in the global address space is fetched through *get* and *put* operations. These operations have a one-sided semantics that is, only one side of the communication is required to invoke the communication.

The PGAS programming model is used by languages for parallel programming which aim at development productivity of parallel programs. The Unified Parallel C (UPC) [24] and Co-Array Fortran [104] are extensions of the C and Fortran programming languages respectively, which aim at high performance computing. With the same aim, X10 [18], Chapel [12], Fortress [73] are novel parallel programming languages, designed with scalable parallel applications and high programmer productivity in mind.

There has also been a substantial amount of work on communication libraries and runtime systems that support the PGAS model. ARMCI [69] is a communication library that implements RDMA operations on contiguous and non-contiguous data. The target of ARMCI though, is to create a communication layer for libraries and compiler runtime systems. For example, it has been used to implement the Global Arrays library [70] and GPSHMEM [74].

GASNet [10] is another communication library that aims at high performance. It provides network independent primitives and as for ARMCI, several implementations exist. GASNet also targets compilers and runtime systems and is used for the implementation of the PGAS languages UPC, Titanium [108] and Co-Array Fortran.

Older alternatives already exploited the same concepts (e.g. remote memory access, shared memory communication) and goals (e.g. performance). The well-known Cray SHMEM [5] library available on Cray platforms, as well as on clusters, supported put/get operations and scatter/gather operations, among other capabilities.

It is worth to note that the PGAS programming model is a variant and differs from the classical Distributed Shared Memory (DSM) and systems such as Treadmarks [76]. Under PGAS there is no coherency mechanism and it is the programmer who must care for the consistency of data. The direct control aims at, and usually results in, higher performance.

2.3 GPI

GPI[2] (Global address space Programming Interface) is a PGAS API for parallel applications running on clusters [87]. It has already proved its applicability on industrial applications developed by the Fraunhofer ITWM and its partners, in different domains such as seismic imaging, computational fluid dynamics or stencil-based simulations [79, 124, 57].

A crucial idea behind GPI is the use of one-sided communication where the programmer is implicitly guided to develop with the overlapping of communication and computation in mind. In some applications, the computational data dependencies allow an early request for communication or a later completion of a transfer. If one is able to find enough independent

[2]GPI was previously known as Fraunhofer Virtual Machine (FVM)

computation to overlap with communication then potentially a rather small amount of time is spent waiting for transfers to finish. As only one side of the communication needs information about the data transfer, the remote side of the communication does not need to perform any action for the transfer. In a dynamic computation with changing traffic patterns, this can be very useful [6].

Its thin communication layer delivers the full performance of RDMA-enabled networks directly to the application without interrupting the CPU. In other words, as the communication is completely off-loaded to the interconnect, the CPU can continue its computation. While latency could and should be hidden by overlapping it with useful computation, data movement gets reduced since no extra intermediate buffer is needed and thus, bandwidth does not get affected by it.

From a programming model point of view, GPI provides a threaded approach as opposed to a process-based view. This approach provides a better mapping to current systems with hierarchical memory levels and heterogeneous components.

In general, and given its characteristics, GPI enables and aims at a more asynchronous execution. It requires a reformulation and re-thinking of the used algorithms and communication strategy in order to benefit for its characteristics. For instance, GPI does not aim at simply replacing MPI calls and provide an alternative interface for parallel programs.

2.3.1 GPI Programming Model

In GPI, the programmer views the underlying system as a set of nodes where each node is composed of one or more cores. All nodes are connected to each other through a DMA interconnect. This view maps, more or less directly, to a common cluster system. Figure 2.3.1 depicts the architecture of GPI.

GPI applies the SPMD execution model and each node has an identifier, the *rank*. The same application is started on every node and the rank is used to identify each node and for communication between nodes.

As already mentioned, GPI is a PGAS API. As in every PGAS model and from the memory point of view, each node has an internal and a global memory. The local memory is the internal memory available only to the node and allocated through typical allocators (e.g. malloc). This memory cannot be accessed by other nodes. The global memory is the partitioned global memory available to other nodes and where data shared by all nodes should be placed. Nodes issue GPI operations through the DMA interconnect.

At the node level, GPI encourages and in a sense, even enforces a thread-based model to take advantage of all cores in the system. In figure 2.3.1, each core is named a MCTP thread. MCTP, which stand for Many-Core Thread Package, is a library used with GPI and based on thread pools that abstract the native threads of the platform. Each thread has access to the local and the node's partition of global memory as in a shared memory system. To access the global memory on a remote node, it uses one-sided communication

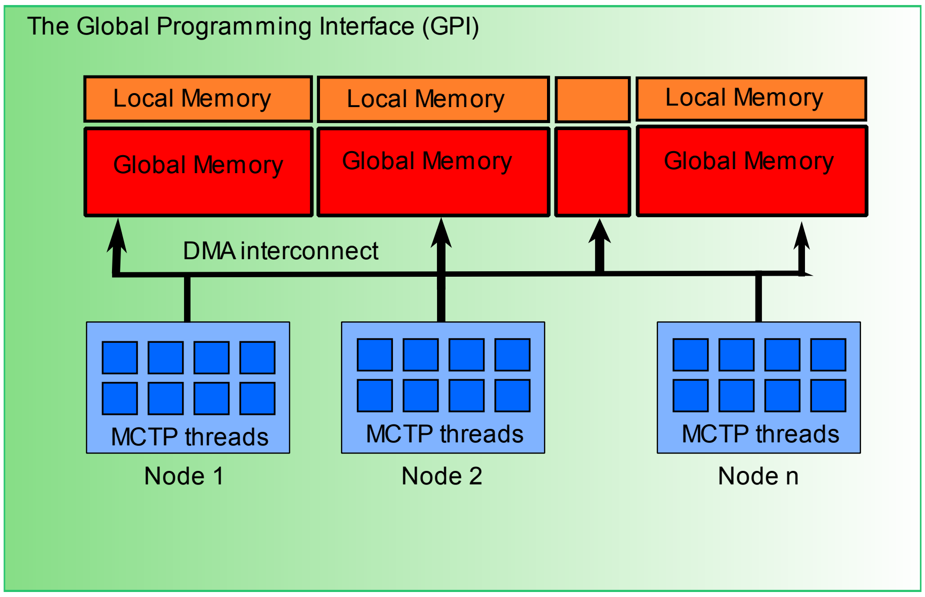


Figure 2.3.1: GPI architecture

such as *write* and *read* operations. The global view of memory is maintained but with more control over its locality.

A thread should work asynchronously, making use of one-sided communication for data access but overlapping it with computation as much as possible. The final objective should be a full overlap of computation and communication, hiding completely the latency of communications. A secondary objective is the reduction of global communication such as barriers and their cost due to synchronisation, allowing the computation to run more asynchronously.

2.3.2 GPI Functionality

One of the most attractive features of GPI is its rather simple and short interface, providing a smooth learning curve specially for someone with some background in parallel programming and/or with MPI.

GPI is constituted by a pair of components: the GPI daemon and the GPI library. The GPI daemon runs on all nodes of the cluster, waiting for requests to start applications and the library holds the functionality available for a program to use.

The GPI core functionality can be summarised as follows:

- Read and write global data,
- Send and receive messages,
- Passive communication,
- Commands,

- Global atomic counters and spin-locks,
- Barriers,
- Collective operations.

Two operations exist to read and write from global memory independent of whether it is a local or remote location. The important point is that those operations are one-sided and non-blocking, allowing the program to continue its execution and hence take better advantage of CPU cycles. If the application needs to make sure the data was transferred, it needs to call a wait operation that blocks until the transfer has finished, asserting that the data is usable.

GPI also supports a message-passing mode, allowing a more Send/Receive style of programming, where nodes exchange messages or commands. Exchanging messages means sending and receiving messages. For each send on one host there should be a corresponding receive on the other side. The send operation is non-blocking allowing the sender to continue computation while a receive blocks until a message arrives.

Nodes can also exchange commands. Commands are comparable to pre-defined messages that correspond to a certain request to which the receiver responds with the appropriate behaviour. The application is responsible for defining its own commands: when to send them, when to receive, what does each command mean and whether the command is a global command or targeted at/expected from a particular node.

One aspect related to all types of communication is the possibility of using different queues for handling the requests. The user provides which queue she would like to use to handle the request. These queues build on the queue pair concept of the Infiniband architecture and work similarly to a queue, in a FIFO fashion. At the moment, there exist eight such queues with 1024 entries each. These queues allow more scalability and can be used as channels for different types of requests where similar types of requests are queued and synchronised together but independently from the other requests on a different queue.

The passive communication has a two-sided semantics that is, there is always a sender node with a matching receiver node. However, the receiver node does not specify the sender node for the operation and expects a message from any node. Moreover, the receiver waits for an incoming message in a passive form, not consuming CPU cycles. On the sender side the call is non-blocking but the sender is more active and specifies the node where the message should be sent. After sending the message, the sender node can continue its work, while the message gets processed by the interconnect.

Global atomic counters allow the nodes of a cluster to atomically access several counters and all nodes will see the right snapshot of the value. There are two operations supported on counters: *fetch and add* and *fetch, compare and swap.* The counters can be used as global shared variables used to synchronise nodes or events. As an example, the atomic counters can be used to distribute workload among nodes and threads during run-time. They can also be used to implement other synchronisation primitives.

Although GPI aims at more asynchronous implementations and diverging from the bulk-synchronous processing, some global synchronisation might be required. Thus, GPI implements a scalable barrier at the node level. The same justifies the implementation of other collective operations such as the AllReduce that supports typical reduction operations such as the minimum, maximum or sum of standard types.

2.3.3 GPI Programming Details and Example

It is important to detail the GPI programming model and its nuances and difficulties. The two aspects which are very important to comprehend GPI and its model are the global address space and the communication primitives.

Each node contributes with a partition of the total global space (global memory in figure 2.3.1). This global memory is not accessed by global references (pointers) but with the communication primitives (read /write) of GPI, the already mentioned one-sided communication, that acts over the DMA interconnect. The advantage here is that the communication happens without the remote node's intervention. Thus, a global address is in fact a tuple (address, node) which is passed to the communication primitive when the data required is in a remote node. Moreover, this address is not a typical virtual address but rather an offset within the space reserved for global memory (offset, node). GPI only provides the pointer to the beginning of the global memory for each node locally.

The following example illustrates the basics of GPI. At start, GPI is initialised, the size for each partition of the global memory is defined to be one Gigabyte. (For instance, if the program is executed on two nodes, the total global memory of GPI is two Gigabytes.) Then after retrieving its rank and pointer to global memory, each node will fill part of its global memory with its own rank and when everyone is finished, each node reads the filled portion of global memory directly from its neighbour. When finished, all nodes wait for each other and terminate. Note that this is one of the most basic examples possible with GPI and there are no threads at all involved (except for the main one).

The GPI global memory on each node is always present and accessible for application usage. The application must however take care of maintaining the space consistent, keeping track of where is what and to use the space wisely. For example, there is no dynamic memory allocation.

Example: Apex-MAP

One way to demonstrate the advantages of GPI is through a small example.

Apex-MAP [126] is an architecture independent performance characterisation and synthetic benchmark. In its version 1, the focus lies on data access. In the characterisation, the performance factors are the regularity of the access pattern, the size of the data accessed, spatial locality and temporal locality.

```
#define _1GB 1073741824

int main(int argc, char *argv[])
{

  // start GPI with 1GB of global memory (per node)
  startGPI(argc, argv, "", _1GB);

  // the pointer to the beginning of the global memory
  char * gpiMem = (char *) getDmaMemPtrGPI();

  // get node rank
  int rank = getRankGPI();

  //get number of nodes
  int nCount = getNodeCountGPI();

  //fill buffer locally with my data
  int i;
  for(i=1024; i < 2048; i++){
    gpiMem[i] = rank; //assuming rank < 256
  }

  int neighb = ((rank + 1 == nCount) ? 0 : (rank + 1));

  //sync to make sure everyone did it
  barrierGPI();

  // read 1024 bytes from next neighbour to the right (neighb)
  // read it _to_ offset 0 of my partition (global memory)
  // read it _from_ offset 256 of remote node's partition (global memory).
  // use GPIQueue0 as the queue for communication
  readDmaGPI(0, 256, 1024, neighb, GPIQueue0);

  //.... we could be doing useful work ....

  //wait for communication to complete
  waitDmaGPI(GPIQueue0);

  //final sync
  barrierGPI();

  shutdownGPI();

  return 0;
}
```

Using Apex-Map, the objective was to simulate the behaviour of a shared memory seismic application. The application accesses local or remote data and performs some computation on that data. The two steps - data access and computation - are then repeated for a few iterations. In principle, the access to data that is local has a much lower latency than the access to remote data and thus, the results can be computed with a much lower execution time.

We create a GPI-based Apex-MAP implementation, by including remote memory accesses using the GPI API. Moreover, the remote accesses should be overlapped with computation in order to hide their much higher latency.

Figure 2.3.2 presents the obtained results when comparing the time spent to process a data

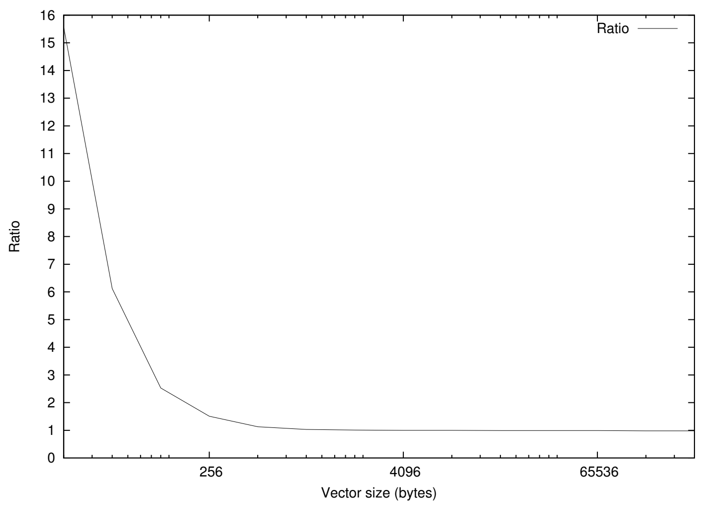


Figure 2.3.2: Apex-MAP

set that is in a remote location with that of a local data set.

As figure 2.3.2 depicts, the ratio between both times converges to one as we increase the size of the data set. In other words, we can hide the latency of the remote access given enough computation and by using programming techniques (here, double buffering) that aim at exploiting the advantages of GPI.

2.4 Summary

This chapter introduced important concepts and tools about Parallel Computing and Programming that will be of importance in the rest of this thesis. More notoriously, GPI was introduced and related to other existing approaches to parallel programming. Its programming model, the advantages of one-sided communication and asynchronous threaded computation are the central aspects and building blocks for all the work in this thesis.

Part I

Complete Constraint Solver

In the first part of this thesis, we concentrate on complete constraint solving and its execution in parallel, at large scale.

We introduce Constraint Programming, discussing modelling and solving with constraints. We then concentrate on Parallel Constraint Solving identifying the candidates for parallelisation, focusing on the search component. One chapter is dedicated to Parallel Tree Search where we, using the UTS benchmark, design an efficient dynamic load balancing mechanism. The designed solution is applied to MaCS, a parallel constraint solver based on GPI, which we present and evaluate in detail.

Chapter 3

Constraint Programming

This chapter introduces Constraint Programming, discusses modelling and solving with constraints, and presents the necessary background on this declarative approach to programming. Given the emphasis of this work on parallelism, we discuss constraint solving from the parallelisation point of view, identifying the parts that may benefit from parallel execution and how this process has been addressed on work found in the literature.

3.1 Introduction

The idea of constraint-based programming is to solve problems by stating constraints (properties, conditions) which must be satisfied by the solution(s) of the problem and then let a solver engine find a solution. Consider the following problem as an example: a bicycle number lock [43]. You forgot the first digit but you remember a few properties about it: it is an odd number, it is not a prime number and greater than 1. With this information, you are able to derive that the digit must be the number 9.

Constraints can be considered pieces of partial information. They describe properties of unknown objects and relations among them. Objects can mean people, numbers, functions from time to reals, programs, situations. A relationship can be any assertion that can be true for some sequences of objects and false for others.

Constraint Programming is a declarative approach to programming where first a *model* is defined and then a *solver* used to find solutions for the problem.

3.2 Constraint Modelling

The first step in solving a given problem with constraint programming is to formulate it as a Constraint Satisfaction Problem (CSP). This formulation is the **model** of the problem.

Definition 1. A Constraint Satisfaction Problem (CSP) over finite domains is defined by a triplet (X, D, C), where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is an indexed set of variables;
- $D = \{D_1, D_2, \ldots, D_n\}$ is an indexed set of finite sets of values, with D_i, being the domain of variable x_i for every $i = 1, 2, \ldots, n$;
- $C = \{c_1, c_2, \ldots, c_m\}$ is a set of relations between variables, called constraints.

As per Definition 1, a CSP comprises the variables of the problem and their respective domain. The domain of a variable can range over integers, reals or symbols among others but in this work we concentrate on finite domains, encoded as a finite prefix of natural numbers.

Constraint Programming is often used for and deals well with combinatorial optimisation problems. Examples are the Travelling Salesman Problem (TSP) and the Knapsack Problem, two classic NP-complete problems. In such problems one aims at finding the best (optimal) solutions from a set of solutions, maximising (or minimising) a given *objective function*.

We can define a Constraint Optimisation Problem by extending the definition of CSP with an objective function:

Definition 2. A Constraint Optimisation Problem (COP) is defined by a 4-tuple (X, D, C, obj), where:

- (X, D, C) is a CSP and,
- $obj : Sol \mapsto \mathbb{R}$ where Sol is the set of all solutions of (X, D, C)

A problem can have different models and their definition can be straightforward or not. Moreover, their efficiency can greatly vary. Modelling a problem with constraints is *per se* a task that requires expertise.

Consider the following example:

The N-Queens problem is a classical CSP example. Although simple, the N-Queens is compute intensive and a typical problem used for benchmarks. The problem consists of placing N queens on a chessboard (N x N) so that it is not possible for a queen to attack another one on the board. This means no pair of queens can share a row, a column nor a diagonal. These are the constraints of the problem.

Modelling the problem as a CSP we have:

- N variables, $\{x_1, x_2, \ldots, x_N\}$, where x_i models the queen on column i
- each variable x_i with a domain {1,2, …,N}

- the constraints:
  - $x_i \neq x_j$ *i.e.,* all variables with a different value
  - $x_i - x_j \neq i - j$ *i.e.,* no two queens in the first diagonal
  - $x_i - x_j \neq j - i$ *i.e.,* no two queens in the second diagonal

Depending on the used environment and the level of expressiveness it allows when specifying a problem, modelling the present example (N-Queens) can be as high-level and declarative as shown by the previous informal specification. One only needs to specify the properties of the desired solution, hence the classification of Constraint Programming as a *declarative* paradigm.

3.3 Constraint Solving

After correctly modelling the problem at hand, a *constraint solver* is used to get solutions for the CSP. When solving a CSP we can be interested in finding:

- a single solution,
- all solutions or the number thereof,
- an optimal solution, in the case of solving a COP with a given objective function.

A solution to a CSP is an assignment of a value from the domain of every variable, in such a way that every constraint is satisfied. When a solution for a CSP is found, we say the CSP is **consistent**. If a solution cannot be found, then the CSP is **inconsistent**. In other words, finding a solution to a CSP corresponds to finding an assignment of values for every variable from all possible combinations of assignments. The whole set of combinations is referred to as the *search space.*

Definition 3. Given a CSP $P = (X, D, C)$ where $D = \{D_1, D_2, \ldots, D_n\}$, the search space S corresponds to the Cartesian product of the domains of all variables *i.e.,* $S = D_1 \times D_2 \times \ldots \times D_n$.

A CSP can be solved by trying each possible value assignment and see if it satisfies all the constraints. However, this possibility is a very inefficient one and for many problems simply not feasible. Much more efficient methods for constraint solving exist and can be distinguished, in a first instance, between *complete* and *incomplete* methods.

Incomplete methods refer to methods that do not guarantee to find a (optimal) solution. Nevertheless, such methods often find solutions - or in case of optimisation problems, good quality solutions - and often, much faster than complete methods. This kind of algorithms is more thoroughly discussed in the second part of this thesis.

Complete methods, on the other hand, always find a (best) solution or prove that no solution exists. For that, two main techniques are used: *search* and *constraint propagation.*

Search and constraint propagation can be combined in various forms. One particular class of constraint solvers combines search and constraint propagation by interleaving them throughout the solving process. At each node of the search tree, propagation is applied to the corresponding CSP, detecting inconsistency or reducing the domains of some variables. If a fix-point is achieved, search is performed. This process continues until the goal of the solving process is reached.

Propagation

One problem with using only search techniques is the late detection of inconsistency. *Constraint propagation* is a technique to avoid this problem and its task is to prune the search space, by trying to detect inconsistency as soon as possible. This is done by analysing the set of constraints of the current sub-problem and the domains of the variables in order to infer additional constraints and domain reductions. The entities responsible for constraint propagation are called *propagators.*

Constraint propagation is executed until no more domain reductions are possible. At this point, we hit a *fix-point* and we say that the CSP is *locally consistent.*

Each time the domain of a variable has been changed, constraint propagation is usually applied. When only the domains of the variables are affected by the propagation process we call it **domain propagation**. If the propagation only tightens the lower and upper bounds of the domains without introducing “holes” in it, we call it **bounds propagation**.

Search

With *search*, a problem is split into sub-problems which are solved recursively, usually using backtracking. Backtracking search incrementally attempts to extend a partial assignment toward a complete solution, by repeatedly choosing a value for another variable and keeping the previous state of variables so that it can be restored, should failure occur. Solving a CSP is therefore, the traversal of a tree whose nodes correspond to the sub-problems (partial assignments) and where the root of the tree is the initial problem with no assignments.

An important aspect is the order in which variables and values are considered. The efficiency of search algorithms such as backtracking, that attempt to extend a partial solution, depends on these orders. The order of variables and the values chosen for each variable affects the execution time and shape of the search tree and may have a different effect on different problems. There exist various heuristics for ordering of values and variables. An example of a variable ordering heuristic is *most constrained* where the variable which appears most in the constraints is ordered least. In the case of value ordering, one example is the *random value* which orders values randomly.

Another important aspect is how the domain of a variable is split. Here too there exist different procedures. For example, *bisection* divides the values in a variable domain into two equal halves *i.e.,* for a domain $D = d_1, d_2, ..., d_n$, bisection splits the domain into $\{d_1, d_2, ..., d_k\}, \{d_{k+1}, ..., d_n\}$ where $k = \lfloor n/2 \rfloor$. Another example is *labelling*, where the domain D is split into $\{d_1\}, \{d_2, ..., d_n\}$.

The last relevant aspect related to search is the *search strategy.* It defines how the search tree should be traversed. Starting from a node, the process decides how to proceed to one of its descendants. For example, in *depth-first search (DFS)*, the procedure starts at the root node, descending to the first of its descendants until it reaches a leaf node *i.e.,* a node with no descendants. When this leaf node is reached, it backtracks to the parent node and descends again to the next descendant. This procedure is repeated until all leaf nodes have been visited and the procedure has returned to the root node.

Optimisation Problems

Solving a Constraint Optimisation Problem (COP) can be performed solely by using constraint solving techniques, taking the COP as a single CSP or a series of CSPs. In the former case, the COP is solved by finding all the solutions of the problem and choosing the one that better fits the objective function. In the latter case, several CSPs are solved by setting the domain of the constrained variable of the objective function to the smallest possible value. The value is then increased gradually until a solution is found. This applies to a minimisation problem but is easily applied to a maximisation problem where, instead of starting with the smallest value, the largest value is set and then gradually decreased.

The problem with both approaches is the generation of redundant work. A more efficient and popular algorithm for solving COPs is the *branch-and-bound* general algorithm [83]. With this algorithm, large parts of the search space are *pruned* by keeping the best solution (bound) found during the process, using it to evaluate a partial solution. If a partial solution does not lead to a better solution, when compared with the current bound, that part of the search space is avoided and left unexplored.

3.4 Parallel Constraint Solving

Due to the declarative nature of Constraint Programming, taking advantage of parallelism should be achieved at the solving step. The modelling step should remain intact as well as the user's awareness of the underlying implementation.

Parallelising Constraint Solving is, as in all kinds of algorithms and applications, a matter of identifying parts that can take advantage of parallel execution. In the literature, the parallelisation of constraint solving has faced many approaches where the different parts of the process have been experimented with.

The first obvious candidate to parallelisation is propagation where several propagators are evaluated in parallel. For problems where propagation consumes most of the computation time, good speedups may be achieved if propagation is executed in parallel. On the other hand, the overhead involved due to synchronisation and the fine grained parallelism can limit the scalability. The implementation of parallel constraint propagation involves keeping the available work evenly distributed among the entities responsible for propagation and guarantee that the synchronisation overhead is low.

The other obvious and most common approach found in the literature is to parallelise the search. Search traverses a (virtual) tree, splitting a problem into one or more sub-problems that can be handled independently and thus, in parallel. The scalability potential of tree search is encouraging, specially when the interest is large scale. However some challenges are unavoidable and since it is one of the main focus in the work of this thesis, we provide more details in Chapter 4.

Besides the two most obvious parts mentioned before, other approaches are possible. One is the use of a *portfolio* of strategies and/or algorithms where different strategies are started in parallel on the same problem. The reasoning behind this approach is that with this variety of strategies, it is to expect that one of them will outperform the others. As more parallel power is employed (and consequently more strategies) the probability that the best strategy is used, increases. This is a often and successfully used approach within the SAT solving community [135, 60]. Another approach is to distribute the variables and constraints among agents, where each agent is responsible for its own portion of the CSP. Constraints are *local* if they only involve variables local to the agent and are *non-local* if they involve variables belonging to different agents. Each agent instantiates its local variables in such way that local constraints are satisfied and non-local constraints are not violated. When an agent finds a solution, it sends a partial assignment to other agents that need to check their non-local constraints. This approach is usually referred as *Distributed Constraint Satisfaction* and the problems are referred as Distributed Constraint Satisfaction Problems (DisCSPs) [140, 141].

3.4.1 Related Work

Parallelism in CP has focused mainly on parallel search, although propagation (consistency) has also received some attention.

The research in parallel consistency has mostly focused on arc-consistency even when it has been shown to be inherently sequential [75]. Still, a great portion of research in the 90's has been produced, resulting in new algorithms and results [142, 103, 120, 121]. After that, in 2002, [59] presented *DisAC-9*, an optimal distributed algorithm for performing arc-consistency which focuses on the number of passed messages. More recently, parallel consistency has again received some interest in the work of Rolf [118, 119]. Specifically, their work studies parallel consistency with global constraints. In [119] the authors extend the work with parallel consistency with parallel search, combining both.

There has been a much more extended stream of work in parallel search as a way to exploit parallelism in CP. Some work steams from previous research in the field of Logic Programming([61, 100, 3]). Often cited work in parallel search integrated in a constraint solver is that of Perron [111, 110]. Here, the search space is represented as a tree and each processor explores a different part of the search tree. New states are entered using re-computation and a communication layer is responsible for load balancing and termination detection. The presented results are rather modest and in a small-scale (up to 4 workers).

Parallel constraint solving is included into COMET [94, 95] by parallelising the search procedure [96]. Workers “consume” their sub-problem and when idle, a work stealing mechanism is used. The generation of work to be stolen is lazy, only occurring at the time of a work request from an idle worker. The sub-problem is placed in a centralised work pool where it can be stolen by the idle worker. With up to 4 workers, several problems were tested resulting in good speedups, sometimes super-linear in the case of optimisation problems.

In [19], the authors observe that when using work stealing, the place where work is stolen can have a great impact on the efficiency of the parallel execution. Thus, they investigate a confidence based scheme for work stealing in parallel constraint solving. Given an estimate of the relative solution density of a sub-tree (confidence) at each node, each worker will follow the branching heuristic depending on its confidence value. The confidence value is constantly being updated as search progresses and each worker adapts to these updates using a timed restart mechanism: after a defined restart time, the worker tries to find a more promising sub-tree. Obvious drawbacks of this approach, specially if considering scalability, is the constant update of the confidence values along the whole tree and the need to keep the global state of the search tree.

The work mentioned so far presents results in a smaller scale (up to 16 processors) but work on larger scale has also been investigated. In [71], the authors experiment with up to 64 processors using a work stealing strategy and where workers are organised in a worker tree. All communication happens along the structure of workers where bounds, solutions and requests for work are passed. Work stealing is done directly by the *idle* worker after having received the information from the master of whom has the largest amount of work.

In [84], the authors present the first study on the scalability of constraint solving on more than 100 processors. They use two approaches, portfolios and search space splitting, and apply it to a classical CSP (N-Queens) and SAT solving. Using hashing constraints to split the search space (there is no communication involved), their results show good speedup up to 30 processors but not beyond that. The portfolio approach is argued to be the most promising approach, specially for SAT solving, but with a focus on the number of solved instances.

Large-scale parallel constraint solving is investigated in [134]. Experiments are performed on up to 1024 processors in a particular architecture, the IBM Blue Gene L and P. In their approach, processors are divided into master and worker processes, where workers explore a particular sub-tree and master processes coordinate the workers, dispatching work to them. The master keeps a tree-shaped pool where work to be dispatched is kept. The work in the pool is generated by workers when it is detected that a large sub-tree is being explored. Experiments with up to 256 processors have made clear that a single master can be a bottleneck. After adding multiple masters, scalability improves up to 1024 processors in some problems. The main cause for scalability problems on some problems is attributed to the static load sharing among masters, which is increasingly unbalanced.

It is worth to refer the problem of how to communicate the state of the search or in other words, state copying vs. state re-computation. In [136] the authors' experimental results demonstrate that when constraints are tight, the state-re-computation model has better performance than the state-copying model, but when constraints are loose, the state-copying model is a better choice. In [117], a similar question is investigated going beyond binary constraints and including global constraints. Moreover, the authors design an algorithm to switch between both methods (copying and re-computation) in order to obtain the best of both options.

Chapter 4

Parallel Tree Search

In this chapter we digress into Parallel Tree Search to emphasise its importance in the context of the work in this thesis. We look at the problem from a general standpoint and present common problems associated with the parallelisation and execution at large scale using the UTS benchmark as a representative application. We discuss and present a GPI implementation of scalable dynamic load balancing scheme based on work stealing for the UTS benchmark and evaluate its scalability on up to 3072 cores.

4.1 Introduction

In the previous chapter it was observed that search is a central component for the resolution of Constraint Satisfaction Problems. In fact, many problems are solved by searching through a search space, based on the traversal of implicitly defined trees.

Tree search is interesting from the parallelisation point of view not only because high scalability is possible but also because of the challenges it poses to the load balancing scheme. The load balancing scheme must ensure that all processing elements are active without knowledge about the shape of the tree, with a dynamic and irregular generation/granularity of work and irregular communication.

The UTS benchmark [106] aims at the characterisation of such irregular computations and at measuring their efficiency in terms of load balancing. It accomplishes this using a search space problem where a large tree of parametrised characteristics is traversed. The tree traversal generates imbalance during run-time according to the tree characteristics, therefore requiring a implementation that minimises this imbalance efficiently. This includes low communication overheads and low idle times.

Hence, we can observe that the implementation of a dynamic and asynchronous load balancing scheme based on GPI and required by parallel tree search is orthogonal to the problem at hand. Based on this observation, we focused on an application, the UTS benchmark, which aims exactly at characterising such mechanism. The initial objective was to have a general implementation that could serve as a basis for the work in this thesis, namely the parallel constraint solver.

4.2 Overview of Parallel Tree Search

The resolution of many problems can be achieved by searching through a search space until a (optimal) solution is found. The process of searching can be visualised as the traversal of a tree whose nodes correspond to states of the search space. How this tree is traversed is the subject of different techniques [55] but we focus our attention on depth-first search (DFS) as this is the common technique used by backtracking.

Initially, the problem is fully contained at an initial state, the *root* of the tree. Tree search implies that it is possible to divide a problem into *sub-problems*, that can be explored relatively independently. The *sub-problems* generally have a smaller search space than the (sub-)problem that originated them. The process of search is thus the generation and processing of sub-problems (states) in the tree until there is no more dividable sub-problems or the processing results in the desired result (solution). Algorithm 1 shows a generic version of the process.

Algorithm 1 Generic algorithm

```
1: $S \leftarrow InitialProblem(root)$          ▷ initial set $S$ contains the root (initial state)
2: while $s_i \leftarrow getNext(S)$ do         ▷ get the next (sub-)problem (state) in $S$
3:     $branch(s_i, S)$                         ▷ divide $s_i$ and add children to set $S$
4:     $process(s_i)$                           ▷ process $s_i$
5: end while
```

Implementing Parallel Tree Search is, in principle, simple since the sub-problems can be processed independently and thus, in parallel. The major obstacle though, is how to achieve a good division of the search space in order to have all processing units actively searching and not idle due to a smaller search space.

This load imbalance problem is a general problem when considering parallelism. In the case of a load balancing scheme for parallel tree search, it is independent of the actual problem at hand since the tree search is general enough to represent different kinds of problems.

One interesting approach to solve the load imbalance problem is work stealing. Work stealing is a technique in which idle threads *steal* work from threads with surplus of work (*victims*) and which are working. Semantically, work stealing is in fact a *one-sided* operation since the *thief* does not want to interrupt the *victim*. This suggests a good match to the GPI programming model which almost enforces and yields better results in such *one-sided* scenario. In practice, this is more difficult since threads do need to synchronise around the access to the work packages in order to avoid duplication of work and thus, with such a search problem, avoid the explosion of the search space.

From the parallel execution point of view, the other challenge posed by Parallel Tree Search is termination detection. Given the imbalanced nature of the computation, if a thread becomes idle it must distinguish between idleness due to its lack of work or if in fact there is no more work, at all, to be performed.

4.2.1 Speedup Anomalies

It is worth to provide a note on the scalability of search because some not so obvious behaviour is usually observed.

The scalability of parallel search is subject to a common behaviour usually referred as *speedup anomalies* [115, 114]. These anomalies are due to the difference on the number of states visited by the sequential and parallel versions and the distribution of solutions over the search space.

In case we need to explore the complete search space, linear speedup can be achieved, given an efficient load balancing scheme. This is the case if the objective of the search is to find or count all solutions of the problem or to find an optimal solution. On the other hand, if the goal is to find any solution, there is no need to explore the whole search space. The number of nodes explored by the sequential and parallel versions may differ given the dynamic nature of the search process. If the parallel version explores less nodes, a *super-linear* speedup may be observed.

Another kind of anomaly is to see no speedup at all. If the search space is divided into n equal parts executed by n processors and $n-1$ parts contain no solutions, then $n-1$ processors are executing *useless* and redundant work and this will result in no speedup at all.

4.2.2 Related Work

Load balancing is a central aspect of parallel computing that has been studied and analysed many times in the literature. In [81] several schemes for scalable load balancing are presented and analysed for a variety of architectures. More recently, in [29] the authors discuss the new challenges in dynamic load balancing and how they address them with Zoltan [30]. A common problem concerns task scheduling of tasks organised as a task graph and dynamic and irregular task tree [17, 82].

Work stealing, as a method for efficient load balancing, has been explored and used in different contexts. The seminal work [9] considers a shared memory setting where tasks are stolen when the deque of a processor becomes empty. This work is used in Cilk [86]. Moreover, work stealing has been shown to exhibit good cache locality [1] and to be stable [7].

Recently and aimed at distributed memory machines, work on the X10 [18] programming language presented XWS [23], the X10 Work Stealing framework. XWS extends the Cilk work-stealing framework to include several features to implement graph algorithms, global termination detection, phased computation and more. Directly related, in [92] the authors introduce idempotent work stealing for tasks which are execute *at least once* instead of *exactly* once.

Using the UTS benchmark as a representative of unbalanced computations that require dynamic load balancing has been explored for different programming models, exploring their main features and devising suitable techniques that match the programming model. In [35] and [36], dynamic load balancing using message passing (MPI) is examined using two approaches (work stealing and work sharing). An UPC [24] implementation of the UTS benchmark is presented and evaluated in [107]. Also following a PGAS approach, using ARMCI [69] in this case, the work in [37] aims at the implications and performance of a design targeted at scale where the authors present the first demonstrations of scalable work stealing up to 8092 cores.

Finding scalable implementations of UTS and state space search problems in general, based on work stealing, remains a topic of intensive research and researchers continue to improve methods to deal with large scale computations and the hierarchical setup of current systems [101, 116].

4.3 Unbalanced Tree Search

The Unbalanced Tree Search (UTS) benchmark was designed to represent applications requiring substantial dynamic load balancing. The problem is rather simple: the parallel exploration of an unbalanced tree, by counting the number of nodes in an implicitly constructed tree that is parametrised in shape, depth, size and imbalance. Applications that fit this pattern include many search and optimisation problems that must search through a large state space of unknown or unpredictable structure.

The tree is implicitly generated where each node in the tree can be generated by the description of its parent. Each node in the tree is represented by a node descriptor which is the result of applying the SHA-1 secure hash algorithm to the descriptor of the parent of the node together with the child index of the node. With this generation method, UTS defines different tree types that represent different search types or problems and different load imbalance scenarios.

One interesting point about UTS is the different implementations available. There are MPI implementations (different approaches), UPC, SHMEM, OpenMP and more. And all the implementations are optimised to take advantage of the features of each programming model, creating an interesting comparison point for new implementations.

4.4 Dynamic Load Balancing with GPI

The UTS benchmark requires an asynchronous and dynamic load balancing strategy and we chose a work stealing strategy to address this.

Work stealing is a relatively simple algorithm. It is triggered every time a thread runs out of local work. An application taking advantage of a work stealing algorithm usually enters the following states:

Working
: While a thread has work, it keeps itself busy. In the case of UTS that translates to visiting nodes, generating child nodes and add them to the work list.

Work stealing
: When the work is all processed and the thread will fall into an idle state, it looks for a *victim* to steal work from and if it finds a potential one, it performs a steal. How the search for a *victim* and the actual steal operation are performed is implementation dependent.

Termination detection
: If work stealing fails that is, no *victim* thus no work is found, the thread enters termination detection. Termination detection is a topic in distributed computing *per se* and several algorithms exist.

The GPI implementation of the UTS benchmark [88] focuses therefore on the work stealing and termination detection stages. We leveraged the previous work on UTS with MPI and other implementations, taking them as a starting point for our own implementation.

A common aspect of all the implementations is the use of a data structure that is partitioned into two regions: private and shared. The private region holds the work of a worker thread and this is not available to other threads while the shared region holds the work available to be stolen. The GPI implementation uses a similar implementation of this data layout. Because we wanted to take advantage of GPI, we implemented the data structure to use the global memory of GPI. This allows global availability of the data and meta-data and direct access to it. Moreover and because the global memory of GPI is already available from the start of the application, the data structure operations (*e.g.,* add, remove) become simpler and less costly since no calls to the allocator (*e.g.,* malloc) are made and operations are reduced to the manipulation of GPI global memory offsets. Figure 4.4.1 depicts the organisation of the data structure and its placement on the global memory of GPI.

A thread adds and removes work packages (nodes) to its private region. This translates to a simple movement of the head since there is no synchronisation on this private region. When the private area hits a parametrised threshold, it releases a chunk of work to its shared region. This translates to a simple movement of the split pointer towards the head. A thread can also re-acquire chunks of work from its shared region when it exhausts the

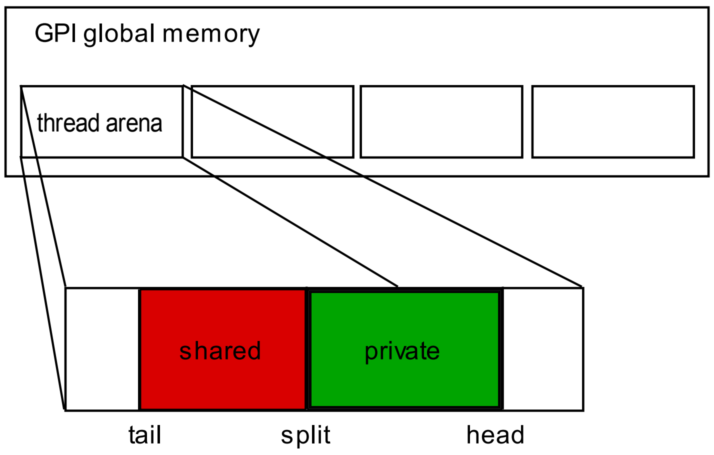


Figure 4.4.1: Data structure and GPI memory placement

work on its private region, by moving the split pointer towards the tail. Finally, when a thread performs a steal, it does it at the *victim*'s tail and moves it towards the head. Since it is a shared region, some mutual exclusion mechanism is required.

From a single thread point of view, the whole program control structure follows **Algorithm 2**.

Algorithm 2 Program control structure

```
    while $\neg done$ do
        while there is work do
            consume work
            generate (if that is the case) new work and save it
5:          share some work if there is a surplus on the private area
        end while
        if there is work on the shared region then              ▷ Step 1: re-acquire
            re-acquire it and go back to work
        end if
10:     if local steal is successful then                        ▷ Step 2: local steal
            go back to work
        end if
        if remote steal is successful then                       ▷ Step 3: remote steal
            go back to work
15:     end if
        enter barrier and termination detection                  ▷ Step 4: termination detection
    end while
```

A GPI implementation usually has two levels: local and remote. The local level deals with how computation and communication is handled within the same node (intra-node). In **Algorithm 2**, it translates to steps 1 and 2. The remote level takes care of the case between nodes (inter-node) which translates to step 3 and most of step 4.

Each thread has its own arena organised as described above. For steps 0 and 1, each thread acts on the data structure placed on its own arena, adding, removing work and re-acquiring it when needed. When they run out of work, threads must look and steal work (Step 2) from other threads. The *thief* thread can and does peek the status of other threads and their arena and if the shared region has more than a chunk of work, it is a potential *victim*. Because a mutual exclusion mechanism is required, the *thief* thread locks the data structure (each data structure has one lock to access its shared region and pointers), makes sure the work is still available and modifies the tail of the *victim*. Finally, the lock is released and the *thief* can move the stolen work to its own private region. The choice of the *victim* and whether a local steal happens follows a rather simple heuristic: the *thief* circulates over all other threads and if a surplus of work is found it immediately takes that thread as its potential *victim*.

When a thread does not find any local work to steal it tries to steal from a remote thread (Step 3) and has to resort to remote work stealing.

4.4.1 Remote Work Stealing

Remote work stealing poses some extra challenges since there is the need to involve the network. In the search for the best solution for remote work stealing, we experimented with three options:

- Remote spinlocks
- Passive communication
- Request/Poll

Remote Spin-locks

The first idea was the use of **remote spinlocks**. The reasoning behind this idea was the objective of keeping work stealing with a *one-sided* semantics. As in the local work stealing case, mutual exclusion would be ensured and the *thief* thread would steal without any participation from the *victim*. To this end, we implemented an extension to GPI, named *GPI SpinLocks*, that would allows us to create and operate on spinlocks on a remote node.

One of the features provided by the Infiniband architecture are atomic operations (*e.g.,* compare_and_swap) and *GPI SpinLocks* was based on these operations to implement remote spinlocks. Hence, instead of a lock for local steals, each data structure would have a single *GPI Spin-Lock* used for local and remote steals.

The obvious drawback of this approach (and confirmed by preliminary tests) is that mutual exclusion for a local steal incurs in a much higher latency and is thus, much more costly since it has to be performed by the interconnect and not the CPU. Therefore *GPI SpinLocks* was implemented by differentiating between local and remote locking where remote locking used atomic operations of Infiniband and local locking used CPU atomic instructions.

Initial tests showed encouraging and correct results. But as the number of nodes was increased and more work stealing was required, increasing the contention on the locks, we observed inconsistency and incorrect results. We realised that, although we were using atomic operations, their atomicity was not being enforced. The reason for this is that, in current systems, it is not possible to have a PCI device and the CPU atomically modifying memory, at least without inconsistencies. Thus, *GPI SpinLocks* was not able to guarantee mutual exclusion.

Passive Communication

With the failure observed with remote spinlocks, the remote work stealing operation had to relax the meaning of a steal (a *thief* usually steals without the *victim* knowing it) and require the participation of the *victim*.

The first approach in this direction was to use passive communication of GPI. As presented in Section 2.3, passive communication has a two-sided communication semantic (send/receive) where the receiver side does not specify a sender and waits from messages from any node. This semantic would fit the requirements of remote work stealing, where a steal request could come from any remote node. However, the receiver side in passive communication blocks for incoming messages and this is undesired since threads should proceed with computation in case there is no stealing request. To cope with this, our approach was to have a dedicated thread to handle remote stealing requests. All the remote stealing requests targeted at one node would be handled by this *stealer* thread.

This approach showed acceptable preliminary results when running the UTS benchmark but not totally satisfactory. One relevant aspect observed was the time that a *thief* thread spent trying to steal remotely, particularly as the number of nodes and thus of steal requests was increased. A directly related point was, as our objective was execution at large scale, there was a high possibility that the *stealer* thread would become a real bottleneck. Moreover, the *stealer* thread is only responsible for overhead and performs no work. Hence there are two options: use $n - 1$ threads that perform work and have one core dedicated to remote work stealing; use n working threads and one *stealer* thread that needs to be scheduled by the OS for execution, interrupting a working thread. Both options are not optimal.

Request/Poll

In order to overcome some of the mentioned problems and (possible) bottlenecks, we implemented a third approach to the remote work stealing problem which revealed itself as the most efficient and scalable.

Our implementation applies a request/polling strategy: the *thief* requests and the *victim* polls and responds to requests. This ensures that the access to the *victim*'s shared region is atomic since the *victim* itself will perform it.

Each worker thread has the added responsibility of handling steal requests from other nodes which are directly targeted at it. Added to the normal program control structure, each thread polls for pending remote steal requests. If it finds a request, a reply to the *thief* is sent. The reply takes the form:

- a work chunk
- no surplus of work but the node still has private work
- no work at all.

If there is work available, it comes from the shared region and the *victim* performs a local steal to itself. The work chunk is reserved for the steal request and the *victim* issues a remote write directly to the *thief*'s private region. This communication is a non-blocking one-sided operation that is queued and off-loaded to the interconnect allowing the *victim* to immediately return to its normal work loop.

Responding that there is no work to steal but that there is private work allows the *thief* to better evaluate the status of the *victim*. As it will be mentioned below, this is useful for detecting termination.

From the *thief* perspective, the remote work steal takes two simple steps: find the *victim* and write the request in case work is found. To find the *victim*, the *thief* thread takes advantage of the one-sided **read** operations of GPI to read the status of the remote node. This is accomplished by reading some meta-data of all threads on the remote node and finding the one with surplus of work. Here again, the heuristic is simple: if one thread has surplus of work it becomes the potential *victim* and the request is written to it. The *victim* only sees a request on the very probable case of having surplus of work, diminishing the possibility of a negative answer from the *victim*. This reduces the communication overhead and the waiting time of the *thief* for a negative answer. If the whole *victim* node is out of work, the *thief* tries another potential node until it tried all nodes. The *thief* tries all nodes in a ring pattern, starting at the node where it performed the last steal and until it has tried all $n - 1$ nodes.

4.4.2 Termination Detection

The current implementation uses a simple termination detection algorithm. When a thread finds no work to steal then potentially it has reached the termination state. As with work stealing, the termination detection works at the two levels, local and remote.

The local termination detection level works as follows: when worker threads of each node are not able to find work to steal they enter a local cancel-able barrier which allows them to return to the stealing state in case new work is made available. All but the last thread of the node stay in this local barrier. The last thread on each node, acknowledging that the node is completely idle and no work was found, enters the remote level of termination detection.

The remote level of termination detection, global termination, is handled by one single thread. The reason for this is - using the current simple implementation - to avoid that all idle threads wildly keep looking for work and thus putting a high pressure on the interconnect.

The last thread keeps looking for the availability of work on remote nodes and trying to steal a chunk. And it knows if the remote nodes still have private work since the response from the remote node includes both situations. When this last thread realises that all nodes are out of work - they all responded with “there is no work at all” - it increases an atomic counter by one and waits until this atomic counter reaches the total number of nodes. The global atomic counters of GPI are used for this termination flag. As all nodes increase the termination flag, the last thread on each node responsible for global termination detection warns the other worker threads waiting on the barrier that termination has been reached and they can exit.

On the other hand, if the last thread waiting for termination detection finds and steals some remote work, it cancels the barrier where the other local threads are waiting making them return to the normal program loop.

4.4.3 Pre-fetch Work

Any application using GPI should exploit its capacity to overlap computation and communication. With that in mind, the remote steal operation is implemented with a split-phase non-blocking semantics where a request has two distinct and independent phases. In the first phase, the request is submitted and the function immediately returns (non-blocking) not waiting for the request completion - the calling worker thread is free to continue its execution with some other work. In the second phase, a request is then checked for completion.

We take advantage of this and implemented a work pre-fetch step in order to overlap the communication involved in remote steal response with the normal program structure.

The pre-fetch step is triggered if the two following conditions are met:

- the worker thread tries to acquire work from its shared region and the work left there is smaller than the defined chunk of work - the thread is running out of work.
- there is a remote steal imminent - the threads on the same node do not have enough shared work for a local steal.

If the conditions are met, one remote steal request is submitted solely to the neighbour node (the nodes are paired in a ring). When the thread actually runs out of work, it first checks if a pre-fetch request was issued and in that case, checks its completion and answer. If the answer was positive, the thread avoids trying to steal work from other threads and can immediately continue.

Although there is some small extra overhead (submitting the request, checking if a pre-fetch was issued, etc.), we observed a 95% success ratio between submitting a remote steal request and getting a positive response on the overall pre-fetch steps performed.

4.5 Experimental Results

In this section, we present experimental results obtained on the evaluation of our implementation.

All results were obtained on a system of up to 256 nodes where each node is equipped with a Intel Xeon X5670 CPU ("Westmere") running at 2.93GHz with 6 cores and 12MB of L3 cache. Each node has two of such processors providing 12 threads (Hyper-Threading is disabled) making up to 3072 threads on the whole system. The nodes are connected via a Mellanox MT26428 QDR (40 Gb/sec) Infiniband card.

The UTS problems used for the performance evaluation are two, representing two different types and sizes: a geometric tree of about 270 billion nodes and a binomial tree with size of about 300 billion nodes.

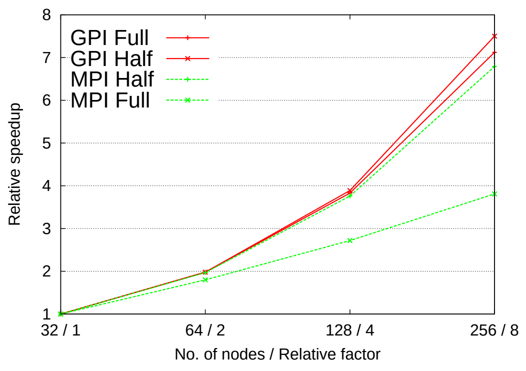
A binomial tree is a tree type where a node has m children with probability q and no children with probability *1 - q*, where q and m are parameters for this type of tree. A binomial tree imposes a great load balancing challenge because of its variation on sub-tree sizes and also since the expected work on all nodes is identical since they follow the same distribution. On the other hand, a geometric tree is depth-limited and has a branching factor that follows a geometric distribution based on a node's depth in the tree. Beyond the depth parameter (d), the tree is not allowed to grow.

We compare our implementation to the work stealing MPI implementation of UTS due to the general availability of MPI and since it is the standard for the development of parallel applications. The MPI implementation used was MVAPICH2 1.5.1.

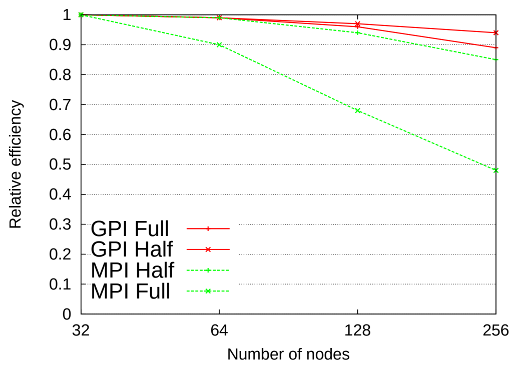
The performance evaluation was run on up to 3072 cores (threads) with 256 nodes. For each node setup, the execution made use of the Full node or Half node. Full node means that we use the maximum number of physical cores available (12) and Half means only half of those (6).

The reason behind this differentiated test is to observe the threads contention effects on the overall performance. Since we are using a mixed scheme for local and remote steals and the single node steals imply a locking mechanism, having a larger number of threads (cores) should present some contention effects.

Figure 4.5.3 depicts the performance for two kinds of trees: Figure 4.5.3a presents the results obtained for a Geometric tree of about 270 billion nodes and Figure 4.5.3b depicts

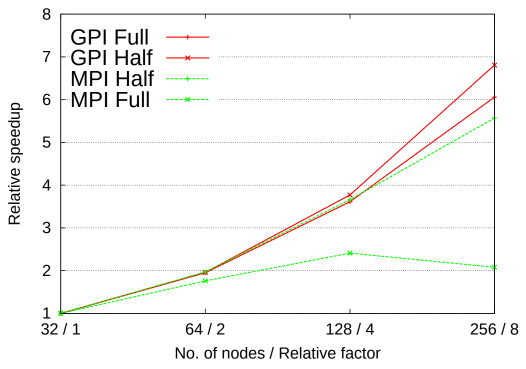


(a) Relative Speedup

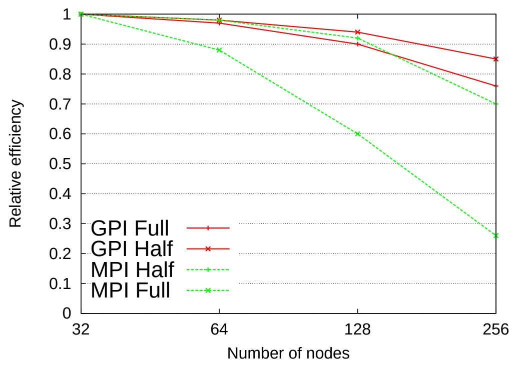


(b) Relative Efficiency

Figure 4.5.1: Scalability on Geometric Tree (≈270 billion nodes)

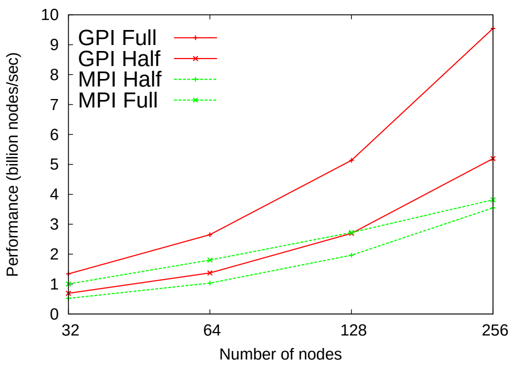


(a) Relative Speedup

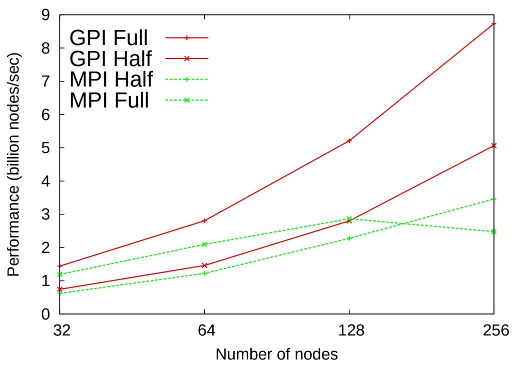


(b) Relative Efficiency

Figure 4.5.2: Scalability on Binomial Tree (≈300 billion nodes)



(a) Geometric Tree



(b) Binomial Tree

Figure 4.5.3: Performance (Billion nodes/second)

the largest problem - about 300 billion nodes. In this case, it is a Binomial tree which presents higher load balancing requirements.

In the geometric tree case, the GPI version scales well reaching a peak performance (shown in figure 4.5.3a) of around 9.5 billion nodes per second which represents a 2.5 times speedup factor over the MPI implementation (MPI best is 3.8 billion p/sec).

Comparing the two approaches in terms of cores used (Half or Full), we see that GPI behaves as expected that is, using more cores yields better performance. In fact and although the number of cores doubles between the Half and Full versions, using all cores attains 84% performance improvement over using half of the cores.

The MPI version suffers harder from the number of threads used, where using all cores only yields a 8% improvement over using half the cores in the largest number of nodes.

In the Binomial tree case(Figure 4.5.3b), the GPI version also scales well on both versions and the performance difference between using all or half of the cores is acceptable - the worst case, using 256 nodes, using all cores achieves a 72% improvement over using half of the cores. On the other hand, MPI has some problems at the largest core count. Using half of the available cores even yields better performance than using all cores.

Besides of performance in terms of the number of nodes processed, we observed the scalability of both types of trees in terms of speedup and parallel efficiency (Figure 4.5.1 and Figure 4.5.2).

For the Geometric tree, Figure 4.5.1a presents the obtained relative speedup to using 32 nodes as origin and, directly related, Figure 4.5.1b presents the relative efficiency when scaling the problem from 32 nodes (origin) to 256 nodes. Both GPI Half and GPI Full obtain high speedup (close to the maximum 8) with a relative efficiency above 89% in all cases. Worth noting is that using all cores yields slightly lower speedup factor and efficiency than just using half of the cores. This is an expected result since we have a fixed problem size and the efficiency should decrease because of the extra overhead involved. This observation is noteworthy since we are interested in measuring the difference and quantify that extra overhead. In this case, the values are acceptable as we observe a decrement in efficiency from 94% (GPI Half) to 89% (GPI Full) with the largest node count.

The MPI implementation has two faces: using half of the cores available (half) yields good results, with a speedup close to the GPI implementations and a relative efficiency above 85% in all cases. On the other hand, using all available cores (Full) and although obtaining higher performance, demonstrates scalability problems since the maximum obtained speedup only reaches a factor of 3.81 and with a rapidly decreasing efficiency that lowers to 48% at the largest core number. In the MPI case, increasing the node count for a fixed problem size decreases by a much larger margin the efficiency.

Figure 4.5.2 depicts the obtained results for the Binomial tree case. The speedup values are not so high as with the Geometric tree problem but that is acceptable since the Binomial tree problem imposes higher load balancing requirements. Nevertheless, we see a 6.06 maximum speedup factor (at 256 nodes) which represents a 76% efficiency when taking 32 nodes as a starting point.

The surprising behaviour of the MPI version is more evident when we observe the speedup (Figure 4.5.2a) and efficiency (Figure 4.5.2b with a lowest point of 26% of relative efficiency.

4.6 Conclusions

In this chapter we focused on parallel search and the problem of load balancing when parallelising such algorithms.

Starting from the observation that the load imbalance and the technique to overcome it are general to parallelisation of search, we designed a solution based on the GPI programming model that targets a recent large system (up to 3072 cores).

We used the UTS benchmark as a representative of that class of problems and evaluated two different kinds of workloads, geometric and binomial trees. In both cases the GPI version shows encouraging results and outperforms the MPI version by a maximum factor of 2.5 in terms of raw performance (number of nodes processed per second).

The obtained results lead us to conclude that we might be on the right track and GPI can give us a very good solution to the proposed problem. The obtained knowledge and implemented algorithms will be useful and serve as a basis when we return to parallel constraint solving with similar parallelism requirements.

Chapter 5

MaCS

This chapter presents the details of MaCS (*'Max'*), the Massively parallel Constraint Solver. It presents the elements and architecture of MaCS and details its main driving forces, the MaCS worker. Leveraging from the work from the previous chapter, this chapter presents constraint solving in MaCS with focus on load balancing and MaCS' work stealing implementation with GPI.

Finally, it presents and discusses the experimental evaluation performed with MaCS on up to 512 cores, providing a detailed analysis on its parallel behaviour.

5.1 Introduction

As we proposed to explore declarative models as a form to cope with the trend towards parallel systems, constraint programming appears as one approach with potential for parallel execution. From the possible options for parallelisation in constraint solving, the search component of a constraint solver is the one that shows more potential. First, because it is a part where a great number of cycles is spent and second, because high scalability is possible in parallel search.

In Chapter 4, we digressed into parallel tree search as a general method to solve different problems and emphasised the main challenge when it comes to parallel execution: load balancing. We designed and presented a dynamic load balancing scheme based on GPI that showed encouraging results in terms of scalability. Given the observation on the orthogonality between the load balancing scheme and the application at hand, we propose to experiment with the same scheme and apply it to parallel constraint solving, under the hypothesis that it could give us similarly good results. MaCS, herein presented, is the result of this endeavour.

MaCS is a parallel complete constraint solver based on GPI. It is a fork from PaCCS, a recently presented parallel constraint solver with some basis from AJACS [40, 41]. PaCCS was designed from scratch, with parallel execution on a network of multiprocessors in mind and exhibits good scalability on different systems.

The main objective of MaCS is to provide an efficient and scalable constraint solver that takes advantage of recent hardware and of the characteristics of GPI. It aims at large scale parallelism, exploiting the declarative nature of constraint programming to allow users to benefit from large and recent parallel systems. On the other hand, MaCS provides yet another use-case for a study on GPI and its programming model and a verification of its adequacy when implementing parallel constraint solvers targeted at large scale.

5.1.1 PaCCS

PaCCS is a recent and state-of-the-art parallel constraint solver which proved scalable performance on different problems.

PaCCS' main and distinguishing features are:

Distributed parallel solver PaCCS was designed for parallel execution on a network of multiprocessors.

No master process No process has the role of coordinating the other processes. There is only a distinguished process which initiates the search, collects solutions, detects termination and returns answers.

Workers steal directly Co-located workers do not interact to share work. Instead work is stolen from a co-worker by the worker who requires additional work.

All work is shareable The search process embodies the creation of immediately shareable work, usable by other workers.

Multiple search space splitting points PaCCS splits the search space twice before beginning with the search process. First, among teams (processes) and then among the workers of a team.

From all of the distinguishing features of PaCCS we emphasise, due to its relevance, the fourth feature, that all work is shareable. This allows the processing of work independently, without having to maintain any global state of the process. In terms of scalability, this is very important.

PaCCS is the foundation of MaCS and therefore many components are shared with modifications where appropriate. On the other hand, MaCS departs from this design, adopting GPI, its programming model and advantages, and implementing an alternative mechanism to keep load balancing when executed in parallel.

5.2 Execution Model

A MaCS program is a model of a problem where the user defines the variables, their domains, and the constraints of the problem. After these definitions, the *solver* is invoked to look for solutions.

Following the SPMD nature of GPI, the model is transparently loaded by all participating nodes when MaCS is initialised. It does not include any references to the parallel system and thus requires no user-awareness of the underlying implementation. Executing a MaCS program on a single core or on a large distributed system is, from this point of view, exactly the same.

When the solving procedure is invoked, it is up to the underlying implementation to take care of the parallelisation.

5.3 Elements

The basic elements for a problem representation in MaCS are the *variable*, the *constraint* and the *(domain) store*, following directly the definition of a CSP. These elements are global to the solver and visible by all workers. Again, since GPI has an SPMD execution model, the elements are available to all participating workers from the point the MaCS program is loaded and defined.

For each variable from the problem definition, there is a *variable* object which includes information about that variable:

index
: A unique identifier of the variable that is used as index within the store of domains.

connections
: The number of variables sharing a constraint.

constraints
: The constraints which the variable appears in.

Similarly, a constraint is represented with the following information:

index
: A unique identifier of the constraint.

variables
: The variables involved in the constraint.

constants
: The constants involved in the constraints, where applicable.

number of constants

The number of constants involved in the constraints.

A central element is the *store*. The store represents the set of variables' domains of the CSP. Each variable's domain is implemented as a fixed-size bitmap. A store is self-contained and implemented as a continuous region of memory where each cell is the bitmap of the domain of each variable. This turns a store into a relocatable object that can be moved or copied to other memory regions. This self-contained representation is essential in a distributed setting and a key point in MaCS' parallel performance.

From a different point of view, a store is also the unit of work where computation happens and that shapes the solving process. Consequently, it is the piece of data that is communicated between workers in order to keep the whole computation balanced.

5.4 Architecture

MaCS has a different architecture from that of PaCCS, a consequence of the focus on the use of GPI and the previous results with parallel search and the UTS benchmark.

In MaCS, the main and single entity is that of a *worker*. Each worker maintains a pool of work from which we can retrieve work packages when the current one is exhausted. There is no other entity to control and manage communication (controller). This is one of the points where MaCS departs from PaCCS.

The architecture of MaCS mimics directly that of GPI in which there is a notion of locality in terms of data. In GPI, the underlying system is viewed as a set of nodes where each node is composed of one or more cores. Cores in a node are closer to each other and communicate through shared memory whereas cores in another node are remote and communication and access to data is done through the DMA interconnect. In MaCS, workers on the same node are closer to each other when compared to those on a remote node and the view of data is thus different. Hence, it is a natural choice to treat the local and remote cases differently.

5.5 Worker

A MaCS worker (or simply worker) is at the core of the MaCS solver and it is its main driving force. A worker is a completely self-maintained entity which tries, as much as possible, to be independent and responsible for all actions required to find solutions. This includes generating and processing new work, retrieving new work from its pool or from other workers' pool and detecting termination.

The mechanism to retrieve new work from other workers is called *work stealing*, which each worker tries to perform independently, as much as possible.

5.5.1 Work Pool

A central aspect of the architecture of MaCS is the work pool. Each worker has one pool from where work is retrieved, new work packages are inserted and from where other workers can steal, in order to maintain load balance. Hence, its implementation is critical because it determines the amount of work available to other workers and must be efficient since it is the central data structure that stores the tasks to be performed by the worker.

As mentioned before, we wanted to leverage our previous work with UTS and general parallel tree search (Chapter 4). Since that solution has proved scalable to implement dynamic load balancing with work stealing, the work pool uses the same data structure used in that work.

Figure 5.5.1 depicts the work pool again, now in a closer view.

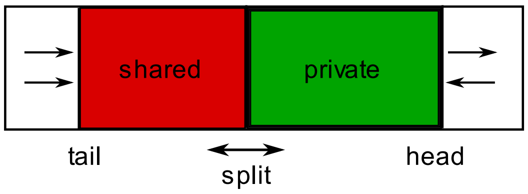


Figure 5.5.1: Work pool

Recall that the pool is divided into two regions, the shared and private regions. The private region is only accessed by the worker who is the owner of the pool whereas the shared region can be accessed by other workers (*e.g.,* to steal work). The private region is between the *head* and *split* pointers and, as illustrated by the arrows, grows and shrinks by updating the *head* and *split* pointers. Both pointers can be moved back and forth. The shared region is between the *split* and *tail* pointers. It can also grow and shrink. Shrinking happens by updating the *tail* pointer or the *split* pointer but growing only happens when the *split* pointer is updated towards the *head.*

The reason to divide the pool in two regions is that it allows us to have an efficient mechanism to add and retrieve from the pool, a very frequent operation. Both operations can, if there is work, be performed without mutual exclusion or conditional statements since they only require the manipulation of the *head* pointer which, as it is known to be private, is only manipulated by the worker owning the pool.

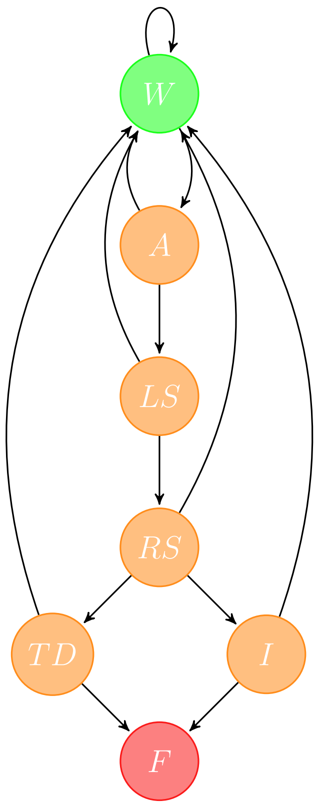
Clearly, the access to the shared region must be synchronised to ensure correctness. Moreover, each work package can lead to the generation of further work and if two workers take the same work package, a redundant work generation (and processing) might happen. Each work pool has a lock that is used when the shared region is to be updated *i.e.,* updating the *tail* or *split* pointers.

The *split* pointer divides the two regions and has to be periodically updated, keeping a good balance between both regions. Particularly important is to maintain enough work in the shared region for idle workers to take. The pointer is moved towards the *head* when there is the need to share work, increasing the shared region. On the other hand, it is moved towards the *tail* when the private region is empty and the shared region still has available work.

The work pools are placed in the GPI global memory, retaining the same layout as depicted in Chapter 4 in Figure 4.4.1.

5.5.2 Worker States

Being responsible for all steps of computation, the worker transitions between several states illustrated by Figure 5.5.2.



State	**Name**
W	Working
A	Acquire
LS	Local Steal
RS	Remote Steal
I	Idle
TD	Termination Detection
F	Finish

Figure 5.5.2: Worker state diagram

A worker is on the **W**orking state as long as it has work in the private region of its own pool. When all work is exhausted, the worker tries to **A**cquire some work from its shared region of the pool. If it succeeds (there is work available), it returns to the **W**orking state. If it fails, the worker transitions to the **L**ocal **S**teal state where it tries to steal from workers on the same node. In case of success, it returns to the **W**orking state. If it fails, a **R**emote **S**teal is tried. The worker searches for a remote node from where to steal and if it becomes a positive feedback, it returns again to the **W**orking state. In case it is not possible to get new work, the worker transitions to one of two states: **I**dle or **T**ermination **D**etection. The decision to which state the worker should transition to depends on the number of idle workers on the same node. If all other workers are already idle, the last worker transitions to the **T**ermination **D**etection state and global termination detection is initiated. Otherwise, the worker transitions to the **I**dle state, sitting idle until some new work is made available or termination is detected by the last worker. In both states (**TD, I**) and if new work is made available, workers return to the **W**orking state. If termination is detected, workers transition to the **F**inish final state.

5.6 Constraint Solving

Until now, we have treated a MaCS worker as a very general entity that processes work packages, generates new work and attempts to maintain load balance by means of work stealing. This general view must be refined for the purpose of a MaCS worker, namely solving CSPs through propagation and search.

Constraint solving in MaCS is implemented by interleaving propagation with variable instantiation. Variable instantiation corresponds to the search component of MaCS. When a variable is selected for instantiation, its domain is split into two parts creating two branches of the search tree. One part corresponds to the search space where the domain of the selected variable is a singleton *i.e.,* only with the selected value. The other part corresponds to the search space with the complement of the domain of the selected variable that is, without the value with which the variable was instantiated. Recall from Section 3.3, this procedure is called *labelling*.

The part where the selected variable is singleton becomes the new current store whereas its complement is added to the work pool. The change in the domain of the variable (now singleton) is propagated and if successfully, the whole process repeats until a solution is found or propagation fails. If the propagation fails, the current store is discarded and a new store is retrieved from the pool or from some other worker, until no more work is available.

The process of constraint solving is the main loop of a worker as presented in Algorithm 3.

Algorithm 3 Worker main loop

```
1: function WORKER(Store s)
2:     while $var \leftarrow var\_selection(s)$ do          ▷ Select variable to instantiate
3:         $s \leftarrow branch\_add\_pool(var, pool)$      ▷ Choose value and add branch to pool
4:         while $propagation(s, var) = FAIL$ do
5:             $s \leftarrow restore(pool)$                 ▷ Restore from failure
6:         end while
7:     end while
8:     return $solution(s)$                                 ▷ Return solution in store $s$
9: end function
```

When dealing with optimisation problems, the worker main loop is extended to deal with the bound variable. Algorithm 4 shows the worker main loop extended with lines 5 and 7 for the case of an optimisation problem. The difference brought by the extended procedure is that, when retrieving a new store from the work pool, the domain of the bound variable is updated by removing the values which are worse than the current global bound value, pruning unnecessary parts of the search tree. If necessary, the change in the domain of the variable is propagated to the remaining variables.

Algorithm 4 Worker main loop (with optimisation)

```
1: function WORKER(Store s)
2:     while $var \leftarrow var\_selection(s)$ do          ▷ Select variable to instantiate
3:         $s \leftarrow branch\_add\_pool(var, pool)$      ▷ Choose value and add branch to pool
4:         while $propagation(s, var) = FAIL$ do
5:             repeat
6:                 $s \leftarrow restore(pool)$             ▷ Restore from failure
7:             until $\neg optimisation \vee bound\_propagate(s) \neq FAIL$
8:         end while
9:     end while
10:    return $solution(s)$                                 ▷ Return solution in store $s$
11: end function
```

Intuitively, it can be observed how the main loop of a worker relates to the general view of a worker. When a variable is instantiated and a domain split happens, creating two branches of the search tree, the worker is generating new work. By taking one of the branches and propagating the change, the worker is performing work or in other words, computation on a work package. If the propagation succeeds, the worker continues that path and continues in a **Working** state. If it fails, the worker exhausted the work package and must retrieve more work from the pool or other workers to avoid idleness. The *restore* procedure represents the whole chain of transitions that a worker performs to tentatively return to a Working state and before reaching an idleness or final state.

5.6.1 Variable instantiation and Splitting

A worker, having a store to process, starts by instantiating a variable with a particular value. This includes selecting a variable, choosing a value from its current domain and splitting it accordingly. Variable instantiation has a decisive role in defining the shape of the search tree and hence on performance.

Figure 5.6.1 helps to visualise the process of instantiation and domain splitting (*labelling*). The variable x with domain $\{1, 2, 3\}$ is instantiated with 1. This results in the domain being split in two branches where in one branch (left) the variable x has the singleton domain $\{1\}$ and on the other branch (right), the variable x has the complement domain $\{2, 3\}$.

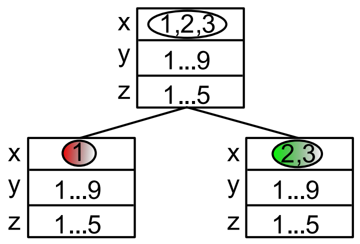


Figure 5.6.1: Domain splitting example

After creating both branches of the search tree, a worker takes the left branch and adds the right branch to its pool, to be processed later. Taking the left branch entails propagating the change in the domain of the instantiated variable to the other variables. If propagation is successful, a new variable must be instantiated and the whole procedure starts again.

Since, at this point, more work (right branch) is added to the worker pool , the worker must ensure that it is exposing enough work for other peer workers to steal. Hence, it is at this point that work is released from the private to the shared region of the worker pool. Clearly, it is not always necessary to release work on each addition to the pool and thus, releasing work is parametrised by an adjustable coefficient on the size of the private region. The major drawback of this extra step is the introduction of overhead not related to the solving process but which should pay off in terms of parallel efficiency. In fact, this parameter is of crucial relevance as it will become clearer with the experimental evaluation.

5.6.2 Propagation

Propagation in MaCS follows that of PaCCS. It is rule-based and each constraint has two *propagators*. The first is used initially when starting the search for a solution, filtering out values from the domains of variables in the scope of the constraint.

The other propagator is used during the rest of the process of searching for a solution. Every time the domain of a variable in the scope of the constraint changes, that change is propagated onto the domains of the other variables in the scope.

5.6.3 Load Balancing

Propagation may fail at some point, invalidating the current store and search path. A worker therefore invokes the restore procedure in order to obtain a new store to work on. The restore procedure encompasses all the mechanisms to keep a worker as busy as possible and is hence responsible for the whole load balance of the parallel computation.

The first step of restoring a store is to acquire, if possible, work from the worker's own pool or in other words, another store. Already this acquire operation can contribute to a good or bad load balance.

However, if the work pool is empty and no new store can be retrieved, the restore procedure must resort to work stealing from some other worker.

Recall that the work pool (Figure 5.5.1) is divided in two regions: private and shared. The restore procedure tries to first retrieve a new store from the private region of the work pool. In case this region is empty, the shared region will be inspected for the availability of work. If work is available in this shared region, some work is acquired, shrinking the shared region.

The work stealing approach used in MaCS distinguishes local and remote work stealing. This is, as previously noted, because it elegantly maps to the GPI programming model and our target systems.

Local Work Stealing

When a worker has no more work in its work pool, the first measure it applies is to steal work from a worker on the same node. The worker trying to steal becomes the *thief* and the worker where work is to be stolen becomes the *victim*. Each thief can access the pool of the victim without disturbing it and the stealing operation is entirely driven by the thief.

Local stealing only happens at the shared region of the pool of the victim. If this region is empty, a local steal cannot succeed even if the victim has any work in its private region.

One important aspect is the choice of the *victim*. Different heuristics can be used for this: a random victim, the victim with more work available, the next victim according to each worker's identifier, etc. Currently, MaCS includes two different options for selecting a victim: *greedy* and *max_steal*. With the *greedy* variant, the first victim found with available work is chosen. The *max_steal* variant is less eager but more costly: the thief checks all $n - 1$ possible victims and chooses the one with the largest shared region.

This local work stealing procedure is shown in Algorithm 5.

Algorithm 5 Local Work Stealing

```
function STEAL_FROM(v, k)
    $result \leftarrow FALSE$
    $pollLock(v)$
    if $poolSharedDepth(v) \geq k$ then                ▷ Re-check state of victim pool
        $result \leftarrow TRUE$
        $poolUpdateTail(v)$                            ▷ Shrink shared region
    end if
    $poolUnlock(v)$
    if result == TRUE then
        $store \leftarrow poolRemoveTail(v)$
    end if
    return $result$
end function

function LOCAL_STEAL
    $goodSteal \leftarrow FALSE$
    $victimId \leftarrow findVictim(k)$
    while $victimId \neq -1 \wedge \neg goodSteal$ do  ▷ While no victim found
        $goodSteal = steal\_from(victimId)$            ▷ Try to steal from victim
        if $goodSteal == FALSE$ then
            $victimId \leftarrow findvictim(k)$
        end if
    end while
    return $victimId$
end function
```

The *findVictim* procedure in Algorithm 5 points to the victim selection heuristic (*greedy* or *max_steal*). Both implemented heuristics are very simple as shown by Algorithms 6 and 7. Furthermore, more complex victim selection heuristics could be implemented.

Algorithm 6 Greedy Victim Selection

```
function GREEDY_STEAL(k)
    for $i = 0 \rightarrow TOTAL\_NUM\_WORKERS$ do
        $v \leftarrow (workerID + i) \mod TOTAL\_NUM\_WORKERS$
        if $poolSharedDepth(v) \geq k$ then
            return $v$
        end if
    end for
    return $-1$
end function
```

Algorithm 7 Max_Steal Victim Selection

```
1: function MAX_STEAL(k)
2:     $maxW \leftarrow 0$
3:     $maxV \leftarrow -1$
4:     for $i = 0 \rightarrow TOTAL\_NUM\_WORKERS$ do
5:         $v \leftarrow (workerID + i) \mod TOTAL\_NUM\_WORKERS$
6:         $avail = poolSharedDepth(v) \geq k$
7:         if $avail > maxW$ then
8:             $maxV \leftarrow v$
9:             $maxW \leftarrow avail$
10:        end if
11:    end for
12:    return $maxV$
13: end function
```

Remote Work Stealing

The last step a worker performs before turning to idleness it to try to steal remotely, from workers on a different node. As mentioned, the stealing operation should disturb the victim as little as possible. Although mainly driven by the worker that needs to find work (*thief*), the remote steal operation needs the some cooperation from the remote worker (*victim*).

The first step to a remote steal is to decide where to steal from that is, the remote node to steal from. The choice of the potential victim can, as in the local case, be subject to different heuristics. Once the potential victim is selected, the thief must look for work on that node. Instead of sending a request message for work, the thief can simply read the state of the remote node *i.e.,* the state of all worker pools on that node and choose the one which has a surplus of work, since all work pools are in global memory and thus accessible to a GPI read operation. To read the pool state of a worker means accessing the meta-data of the work pool and see if its shared region has work packages available to steal. Only when the thief has found a worker which has work in its shared region of the pool, is an actual request written to that worker. This reduces the probability of requests which yield a negative answer (failed steals) since the request is only sent to a worker that has a surplus of work, when the read was performed. The requirement to write an explicit request is due to the fact that stealing work must be atomic, to avoid redundant work. If two workers could steal the same store from a pool, this store would be processed twice and generate the same sub-problems. After sending the request, the thief will wait for a response from the victim.

To be able to respond to work requests, each worker must poll for these requests, introducing an extra step in the worker main loop. When a worker finds a request for a remote steal and has available work, part of it is reserved for the steal. The reservation is simply a shrink of the shared region of the work pool, moving the tail towards the head and, as in the case of a local steal, the operation is atomic. The victim queues a request to write the reserved work directly to the worker that issued the request and returns to its normal work loop. The objective here is to overlap that communication with the computation of the victim worker, reducing the total overhead of polling and communication. Moreover, the queued write request is performed in-place *i.e.,* directly to the head of the thief's pool, avoiding any intermediate copies.

Although a thief only writes a request to a victim when a surplus of work is visible, sometimes this request might yield a failed steal. When the victim acknowledges the request, it can happen that its pool no longer has a surplus of work (e.g. it was consumed or locally stolen). To avoid this situation and reduce the number of failed steals, in MaCS, the victim tries to fulfil that request. Since it does not have a surplus of work, it performs a reservation of work from some other local worker which has a surplus of work and writes that work back to the thief. This is possible since, locally, all workers can access each others pool and these are placed on the global memory of GPI. Not only can a worker access and communicate work from other pool, it can do so without much overhead except that coming from finding the worker with a surplus of work.

The MaCS polling approach increases the parallel overhead. On problems that are more regular and require less movement of work among nodes, polling is an unnecessary and excessive step. To cope with this, we implemented a dynamic polling strategy to mitigate the effect in such cases. A polling interval is introduced which grows and shrinks according to the number of successful poll operations: if the poll fails, the polling interval grows and increases the time between poll operations and hence reducing their total number and their negative effect; if a poll succeeds, the opposite happens and the polling interval becomes more frequent.

5.6.4 Termination Detection

The final task of a worker is to detect termination, realising that it should stop working and exit the solving procedure.

In MaCS, we extended the implementation for termination detection used in the UTS benchmark since there is a difference between searching for one or all solutions. In UTS there was a fixed amount of work to be performed and hence termination was achieved as soon as all workers realised that all nodes had been processed. In MaCS and if we are counting solutions or searching for the optimal solution, this pattern holds as the full search space must be explored. However, when the objective is to find a single solution, termination must be detected at this point even if workers still have work.

To cope with both aforementioned cases, we distinguish between active and passive termination. When the solving process is searching for a single solution, the workers must *actively* detect termination *i.e.,* even if workers are busy, termination must be detected within the normal work loop. The drawback is the generation of extra overhead for the detection. On the other hand, if the whole search space must be explored, the workers can detect termination *passively i.e.,* when they have no work and are idle.

With respect to load balancing, termination detection is dependent on whether it is local or remote. In the local case, termination is detected by a local flag set by a single thread. When this flag is set, termination is detected by all workers. Setting this flag to $TRUE$ is the responsibility of a single thread which is also responsible for global termination that is, termination of all remote nodes.

Global (remote) termination differs from the version implemented previously. In MaCS, the algorithm (similar to [34, 106]) uses a coloured token that is passed to neighbour nodes until all nodes have received the *red* token, signalling termination. As already mentioned, one thread is responsible for this and as global termination has been detected, the local flag for termination can be set, alerting the remaining threads.

5.7 Experimental Evaluation

In this section, we present the results obtained with MaCS for different benchmark situations. The obtained results are compared with those of PaCCS, the foundation of MaCS and the basis for comparison as a state-of-the-art constraint solver targeted at parallel execution.

The experiments were conducted on a cluster system where each node includes a dual Intel Xeon 5148LV ("Woodcrest") (*i.e.,* 4 CPUs per node) with 8 GB of RAM. The full system is composed of 620 cores connected with Infiniband (DDR). Since we aim at large scale, we performed our experiments on the system using up to 512 cores.

5.7.1 The Problems

To assess the implementation of MaCS, we used problems with different characteristics. The chosen instances guarantee an adequate problem size to be executed on a large number of cores. The problems are described in the next sections.

N-Queens

The N-Queens problem is a classical CSP example. Although simple, N-Queens is compute intensive and a typical problem used for benchmarks.

The problem consists of placing N queens on a NxN chessboard so that it is not possible for a queen to attack any other one on the board. This means no pair of queens can share any row, a column or a diagonal and that these are our constraints.

Golomb Ruler

A *Golomb ruler* of size M is a ruler with M marks placed in such a way that the distance between any two marks is different from the distance between any other two marks. It is a hard problem (NP-complete) for which an algorithm to find the optimal solution for $M \geq 24$ is not yet known. This problem has practical applications in sensor placements for x-ray crystallography and radio astronomy [8].

Langford's Problem

Langford's Problem consists in arranging k sets of the integers from 1 to n, L(k,n), so that each occurrence of integer i is i integers on from its last occurrence. Some instances of the problem have no solution whereas other have many solutions.

Consider the instance $L(2,4)$ depicted by Figure 5.7.1: the set of integers($\{4,1,3,1,2,4,3,2\}$ are arranged so that between each integer n there are n other integers.

4	1	3	1	2	4	3	2

Figure 5.7.1: Example: Langford's problem instance $L(2,4)$

Quadratic Assignment Problem (QAP)

The Quadratic Assignment Problem (QAP) is an NP-hard problem and a fundamental combinatorial optimisation problem. In this problem for a given set of n locations and n facilities, the objective is to assign each facility to a location, with a minimal cost. The cost of each possible assignment is the result of multiplying the prescribed flow between each pair of facilities by the distance between their assigned locations, and sum over all the pairs.

A formal mathematical definition of the QAP can be written as follows:

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} D_{p(i)p(j)} \times F_{i,j}$$

where F and D are two n by n matrices. The element (i,j) of the flow matrix F represents the flow between facilities i and j and the element (i,j) of the distance matrix D represents the distance between location i and j. The vector p represents an assignment as a permutation where $p(j)$ is the location to which facility j is assigned.

5.7.2 Results

This section presents and discusses the results obtained for instances of the problems presented above.

We analyse each problem according to different aspects which help to explain the results obtained. These aspects are related to overhead and scalability, the work stealing mechanism and constraint solving.

We start by examining the execution with MaCS, to make sure the computation is well-balanced and to identify any possible bottlenecks. We detail the average overhead of workers *i.e.,* how much, from the total running time, does a worker spend in the working state and in the other different possible states (overhead). Another way to look at this is by measuring the performance according to the number of nodes processed per time unit. Ideally, the number of nodes processed per second should increase linearly with the number of cores used indicating that the overhead does not substantially increase.

We also measure the number of stealing operations, successful and failed, in order to get an impression of the imbalance and thus the load balancing requirements of each problem. A problem that requires a large number of steal operations, is more imbalanced than a problem with a low number. In the latter case, each worker often has enough work to progress without resorting to work stealing and the overhead required to maintain load balance becomes redundant, hindering performance.

From the perspective of the constraint solving process, MaCS also includes statistics which allow us to evaluate and characterise problems according to how much time is spent the different steps: propagation, splitting and restoring. Propagation and splitting (variable instantiation) are the interleaved steps of the constraint solving process whereas restoring corresponds to the retrieval of stores from the work pool, including steal (local or remote) operations.

Finally, scalability is measured by observing the obtained parallel speedup and parallel efficiency. We present the obtained results together with those of PaCCS: this allows us to position MaCS when compared to a state-of-the-art parallel solver.

N-Queens

The first problem to be evaluated is the N-Queens problem. We choose the instance where $n = 17$ and count the total number of solutions. This represents a problem with a considerable size to experiment in a larger scale.

Running this problem with MaCS built-in statistics provides a closer view of the computation and gives some hints at possible improvements. Figure 5.7.2 depicts how much time, on average, is spent by workers on each of the major states.

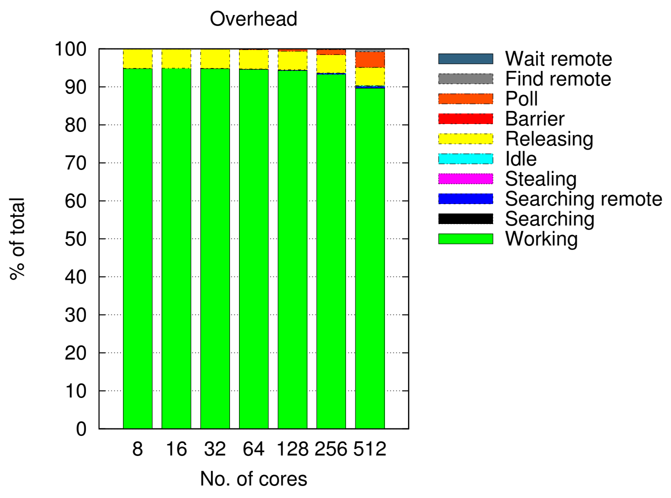


Figure 5.7.2: Working time and Overhead - Queens (17).

From Figure 5.7.2 we can see that workers are most of the time busy (Working state). But although most of the time is spent working, there is a considerable amount of time spent in *Releasing*, a state which refers to the time, within the main work loop, of releasing work on the work pool. Also, at a larger scale, the influence of *Polling* for remote requests constitutes another source of overhead although this is expected since there is a large number of workers and a growing number of steal operations. All other states (e.g. waiting for steals, idle) have a very low contribution in terms of overhead and are almost negligible.

Figure 5.7.3 presents performance in terms of number of stores processed per second. The performance of MaCS continues to increase as we grow the number of cores but is lower and even slightly diverging from the ideal case.

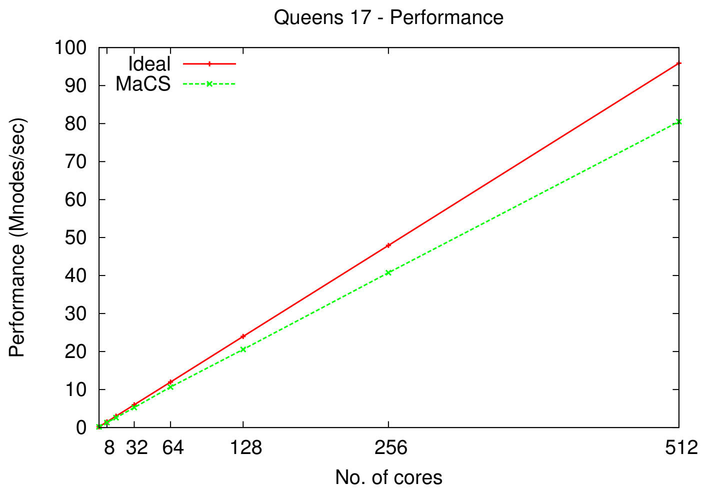


Figure 5.7.3: Performance (Million of nodes per second) - Queens (17).

To this performance aspect, it is also interesting to add how the processing of stores (nodes) is performed in this problem. In terms of representation, the N-Queens problem is rather small: 17 variables which represents a store size of 136 bytes. On the other hand, the total number of nodes processed is quite large (757914186), at a very high rate (e.g. around 40 million nodes per second with 256 cores). Which means that the necessary time to process each node is very small and that the generation and consumption of nodes also happens at a high rate.

In fact, looking at MaCS statistics and the time spent on the three steps of the solving procedure (propagation, splitting and searching) we observe the following distribution: propagation takes around 48%, splitting around 10% and restoring takes around 42% of the total time. This distribution is constant and independent of the number of cores used. The most noteworthy fact is that a large portion is spent on retrieving stores from the local pool (restoring) which also helps to explain the high overhead incurred by releasing work: releasing happens often and since processing a store is a fast operation, the overhead of releasing becomes more noticeable.

With respect to load balancing and work stealing, Table 5.7.1 presents the number of successful local and remote steals as well as the number of failures. The total number of attempts to steal (local and remote) is thus, the sum of these values (not presented). Note that the numbers refer to attempts to steal, that is, in case a worker realises it cannot steal (locally or remotely) because there is no work to steal, no steal is attempted and thus, does not count as failure or success.

Cores	Total Nodes	Local Steals				Remote Steals			
		Total	per core	Failed	Rate	Total	per core	Failed	Rate
8	757914186	888	111.00	6	0.67%	59	7.37	3	4.84%
16	757914186	3895	243.43	41	1.04%	409	25.56	38	8.50%
32	757914186	18360	573.75	250	1.34%	2307	72.09	204	8.12%
64	757914186	53936	842.75	785	1.43%	7308	114.18	635	7.99%
128	757914186	187097	1461.69	2924	1.54%	29008	226.62	2483	7.88%
256	757914186	451624	1764.15	7744	1.69%	75251	293.94	7135	8.66%
512	757914186	854633	1669.21	17170	1.97%	168859	329.80	21498	11.29%

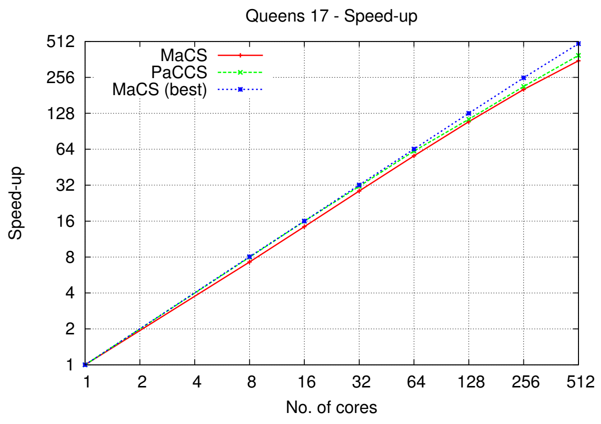
Table 5.7.1: Work Stealing Information - Queens (n=17).

Unsurprisingly, the number of steals (local and remote) increases as more cores are used although at different rates as the number of remote steals increases slightly faster.

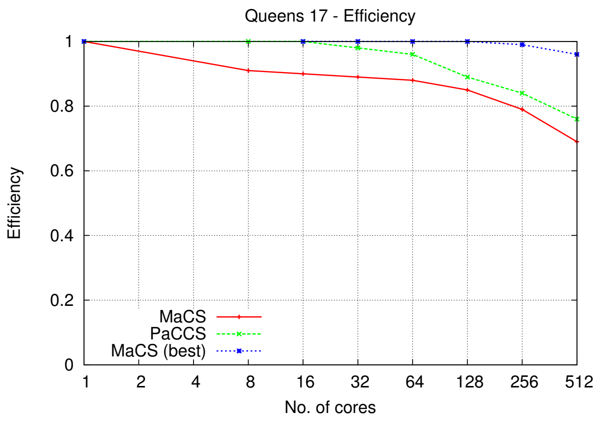
A more meaningful aspect is that the number of total steals is significantly low when compared with the total number of nodes processed, reflecting a lower requirement in terms of load balancing. Another important point is the relatively large number of failed steals, in particular of the remote steals. Failed remote steals incur in high overhead and are very detrimental to parallel efficiency.

The results we obtained reflect and are in accordance with the characteristics of the N-Queens problem *i.e.,* many solutions which grow fast with the number of queens and sub-search spaces of similar size and representing a balanced search space tree.

Figure 5.7.4 depicts the parallel speedup (Figure 5.7.4a) and parallel efficiency (Figure 5.7.4b) graphs of MaCS and PaCCS.



(a) Parallel Speedup



(b) Parallel efficiency

Figure 5.7.4: Scalability - Queens (17).

For the case of MaCS, Figure 5.7.4 shows both the default and best cases. As the default execution of MaCS leaves some room for improvement, we set to improve it based on the analysis of previous information on overhead, performance and load balancing. The best case corresponds to the results obtained after that analysis. More concretely, to reduce the constant overhead on the execution caused by the work pool maintenance, the work release interval is optimised for better performance.

Both MaCS (default) and PaCCS show good behaviour, scaling well as the number of cores is increased, but it is notorious - particularly in Figure 5.7.4b - that the default settings of MaCS are not as effective as they could. With eight cores (two nodes), the parallel efficiency drops considerably (to 91%), although less steeply after that. The overhead (Figure 5.7.2) is limiting better scalability. After improving MaCS execution based on the interpretation of previous data on overhead and load balancing, we observed almost linear speedups with a parallel efficiency of 96% with 512 cores. The improvement is simply based on the reduction of the number of (extraneous) release operations.

Golomb Ruler

The Golomb Ruler is the first evaluated optimisation problem. The observed results are for the 13-mark ($M = 13$) instance.

As with the N-Queens problem, we start by analysing how the overhead has an influence in the total solving time (Figure 5.7.5). Contrarily to the previous problem, for the Golomb Ruler problem, overhead remains low up to 256 cores. Nevertheless, as the number of employed cores and stealing operations grows, polling consumes more time from a worker's solving time.

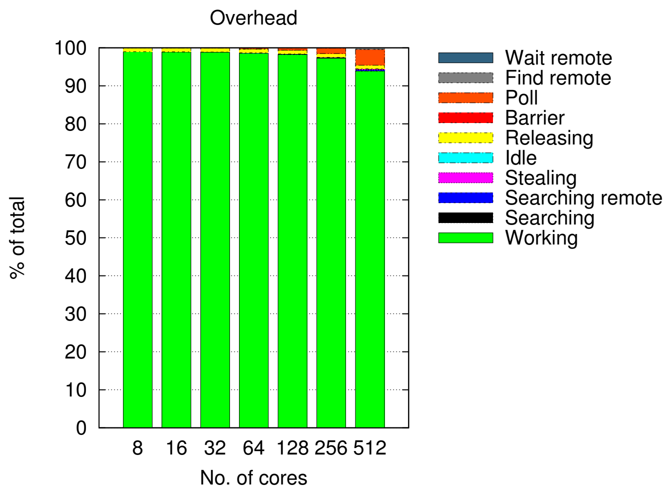


Figure 5.7.5: Working Time and Overhead - Golomb Ruler (n=13).

As we can see from Figure 5.7.5, the overhead only becomes more noticeable at 512 cores, where cores spend more time polling for remote steals. As for the overhead caused by other operations such as releasing work, the influence is low and workers are kept busy almost all the time.

When we observe the performance in terms of processed nodes as depicted by Figure 5.7.6, the obtained results are close to ideal (linear). These results confirm that the overhead is kept low.

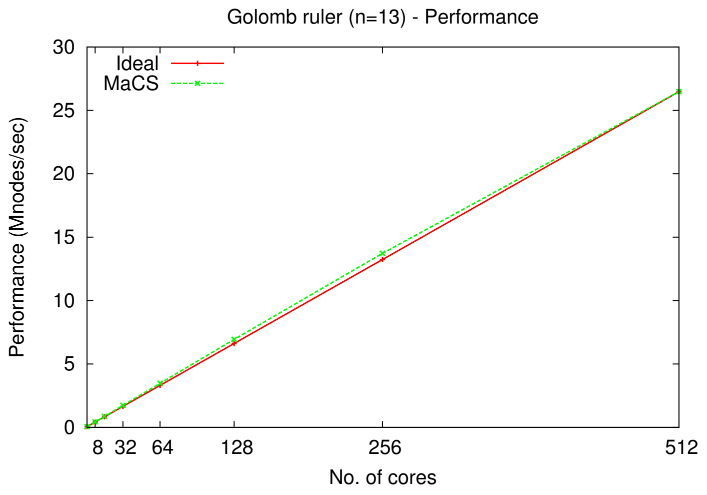


Figure 5.7.6: Performance (Million of nodes per second)- Golomb Ruler (n=13).

One aspect relates to the achieved performance values: in comparison with the N-Queens problem, this problem has a much lower rate of processed stores per second which indicates a higher processing time per store. In fact, in terms of constraint solving, MaCS spends on average 89% on propagation, 8% restoring and 3% on splitting.

From the perspective of work stealing and load balancing, Table 5.7.2 characterises the Golomb Ruler problem.

Cores	Total Nodes	Local Steals				Remote Steals			
		Total	per core	Failed	Rate	Total	per core	Failed	Rate
8	582527453	814	101.75	2	0.25%	49	6.12	2	3.92%
16	573035135	5299	331.23	17	0.32%	506	31.67	44	8.00%
32	630746327	22243	695.11	64	0.29%	2408	75.26	262	9.81%
64	722719476	109314	1708.04	353	0.32%	11504	179.75	851	6.89%
128	710208857	245152	1915.25	886	0.36%	29140	227.66	3220	9.95%
256	716886676	571054	2230.68	2407	0.42%	75035	293.11	10646	12.43%
512	763994075	1140119	2226.80	5428	0.47%	166404	325.01	35319	17.51%

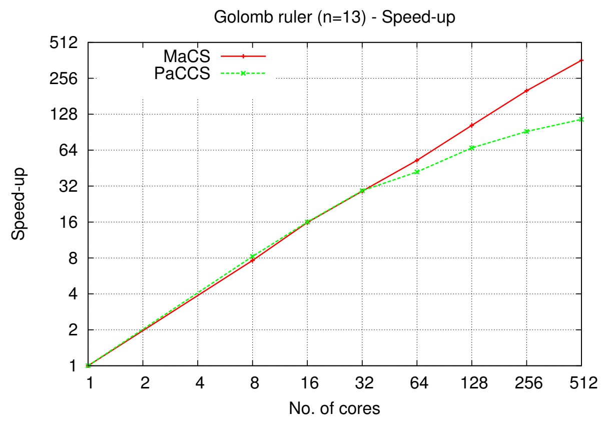
Table 5.7.2: Work Stealing Information - Golomb Ruler (n=13).

As expected, the number of steals (local and remote) also increases although with differences between local and remote steals. The number of local steals per core tends to stagnate whereas the number of remote steals keeps increasing. More noteworthy though, is that this problem exhibits a large and increasing rate of failed remote steals. Another relevant point, observable in Table 5.7.2, is the increasing number of total nodes that are processed until the solving process is finished.

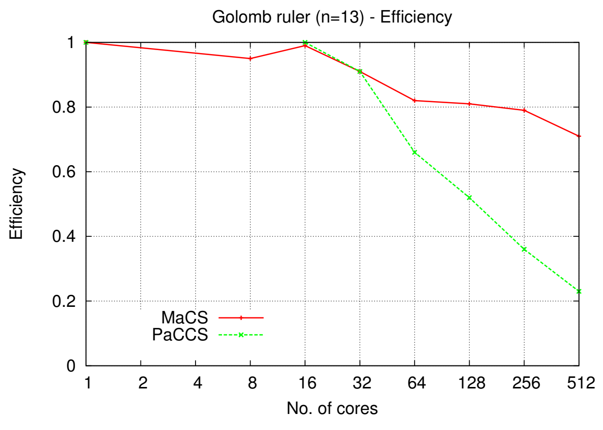
Lastly, Figure 5.7.7 presents the actual obtained parallel speedup(5.7.7a) and parallel efficiency(5.7.7b) for up to 512 cores, using MaCS and PaCCS. Contrarily to the previous problem (N-Queens) only the execution with the default settings of MaCS are plotted as they represent the best results.

The obtained results are encouraging but still leave room for improvement: MaCS sees speedups up to 512 cores although with moderate efficiency (71% at 512 cores). When related with the previous information on performance and load balancing, it becomes clear that the moderate speedups and parallel efficiency are due to the increasing number of nodes that must be processed. Although MaCS is close to ideal in terms of node throughput and low overhead, the increasing number of nodes does not allow similar values in terms of speedups and efficiency.

In summary, for both MaCS and PaCCS and when dealing with such optimisation problem, the most hindering factor is the growth on the number of nodes to process. As such, it has a less deterministic execution than a satisfaction problem since the number of processed stores depends on how fast the optimal solution is found and how fast this optimal solution is received and used by all workers. As the number of cores increases, the number of processed nodes most often increases as well, increasing the size of the problem when compared with the sequential execution.



(a) Parallel Speedup



(b) Parallel efficiency

Figure 5.7.7: Scalability - Golomb Ruler (n=13)

Langford's Problem

Langford's Problem is another CSP that we used for our experimental evaluation. The instance chosen was the $L(2, 15)$ where $k = 2$ and $n = 15$.

As before, we started by analysing MaCS' default execution in terms of overhead, in order to assess its behaviour for the problem at hand.

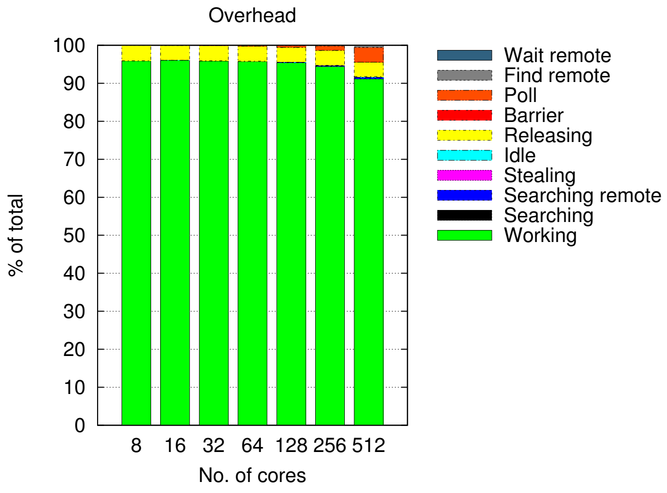


Figure 5.7.8: Working Time and Overhead - Langford's Problem (L(2,15).

From Figure 5.7.8 we can see that most of the time is spent on the *Working* state. The other notorious state is, similarly to the other CSP (N-Queens), *Releasing i.e.,* the time spent releasing work. This action is redundant in case there is a low requirement on work stealing. As the number of cores grows, we can see that *Poll* increases its influence, given the increasing number of cores and remote steals to handle.

The observed overhead is reflected on the performance as depicted by Figure 5.7.9. The performance of MaCS is more distant and diverging from the ideal case as we grow the number of cores yet remains very competitive with other systems, which normally only do multi-threaded parallelism or MPI, not both.

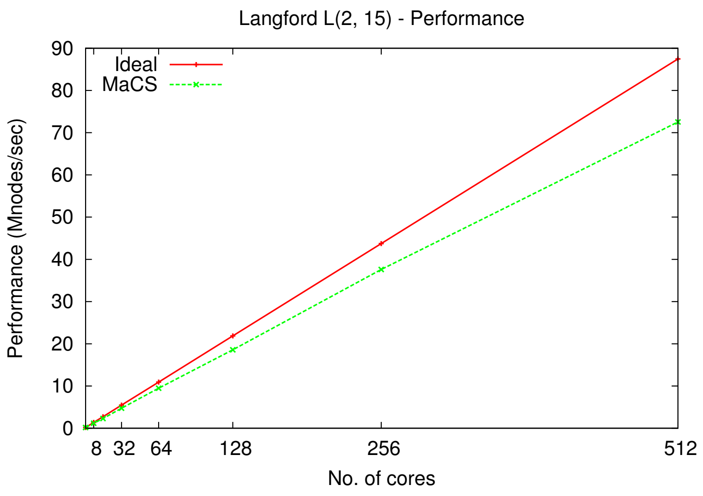


Figure 5.7.9: Performance (Million of nodes per second) - Langford's Problem (L(2,15).

This problem has many similarities with our first problem (N-Queens). The solving process also includes a high rate of processed nodes, as shown in Figure 5.7.9 and a similar distribution of the time spent on the three steps of solving: 53% on propagation, 8% on splitting and 39% on restoring. Again, a large portion of time is spent on retrieving nodes from the pool, which explains the consistently high overhead incurred by redundant release operations.

In terms of load balancing, the obtained information on work stealing allows us to observe further similarities between this problem and the N-Queens problem. Table 5.7.3 presents the obtained information.

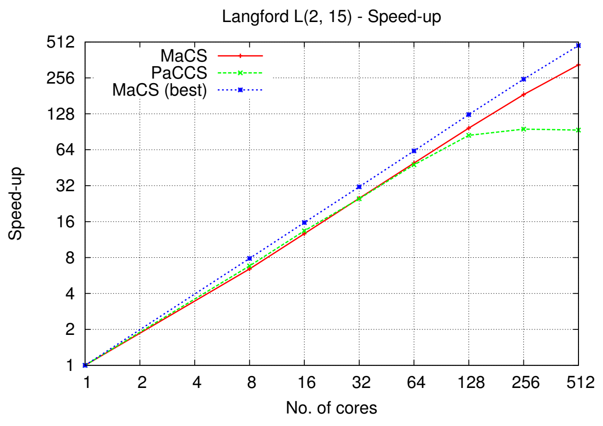
The imbalance is rather low given the low number of steals (local and remote) when compared to the total number of nodes processed. From Table 5.7.3 we can also observe a high number of failed steals at large scale, resulting in useless cycle consumption and increased overhead. The table also presents the total number of nodes processed, which for this problem remain constant in the case of MaCS.

Finally, we depict the parallel speedup (Figure 5.7.10a) and parallel efficiency (Figure 5.7.10b) of MaCS and compare them to those of PaCCS.

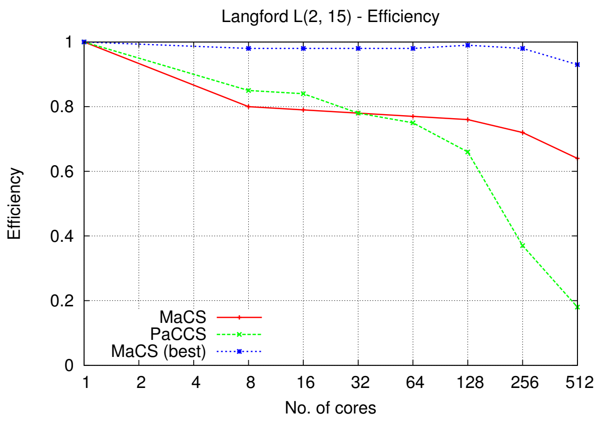
From Figure 5.7.10 we can observe that this problem also presents some challenges in terms of scalability, as both MaCS and PaCCS demonstrate moderate parallel speedup

Cores	Total Nodes	Local Steals				Remote Steals			
		Total	per core	Failed	Rate	Total	per core	Failed	Rate
8	844084349	784	98.04	4	0.51%	61	7.71	4	6.15%
16	844084349	3839	239.96	39	1.01%	416	26.00	43	9.37%
32	844084349	18154	567.31	200	1.09%	2163	67.61	183	7.80%
64	844084349	56386	881.03	709	1.24%	7445	116.34	770	9.37%
128	844084349	184782	1443.61	2403	1.28%	26094	203.86	2357	8.28%
256	844084349	462595	1807.01	6628	1.41%	73149	285.74	7243	9.01%
512	844084349	1141629	2229.75	18008	1.55%	194806	380.48	21450	9.92%

Table 5.7.3: Work Stealing Information - Langford's Problem (L(2,15).



(a) Parallel Speedup



(b) Parallel efficiency

Figure 5.7.10: Scalability - Langford's Problem L(2,15)

and efficiency. Already with 8 cores, the parallel efficiency of MaCS drops considerably and further down to 64% with 512 cores. PaCCS behaves slightly worse, mainly due to the increment on the number of nodes explored [109] from which MaCS does not suffer.

The default behaviour of MaCS is not optimal. The main cause is again the overhead caused by redundant release of work, causing a constant overhead that can be removed, as in the case of the N-Queens problems. Figure 5.7.10 also shows the best possible behaviour of MaCS when we optimise the release interval. In this case, almost linear speedups up to 512 cores are possible with 93% of parallel efficiency at the maximum number of cores.

Quadratic Assignment Problem (QAP)

The last problem evaluated was the QAP. The results were obtained with the *esc16e* instance.

The QAP exhibits similar results (Figure 5.7.11) as the other optimisation problem (Golomb Ruler) in terms of overhead and how workers spend their time. The overhead remains low for all states of execution, nevertheless with enlarging polling overhead as we increase the number of used cores and consequently the number of remote operations.

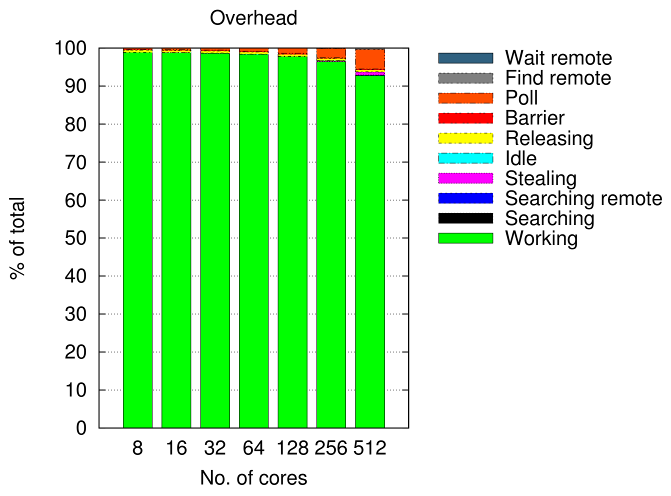


Figure 5.7.11: Working Time and Overhead - QAP (esc16e).

Figure 5.7.12 depicts the obtained performance with MaCS when compared with the ideal case. The results are near optimal, with scaling performance up to 512 cores.

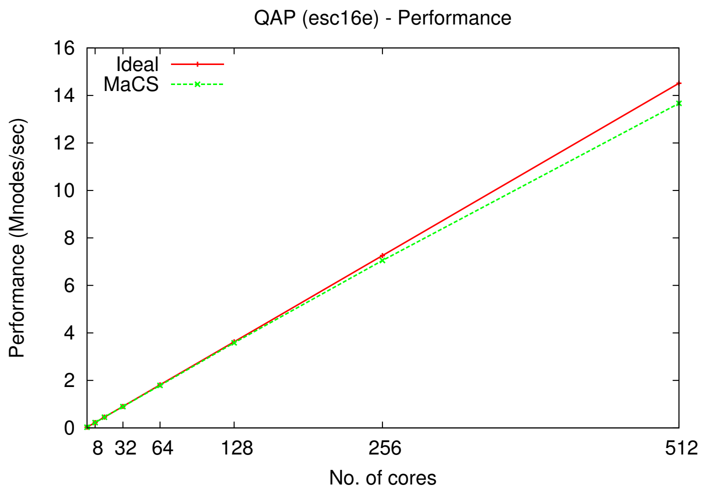


Figure 5.7.12: Performance (Million of nodes per second) - QAP (esc16e).

This problem can be further paired with the other optimisation problem in what respects how the constraint solving processed is divided: most of the time (80%) is spent on propagation whereas splitting and restoring consume 5% and 15%, respectively.

Table 5.7.4 presents the information about work stealing. It shows that the total number of steals (local and remote) increases as the number of cores increases. But interestingly,

Cores	Total Nodes	Local Steals				Remote Steals			
		Total	per core	Failed	Rate	Total	per core	Failed	Rate
8	328312656	1026	128.29	6	0.58%	54	6.75	0	0.00%
16	327522857	5499	343.71	24	0.43%	396	24.75	0	0.00%
32	327263896	19492	609.12	95	0.49%	1764	55.14	0	0.00%
64	327927026	78562	1227.54	403	0.51%	7325	114.45	13	0.18%
128	330423697	296747	2318.34	1612	0.54%	31601	246.88	52	0.16%
256	332779179	558626	2182.13	3007	0.54%	63200	246.88	266	0.42%
512	330921152	1203946	2351.46	4467	0.37%	136694	266.98	982	0.71%

Table 5.7.4: Work Stealing Information - QAP (esc16e).

the rate at which the number of steals increases is not constant. Up to 128 cores, the number increases fairly constantly at a factor of 4. If we observe the number of steals per core, we can observe an increase factor of two, roughly coincident with the increase of the number of cores. However, at a larger scale, the number of steals increases more slowly (around of factor of two) and when we observe the number of steals per core, we see this number staying more constant, with a slight decrease at 256 cores. This tendency is valid for both local and remote steals.

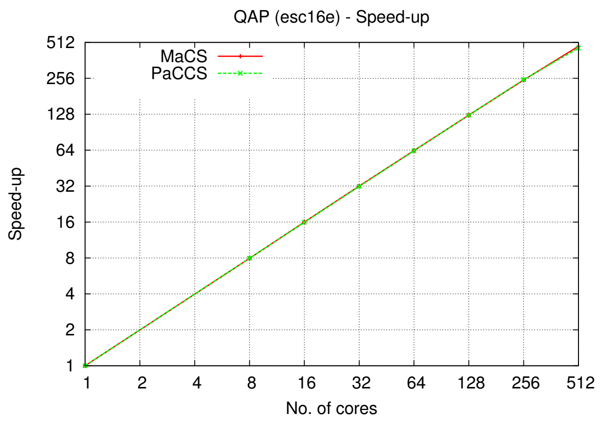
Another aspect relates to the number of failed steals: their number increases as the number of cores used is increased but remain very low - ideal up to 32 cores with zero failures - when compared with the total number of attempts to steal and when compared with the results obtained with the other problems.

The QAP is an optimisation problem and for the reasons mentioned before, the number of nodes (stores) processed by each run is not always the same, usually increasing as we grow the number of cores. However, from Table 5.7.4 we can see that the growth is not substantial.

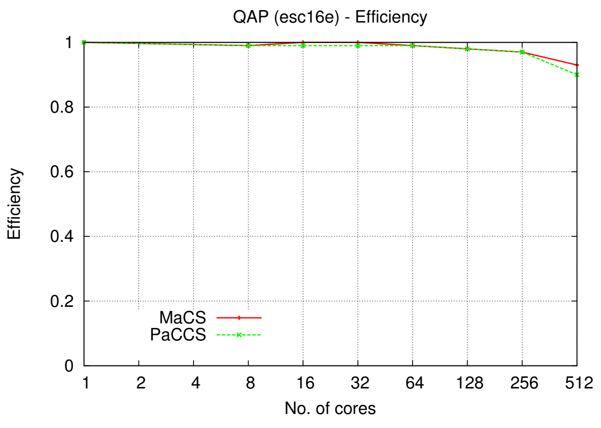
Figure 5.7.13 depicts the scalability obtained for the QAP problem for up to 512 cores. The obtained speedups are almost linear (Figure 5.7.13a) with a parallel efficiency above 90% (Figure 5.7.13b).

Both MaCS and PaCCS show good scalability up to 512 cores. With 512 cores, MaCS shows a slightly better efficiency (93%) than PaCCS (90%) but both results are very similar. Also in this problem, MaCS' default settings reveal to be enough and the best.

The QAP sees, on average, a less significant increase on the total number of nodes than the Golomb Ruler problem and therefore does not suffer as much in terms of scalability. Moreover, the work stealing mechanism is more efficient, with much less failures.



(a) Parallel Speedup



(b) Parallel efficiency

Figure 5.7.13: Scalability - QAP (esc16e).

5.8 Discussion

The performance evaluation from the previous sections showed that it is possible to obtain good scalability with MaCS on a large number of cores and with different kinds of problems.

The problems have similarities but also some differences which were made clear by the performance evaluation. The sizes of the different problems, in terms of the number of nodes processed, are of the same order of magnitude. The same can be said of the requirements in terms of load balancing where the ratio between the number of stolen stores and the total number of nodes processed is, on all problems, less than 0.5%. However, there are two important differences: the rate at which steals happen and their failure rate. These differences are related to the node throughput (performance) and the constraint solving process.

The evaluated CSPs, N-Queens and Langford's Problem, exhibit a high node throughput since processing a single store is a fast operation. As a consequence, the accesses to and maintenance of the work pool play an important role on scalability. On one hand the overhead incurred from *releasing* work in the pool is large, limiting better efficiency. On the other hand, more time is spent retrieving stores from the pool, including stealing operations which happen at a higher rate. Hence, both problems show a high rate of failed steals (local and remote).

The detailed evaluation of the different aspects allowed us to better understand the overall behaviour and, for the problems with scalability issues, namely both CSPs, it was possible to improve their scalability, reducing the overhead caused the *release* operations by simply increasing the interval at which they should happen.

In the case of the evaluated COPs, processing a single store is more costly and hence there is a lower node throughput. Both problems (QAP and Golomb Ruler) have low overhead but suffer from an increasing problem size as more cores are added, which affects their scalability. On the QAP, the growth on the number of nodes processed is not substantial and thus the parallel efficiency stays high (above 90% up to 512 cores). Moreover, the QAP is the problem with the lowest rate of stealing failures (local and remote). On the other hand, the Golomb Ruler problem, although having a linear speedup in terms of node throughput, sees a large growth in the total number of stores processed and this growth is reflected on the scalability.

Finally, it is constructive to refer to the capability and performance of MaCS in terms of sequential execution. In our performance evaluation, we compared the scalability of MaCS with that of PaCCS. PaCCS has showed comparable performance with Gecode and since MaCS is a fork from PaCCS, sharing or re-implementing most of the implementation specially in terms of constraint propagation, similar results were obtained. In other words, the sequential execution of MaCS is comparable with that of PaCCS.

Chapter 6

Conclusion

In this first part of this thesis we set out to investigate a declarative programming approach, Constraint Programming, using GPI as our underlying system and programming model. We focused on complete constraint solving and aimed at developing a system that could extract its potential for parallelism, making use of GPI and its features.

Search was identified as the component with the largest potential for parallel execution and thus, in order to maximise this potential we attacked the problem from a more general perspective. Focusing on parallel tree search, we implemented the mechanisms required to deal with the most important challenge in such problem type namely, dynamic load balancing. We implemented a new version of the UTS benchmark to take advantage of GPI. The obtained performance and scalability revealed encouraging up to 3072 cores, outperforming the MPI-based version by a factor of 2.5.

With the successful implementation of UTS, the comprehension of required mechanisms and of the common arising problems, we developed a system for large-scale constraint solving. We forked PaCCS, a recent and scalable constraint solver and used it as our base point to experiment and further improve. MaCS, the Massively parallel Constraint Solver, was the result of that development.

The main goal of MaCS was to take advantage of GPI to implement a parallel constraint solver which would perform well in large scale parallel systems, validating some ideas of PaCCS and introducing different and new options.

MaCS uses the compact and self-contained representation of a store similar to PaCCS. A store becomes an independent unit of work, suitable to relocation and to be handled by all participating workers. This representation allowed us to directly adapt the developed work with UTS to implement MaCS and hence, benefit from GPI and the work stealing strategy to deal with load balancing.

In the architecture of MaCS only one type of worker exists, responsible for all actions related to constraint solving and the parallel execution. Although the amount of work for each worker increases, it is possible to maintain the overhead low and achieve good scalability.

The approach taken and based on the observation that, from the parallelisation point of view, parallel constraint solving could benefit from a general approach proved fruitful. The experimental evaluation of MaCS showed a scalable parallel constraint solver on different problems with different characteristics, similarly to the previous work with UTS. Moreover, the detailed evaluation of the different components and from different perspectives revealed and identified important aspects such as sources of overhead which, together with MaCS parameters, helped achieving high parallel efficiency and speedups up to 512 cores. Although our objective was to go beyond this number as in the case of the UTS benchmark, the high contention for the utilisation of larger systems available did not permit us to accomplish it.

Our aim of experimenting with large parallel systems was advantageous since as the number of processing units grows, the behaviour of programs is different. It becomes harder to exploit parallelism, requiring different and efficient, even sometimes counter-intuitive, approaches. Unfortunately, it was not yet possible to have further access to a larger system - such as the one used for the experimental evaluation with the UTS benchmark - and continue the research in this direction.

The other important goal of the first part of this thesis was to study GPI and its suitability to such problems and their parallel execution. In both cases, the UTS benchmark and MaCS, the implementation with GPI showed encouraging performance and scalability results. Not only was it possible to achieve high parallel efficiency but also it allowed to improve on previous implementations.

One aspect, as important as the obtained results, is the fact that this work allows us to further understand the requirements of such computations. We argue it is possible to extend GPI with more functionality that matches even further these requirements while keeping an asynchronous and one-sided semantics. More concretely, the active polling strategy as a way to enforce synchronisation is efficient but not optimal. Although it constitutes a form of weak and asynchronous synchronisation, it still holds a source of parallel overhead that could be avoided. Instead of having to poll to perform an associated task, a thread could perform it only when it is really required to. In the case of work stealing, the task requiring polling is the remote steal. One possibility would be extending GPI with Active Messages [133]. Active Messages execute a handler upon message arrival and are therefore convenient for remote (atomic) operations such as the remote steal operation of MaCS.

We can therefore conclude that with such engagement, featuring a recent technology such as GPI and a declarative model based on constraints, it possible to obtain good scalability at large scale. Likewise, the claim that declarative approaches allow using parallel systems with increased productivity holds. With an approach such as Constraint Programming, the user can concentrate on the problem and less on the low-level details of parallel programming. As a byproduct, it encourages the user to tackle larger and harder instances of problems.

The encouraging results obtained with MaCS sustain the possibility of further extending this work in many ways. One is to extend the study of MaCS' behaviour on more problems, with different characteristics and experiment at larger scale in terms of the number of processors (cores) used and type of hardware (Intel Xeon Phi, GPUs).

The implementation of MaCS can also benefit from different optimisations and different strategies to overcome some problems. One example is the design of a new mechanism to avoid redundant polling, specially in problems with a low requirement in terms of work stealing. Here, a possible option would be the use of GPI passive communication. Closely related would be the addition of an automatic mechanism to detect the best work pool management policy, for example, in what respects the interval of release operations.

Another further improvement would be the inclusion of different strategies to find a victim in work stealing operations. This improvement would aim at reducing the observed number of failed steals (local and remote).

As observed in the performance evaluation, Constraint Optimisation Problems suffer from a growth of the problem size as more cores are added to the solving process. Here, a more efficient dissemination of the bound value could potentially mitigate that growth and thus, raise the parallel efficiency at large scale on such problems.

More generally, it would be interesting to answer the often considered question of whether re-computation, copying or even a hybrid approach of both can be better suited to exploit parallelism of large scale.

With the continued dissemination of parallel systems, Constraint Programming has the possibility to be more widely used. The work in this thesis and further research could allow the development of a general framework usable by different constraint solvers to take advantage of such parallel systems.

Part II

Parallel Constraint-Based Local Search

In this second part of the thesis we study an alternative approach to solving problems modelled on constraints: Constraint-based Local Search. Local Search is a very effective paradigm shown to be very efficient with a multitude of problem types. Constraint-based Local Search combines the declarative power of Constraint Programming with the effectiveness of Local Search.

There exist different Local Search methods, a well researched topic due to its usefulness and simplicity. We focus on the Adaptive Search algorithm as a representative of such algorithms. We provide the required to context to understand Local Search and its parallelisation. We then change our focus to the Adaptive Search method, where we present and evaluate different experiments on its parallel execution with new parallel versions based on GPI.

Chapter 7

Local Search

Problems modelled with constraints involve methods that find solutions in a large search space. This chapter presents Local Search as one of such methods and discusses the aspects and approaches to their parallel execution. This chapter also introduces Constraint-based Local Search, an approach that uses local search and constraints to solve hard combinatorial problems.

7.1 Introduction

In a declarative model based on constraints, a problem is presented as a constraint satisfaction problem (CSP) or a constraint optimisation problem (COP). Solving satisfaction or optimisation problems involves methods that require finding solutions in a large search space (*search*) together with techniques to prune parts of that search space that can remain unexplored due to their known lack of (optimal) solutions (*inference*).

Previously, we made the distinction between *complete* and *incomplete* methods when solving a CSP (or a COP). Complete methods have the property of *completeness i.e.,* they can prove optimality of a solution or prove its nonexistence whereas incomplete methods cannot. Completeness often requires the use of search, to look for alternatives when reasoning (inference) is exhausted.

Search is a crucial component of approaches that try to cope with NP-hard combinatorial optimisation and decision problems. If search is performed in a more systematic form as in backtracking-based methods, providing the guarantee to find the optimal solution or prove the feasibility of the problem, we refer to such search methods as *complete*. However, complete search algorithms impose a limitation on the problem size they are able to solve due to the exponential increase in processing time and memory requirements. An alternative are *incomplete* search algorithms which do not guarantee to find optimal solutions in finite time or prove that no solution exists. However, such methods are often necessary for large problem sizes. The solutions found are usually of high quality and thus, these algorithms are very effective.

Local Search algorithms are incomplete and based on the simple idea of “searching” by iteratively moving from one candidate solution to one of its neighbouring solutions. A *move* - a local change - is applied based on local knowledge only (hence the name Local Search). This process is iterated until a stop criterion is met, returning the current solution. Such algorithms are very effective as they converge rapidly to good quality “zones” of the search space. However, it is possible – and such is their main disadvantage – to get trapped in poor quality regions and return a non-optimal solution, i.e. a local minimum.

Notwithstanding the effectiveness of local search methods, for a considerable class of problem instances, the running time required might still be too substantial. One way to cope with this problem is by introducing parallelism. There exist different strategies to exploit parallelism in Local Search algorithms. Besides better performance, the parallel execution can make algorithms more robust, increasing the solution quality over a higher number of problem instances and characteristics.

In the first part of this thesis, we focused on complete search. In this second part, we concentrate on incomplete methods, namely Local Search. This first chapter introduces Local Search, some known algorithms and then focuses on the details about their parallelisation.

7.2 Local Search

Many problems require looking through a large search space to find solutions. While for some problems or instances thereof, it is possible to find solutions through an exhaustive exploration of the search space, many others require exponential run-time. In this case, completeness has be to sacrificed in order to obtain some solutions in polynomial time.

Local search algorithms are among the most effective to tackle hard problems with exponential run-time. Although these algorithms are not complete, they often find high quality solutions very efficiently. These algorithms have been applied to a large number of problem types [68, 54] such as circuit design, scheduling or planning problems.

Local search algorithms are a class of algorithms that move along the search space by replacing a candidate solution by a neighbouring one (Figure 7.2.1). Such algorithms start with an initial candidate solution, identify possible moves in the neighbourhood and select the best one based on some criteria. A move to the best neighbour is performed by applying some local change to the current configuration[1], picking the best candidate to replace the current one.

Local Search algorithms make use of heuristics to be able to identify a ’good’ solution and select neighbours of a configuration. A heuristic defines what constitutes a neighbourhood

[1]We use the term configuration to refer to a candidate solution. We distinguish a candidate solution from a solution in that a solution is the final state returned by the algorithm.

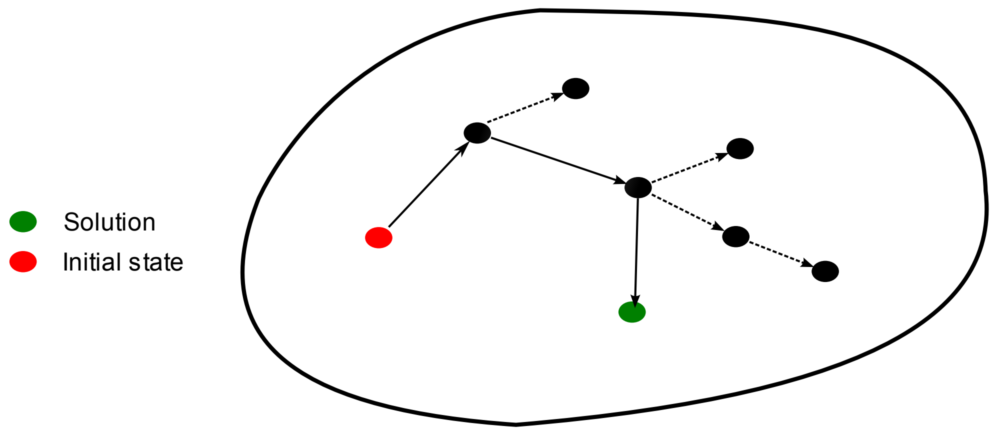


Figure 7.2.1: Local Search - moves along a search space

and is one of the main aspects that differentiates local search methods. In general, it is problem dependent and it is related to the definition of the problem's *objective function*.

There are important entities that need to be defined when discussing local search methods: the neighbourhood structure, the local minimum and the cost function. (Here we adopt the definitions from [127]).

Definition 4. Let S be the search space given by the set of all candidate solutions. A neighbourhood structure is a function $\mathcal{N}(s) : S \mapsto 2^S$ that assigns to every $s \in S$ a set of neighbours $\mathcal{N}(s) \subseteq S$. $\mathcal{N}(s)$ is also called the neighbourhood of s.

The neighbourhood structure can also be seen as a graph, denominated the *neighbourhood graph*. The nodes of the graph are configurations which are connected by an edge in case they are neighbours. A local search algorithm progresses by performing a *walk* on this graph, (usually) moving to the best possible neighbour. This walk is performed by a series of *moves*, transitions between neighbouring configurations. Such transitions are local modifications of some part of a candidate solution s which implicitly move a configuration to its chosen neighbour.

The move to be performed is selected based on the **cost function**. A cost function F associates to each solution $s \in S$ a value $F(s)$ which estimates the quality the solution. It is used to guide the search towards better solutions. In the case of a satisfaction problem, where one is only searching for solutions, the cost function commonly is based on the number of violated constraints, providing a metric on how far the current configuration is from a final solution (where no constraints are violated). In the case of an optimisation problem, the cost function also takes into account the objective function of the problem.

The major drawback of local search is that it relies on the cost function to estimate the quality of a solution as well as the possible local changes. As such, it can only guide the search towards locally good solutions.

Definition 5. A local minimum is a solution s such that $\forall s' \in \mathcal{N}(s) : F(s) \leq F(s')$. We call a local minimum a strict local minimum if we have $\forall s' \in \mathcal{N}(s) : F(s) < F(s')$.

While performing a walk, it is possible that a local search algorithm cannot improve the value of the cost function F or in other words, none of its neighbours represents a move to a solution with a lower value of F. Such a solution is a **local minimum**.

A very informal graphical representation of the progress of a local search algorithm can be depicted as in Figure 7.2.2. The walk is performed within the solution space while decreasing the value of the cost function. In some points, a local minimum is encountered. Assuming that the algorithm maintains mechanisms that allow it to avoid being trapped in local minima, the cost function may increase in some parts before it finds the global minimum.

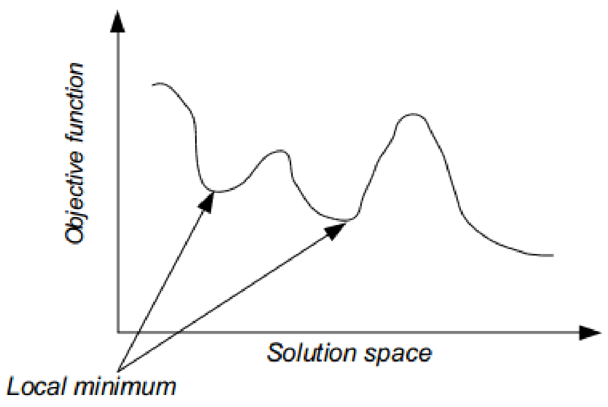


Figure 7.2.2: Representation of local minima

The existence of local minima is very common and different problems can have a large or low number of local minima. Their presence plays an important role and therefore there is substantial work and different techniques that aim at avoiding or escaping local minima. Some of the most simple techniques are restarts and randomisation. When a local minimum is encountered one simple approach is to simply restart the search procedure. Another option is to choose randomly one of the neighbours even though it does not improve on the cost function.

The general ideal of Local Search is simple and easy to understand, a reason which justifies its broad and frequent use. It turns out easy to implement by non-experts. A template of a Local Search algorithm is shown in Algorithm 8.

Algorithm 8 Simple Local Search template

```
1: $s_0 \leftarrow InitialSolution(S)$
2: $i \leftarrow 0$
3: **repeat**
4:     $m \leftarrow SelectMove(s_i, F, N)$;
5:     **if** $AcceptableMove(m, s_i, F)$ **then**
6:         $s_{i+1} \leftarrow s_i \circ m$
7:     **else**
8:         $s_{i+1} \leftarrow s_i$
9:     **end if**
10:    $i \leftarrow i + 1$
11: **until** $StopCriterion(s_i, i)$
```

The initial solution (*InitialSolution*) can be generated differently, according to the specific algorithm. Some algorithms require the construction of the initial solution to be performed in a specific way while others do not. In the latter case, other algorithms (e.g. constructive) can be used to obtain the initial solution or else it can be generated randomly.

The algorithm iteratively proceeds by applying *moves* given an acceptance criterion. If a move is accepted, then it is performed and the search continues from the new configuration (s_{i+1}).

The main loop stops when some termination criterion (*StopCriterion*) is met, specific to an algorithm and related to the quality of the solution and/or a maximum number of iterations.

To make the picture clearer, consider the example of solving a CSP with a local search algorithm (applying the template from Algorithm 8): the algorithm is started with a random initial solution *i.e.,* a random instantiation for each variable. A move is selected by considering the neighbourhood of the solution. The neighbourhood can be defined as the set of solutions that differ in a single variable instantiation from the current solution. From the whole set of neighbours, the one which reduces the number of violated constraints is accepted and a move towards that configuration is performed. This process continues until no more constraints are violated or the maximum number of allowed iterations is reached (the stop criterion), returning the last configuration as the final solution of the algorithm.

7.2.1 Meta-Heuristics

One disadvantage of simple Local Search algorithms is the possibility of getting trapped in local minima or poor quality parts of the search space. Local Search is based on heuristics and as such, it is subject to fail. Heuristics have this inherent limitation because they are just an informed guess, based on limited information. Therefore, local search algorithms may fail to find a solution or result in a sub-optimal one. It is possible to use strategies such as randomisation or restarting but these are only simple techniques which only alleviate the problem.

Trying to overcome this issue has lead to the development of so called *Meta-heuristics*. These are more general and complex schemes to solve large and complex problems. Meta-heuristics can be defined "*as master strategies (heuristics) that guide and modify other heuristics to produce solutions beyond those normally identified by heuristics such as local search*" [49].

Meta-heuristics, similarly to local search, also move between solutions but do so in a more intelligent manner, allowing for example a move to a "non-improving" solution if doing so it can avoid a premature stop of the procedure. Likewise, mechanisms to avoid *cycling* (re-visiting a sequence of solutions), learning and memorising previous solutions and parts of the search space are often used and beneficial.

Sometimes both terms, Local Search and Meta-heuristics, are used to refer to the same approach. Both refer to the use of heuristics to guide the search for a solution.

7.2.2 Intensification and Diversification

Two important concepts when considering Local Search methods are **Intensification** and **Diversification**. The concept of search *intensification* means that a more thorough exploration of the certain areas of the search space is performed, areas which seem heuristically more "'promising". With intensification one can try to ensure that the best solutions in such areas are found.

A general problem of Local Search methods is that these tend to remain very local and restrict their exploration to certain portions of the search space, failing to explore other, more interesting, areas. To avoid this problem, some *diversification* mechanism must by used to 'force' the exploration of other parts of the search space which otherwise would remain unexplored. One already mentioned technique is, for example, restarting the search procedure from another, possibly randomised, point.

It is important to stress that a good trade-off between intensification and diversification is crucial to the good performance of a Local Search technique. Both interact in complex ways, and minor variations can make a considerable difference in the actual performance of the algorithm.

7.2.3 Algorithms

In this section we review some popular and influential algorithms: some are more basic local search approaches such as the Hill Climbing techniques, others rely on more complex mechanisms (Meta-Heuristics) such as memory, probabilities (*e.g.,* Tabu Search, Simulated Annealing) or some nature-inspired structure.

Hill Climbing

Hill Climbing techniques are among the simplest local search techniques. They are based on the simple idea of iterative improvement: at each step, take a move that improves on the current solution where *improving* means to do so for the cost function or leave its value unchanged.

The family of Hill Climbing algorithms includes several well-know techniques:

SHC The *Steepest Hill Climbing* selects, at each iteration, the neighbour with the minimum value of the cost function. The move is accepted if it is an improving one and thus it stops as soon as it reaches a local minimum.

RHC With the *Random Hill Climbing* technique, the neighbour is chosen randomly from the neighbourhood of the current solution. The move is accepted if the cost function improves or remains equal. If the cost degrades, the current solution stays the same for the next iteration.

MCHC The widely used *Min-Conflict Hill Climbing* [122] technique is divided into two steps: first, it chooses a random variable v of the current solution that is involved in at least one constraint violation. Secondly, it selects for variable v, the moves that change the value of v and minimises the number of constraint violations.

Simulated Annealing

Simulated Annealing [78, 15] is a popular technique that received its name by analogy with the physical annealing of solids, a process of cooling a substance to grow a perfect solid, in a controlled manner.

Simulated Annealing starts by creating a random initial solution. At each iteration, a neighbour of the current solution is generated at random. If the new solution is an improvement then it is accepted; otherwise, the new solution is accepted with a probability $e^{-\Delta/T}$ where Δ is the difference of costs between the new and the current solutions and T is the temperature parameter. The temperature parameter T is updated at a defined *cooling rate*, starting from an initial high temperature T_0.

There are many variants of the basic Simulated Annealing, focusing on the different parameters such as the temperature and its decrease or the probabilistic acceptance rule.

Tabu Search

Tabu Search was proposed and introduced in [53, 51, 52] and was used extensively in several problem domains [28, 46, 63, 125]. It is a memory-based strategy as it keeps features of solutions and the search process across iterations. Its basic mechanism (more complex ones are possible) proceeds as the following: at each iteration a subset of the neighbourhood of the current configuration is explored. The element of this subset of the neighbourhood that gives a minimum value for the cost function is taken even if its value is worse than the current solution. The method includes a *tabu list* which keeps track of the last m moves. If a move is in the tabu list, it can not be executed. This prevents *cycling i.e.,* visiting previously visited solutions and stagnation at local minima. One drawback with this mechanism is that it overlooks good solutions. To overcome this, Tabu Search has another mechanism called *aspiration criterion* that ignores the tabu status of a move if the improvement it gives is *large*.

There are several improvements to Tabu Search related to its basic parameters (tabu list, aspiration function, cardinality of the neighbourhood considered) that can be found in the literature [47].

GRASP

GRASP [39] is a multi-start procedure. A multi-start procedure relies on the fact that by using different starting solutions, local search will eventually lead to the global optimum. However, instead of using randomly generated solutions, GRASP uses a greedy algorithm to construct a solution, one element at a time and hence produce solutions of better quality. The general scheme works as follows: on each restart, a randomised greedy construction heuristic is used to generate an initial solution, which is then improved by local search. A defined number of restarts is performed and the best overall solution is returned as the final solution.

One often pointed drawback of GRASP is the total independence of restarts, preventing the method to exploit knowledge obtained in previous iterations to guide the search. To that end, different strategies have been developed to make use of information accumulated during iterations. Reactive GRASP [112] is one example. In [42] a pool of elite solutions is used to implement *path relinking* [50], an approach to integrate diversification and intensification in the search.

Nature-inspired Algorithms

There is a substantial amount of work dedicated to finding optimisation algorithms which go beyond those already described. Some of which are fairly recent developments, and we only discuss a few.

One curious common point of such algorithms is their inspiration on natural and biological processes. The oldest and probably most successful of these nature-inspired algorithms are Genetic Algorithms (GAs) [64]. Essentially, GAs are based on Darwin's ideas of evolution and natural selection using crossover, mutation, fitness and selection of the fittest as operators. Ant Colony Optimisation (ACO) [38] draws its inspiration on ants and their use of pheromones as a chemical messenger. Particle Swarm Optimisation (PSO) [77] is based on swarm behaviour such as that of fishes and birds, a phenomenon usually named *swarm intelligence*. PSO is similar to GAs but simpler, not making use of mutation or crossover but rather real-number randomness and global communication among the swarming particles.

More recent developments include further nature-inspired algorithms. The Firefly Algorithm [137] is inspired on the flashlight characteristics of fireflies. Cuckoo Search [139] is based on the obligate brood parasitic behaviour of some cuckoo species in combination with Lévy flight behaviour of some birds and fruit flies whereas bat-inspired algorithms [138] are based on echolocation behaviour of bats.

These recent algorithms have proved efficiency in at least one problem type, emphasising a trend towards finding the best algorithm for a specific class of problems and less so for general-purpose algorithms.

7.3 Parallel Local Search

The parallelisation of Local Search algorithms or, for that matter, of Meta-heuristics requires, from a programming point of view, the same general approach: one needs to identify the points amenable to parallelisation, thereby extracting data and/or task parallelism. If, on one hand, such algorithms have characteristics that make their parallelisation hard, on the other they present new ways of exploiting parallelism. Instead of the more traditional view where parallelisation is a mean of reducing the execution time, with local search algorithms, and given their stochastic nature, different goals are possible:

Speedup one aims at reducing the execution time, speeding up the search.

Problem size adding more computing power allows one to tackle larger problem instances, out of reasonable reach for a sequential execution.

Solution quality increase the solution quality by exploring different regions of the search space, possibly ones not explored with a sequential execution.

Robustness increase the ability to deal with different problems and/or different instances of a given problem while making the algorithm less sensitive to parameters and problem-related tweaking. This also includes reducing the dispersion of execution times required to find a solution for a problem instance.

Moreover, all the different goals can be combined resulting in for example, a better solution in terms of quality in a reduced processing time.

An algorithm started from different initial solutions will almost certainly explore different regions of the search space and return different solutions. The different regions of the search space explored can then become a source of parallelism for meta-heuristic methods. However, the analysis of parallel implementation of meta-heuristic methods becomes more complex because often the parallel implementation does not return the same solution as the sequential implementation. Evaluation criteria based on the notion of solution quality (i.e., does the method find a better solution?) have then to be used to qualify the more classical performance metrics, such as speedup.

In [132], the authors observed that most algorithms are based on a few ideas only, abstract from implementation details. They distinguish between tailored and general approaches. Tailored approaches are applied to specific problems whereas general approaches can be applied to a wide variety of problems. Furthermore, they distinguish between single-walk and multiple-walk parallelism and between asynchronous and synchronous algorithms. In the class of single-walk parallelism, a distinction is made between multiple-step and single-step parallelism.

Tailored approaches depend on the problem they are applied to. In these approaches a specific part of the algorithm is parallelised such as the computation of the cost. This reduces their applicability but can be of advantage due their specificity (e.g. better performance).

General approaches are more appealing as they can be applied to different problems. They can be classified as single- or multiple-walk. In single-walk parallel local search a single walk is performed in parallel. This approach can be further characterised as single- or multiple-step algorithms. A multiple-walk algorithm performs several single walks *i.e.,* each processor performs a single walk. The walk each processor performs can be independent or an interacting walk. Finally, in single-walk and multiple-walk parallel local search a further distinction can be made, between *synchronous* and *asynchronous* algorithms. In *synchronous* algorithms, one or more steps are performed simultaneously by all processors whereas in *asynchronous* algorithms, the steps of the algorithms do not need to occur at the same time or with fixed time intervals.

In the **Single-Walk** approach, speedup is pursued by parallelising the evaluation of the cost function or on creating the neighbourhood. The walk is likely to go through the same steps, in the same order as the sequential algorithm. The performance advantage comes from a faster evaluation of the cost function or a better search procedure allowed by creating a larger neighbourhood.

Independent Multiple-Walks constitute the simplest approach to parallel local search. A walk is carried out by each processor without any communication between them. Processors (search threads) start at a solution and perform their own walk, intersecting or not, with walks from other processors. The same or different algorithms can be used to perform the walk, with the same or different parameters. Furthermore, each search thread can start from the same or from a different solution.

Due to its simplicity, this approach is often used. But more importantly, the yield results in terms of speedup and solution quality are also very appealing as stated in Proposition 1.

Proposition 1. Let $Q_p(t)$ be the probability of not having found a solution with a given quality in t time units with P independent walks. If $Q_1(t)$ = with $\lambda \in \mathbb{R}^+$ *i.e.,* Q_1 corresponds to an exponential distribution, then $Q_P(t) = e^{-pt/\lambda}$

Proposition 1 tells us that it is possible to achieve linear speedup with the independent multiple-walk approach if the probability to find a (sub)optimal solution within a given amount of time units is distributed exponentially.

A similar proposition can be made for the case of a two parameter (shifted) exponential distribution:

Proposition 2. Let $P_p(t)$ be the probability of not having found a solution with a given quality in t time units with p independent walks. If $P_1(t) = e^{-(t-\mu)\lambda}$ with $\lambda \in \mathbb{R}^+$ *i.e.,* Q_1 corresponds to a two parameter exponential distribution, then $Q_P(t) = e^{-p(t-\mu)\lambda}$

Independent Multiple-Walks are attractive since they are easy to implement, possibly leading to good speedups if problems satisfy Propositions 1 or 2. Since several search threads perform different walks, a good coverage of the search space can be made given a good search space partitioning that avoids intersecting walks and redundant work. Still, this possibility exists and if this redundancy exists, some of the potential of parallelism is being lost. Also, the information collected by the algorithm in its own walk could be used to guide the walks of other threads, improving the overall global search.

Interacting Multiple-Walks try to overcome some of the shortcomings of the Independent Multiple-Walks by introducing interaction (cooperation) between search threads. In these approaches, the different search threads communicate among them, sharing or exchanging information about their own walk, in order to improve their walk with the information from others. One expects that the information exchange would speedup the convergence to the best solution, but also help find better solutions within the same computation time bounds as the Independent Multiple-Walk.

An Interacting Multiple-Walk poses more difficulties in terms of implementation. Several questions have to be answered and a tighter coupling with the programming environment is usually required to be able to answer them, together with higher proficiency in parallel programming. These questions are mostly related to the communication and are key points. The general aim is that the communication, although an extra step, allows the algorithm to achieve better performance and this is only possible by communicating useful information at the right moment. Search threads become cooperation partners searching for a global solution, augmenting their local view.

Among the important questions, we distinguish, similarly to [129], the following ones:

- What information should be communicated?
- When should the communication take place?
- What is the communication structure?
- What to do with the communicated information?

The question of what information should be communicated is probably the most difficult. The information exchange should be helpful and meaningful so that the global search improves. One option is to communicate complete solution(s) such as the current best solution or at least "good" solutions, sometimes referred as *elite solutions*. Another option is to communicate context information which has been gathered by a search thread during its own walk as for example, exchanging statistical information or details particular to the workings of an algorithm (e.g. tabu list in Tabu Search).

The communication exchange constitutes an extra step, to add to the sequential execution of the algorithm. Thus, the question is when should this step be taken so that this "disturbance" is minimal and is of advantage to the procedure. Redundant or useless communication constitutes solely a source of overhead and is detrimental in terms of performance. Search threads can communicate directly with each other through the exchange of messages or indirectly through access to a global data structure (often used terms are *blackboard or elite pool*). The communication can happen at an agreed-upon point of the execution or each thread decides independently when the communication is done, according to its own execution.

One important point related to the question of when should communication take place, is how this communication should happen or what is the communication structure. In fact, this is a general concern when designing a parallel algorithm. It relates to the logical structure of search processes and how they are organised. Search threads can be organised in different structures and exchange information differently. It is possible that search threads communicate with all other threads by broadcasting information or to define topologies such as rings, toruses or other types of communication graphs. Also, considerations about the underlying hardware have to be made (*e.g.,* is the target a shared-memory system or a distributed shared-memory system?).

The last important question – what to do with the communicated information – defines how the procedure acts on the communicated information to its advantage. A search thread, upon receiving a solution, can 'blindly' accept it and use it to progress further. Or it may save it for use in a later stage of execution, or use it directly or even draw more complex information from it.

All of the previous questions are interrelated and answering one of them may influence the answer to another. Moreover, introducing cooperation ought to preserve the trade-off between intensification and diversification of the algorithm. While communication is intended to provide extra meaningful information, it may be biased to the intensification of the search since there may be a tendency to follow well-behaving threads. There might be the need to define extra mechanisms to allow the search to diversify in the presence of such implicit intensifying behaviour.

In [26] the authors adopt a classification, generalising that of [25], to describe different parallel strategies for meta-heuristics. In that classification, three dimensions are identified and indicate: i) how the global problem-solving process is controlled, ii) how the information is exchanged among processes and iii) the variety of methods involved in the search for solutions. Their classification is more comprehensive and fine-grained but proceeds on the same spirit.

Related work

The work on parallel local search and meta-heuristics is very broad, addressing particular algorithms only but also more general and comprehensive issues.

We already referred to [132, 26] as surveys and taxonomies to generalise and describe different strategies for parallel local search and meta-heuristics. More recently, [27] recalled the same taxonomy and discuss implementation issues particularly the influence of the characteristics of the target architecture, focusing on Variable Neighbourhood Search (VNS) [97] and Bee Colony Optimisation (BCO) [128].

It is noteworthy to refer that in [26], the authors provide an important survey with many references to the broad field of parallel meta-heuristics. That includes syntheses to particular methodologies as well as to syntheses that address more than one methodology. We refer to that work for further references on particular algorithms (simulated annealing, tabu search, genetic-based evolutionary methods, scatter search, ant-colony methods and Variable Neighbourhood Search) and more general surveys, to avoid redundant referencing.

Local search methods have also been used to develop SAT solvers and check the satisfiability of Boolean formulae. There exist different approaches for parallel SAT and several solvers were developed [105, 4, 60] with good results.

In spite of the broad and constant work on parallel local search and meta-heuristics, there isn't much work focused on large-scale parallelism that for instance, makes use of more than 32 cores. There is a certain trend that focuses on the use of recent hardware namely GPUs, to accelerate and investigate parallel local search (see for example [85, 131]), acknowledging that research in this direction is gaining momentum.

7.4 Constraint-based Local Search

As presented in previous chapters, constraint programming constitutes a useful methodology to solve a variety of combinatorial problems at a high level of abstraction manageable by domain experts whereas local search approaches to combinatorial optimisation are able to isolate optimal or near-optimal solutions with high performance and handle large problem sizes, but usually require a more involved and sensitive programming effort.

Constraint-based Local Search (CBLS [62]) is a method for solving combinatorial optimisation problems that combines constraint programming and local search, using constraints to describe and control local search. In a sense, this approach tries to get the best of both, with high level modelling and control at high performance and towards real world instances.

In CBLS, the set of constraints C of a problem is associated with a function $\mathcal{F}$ which serves as a violation metric and maps the variable assignments α to a single, non-negative value. Each constraint c can be given a weight w_c to aid at guiding the search. The problem is turned into a minimisation problem:

$$\begin{array}{ll} \underset{\alpha}{\text{minimise}} & \mathcal{F}_C(\alpha) \\ \text{where} & \mathcal{F}_C(\alpha) = \sum_{c \in \mathcal{C}} w_c \mathcal{F}_c \end{array}$$

The function $\mathcal{F}$ is the cost function to which the local search algorithm is applied. If the algorithm finds the global minimum *i.e.,* $\mathcal{F}_C = 0$ then the corresponding assignment satisfies the set of constraints C. if $\mathcal{F}_C > 0$ then C was not satisfied and the returned solution is the best found solution.

The reference work on CBLS is Comet [94, 95], as a successor of Localizer [93]. Comet is an object-oriented programming language specifically designed to support the implementation of local search algorithms and supporting an architecture around constraints. Galinier and Hao [45] present a general approach for solving constraint problems by local search. More recently, Kangaroo [102], a CBLS system, is introduced, reporting a smaller footprint and faster results when compared with Comet. It employs a lazy strategy for updating invariants as its most distinguishing feature.

7.5 Summary

In this chapter we presented Local Search, Meta-heuristics and referred to Constraint-Based Local Search as an approach to solve hard combinatorial problems that require searching through a large search space.

The idea of Local Search is simple, intuitive and easy to implement by domain experts. However, it is an incomplete method and it may get trapped in low quality regions of the search space and lead to local minima. A good trade-off between search intensification and diversification must be considered.

Local Search is based on heuristics and has a stochastic behaviour. Its characteristics pose a hard challenge to the parallel execution and on how to take advantage of parallelism. Elementary questions such as if communication is at all required or what to exactly communicate must be answered.

We have presented Parallel Local Search as a topic based on a few general ideas: whether search threads perform a single or multiple walks, starting from the same or different configurations or if they cooperate by communicating and acting on meaningful information. There is no definitive answer to the best approach. Different algorithms can be used to tackle problems with fundamentally different characteristics.

In the next chapter, we concentrate on one of such algorithms and explore its parallelisation using GPI.

Chapter 8

Adaptive Search

The Adaptive Search algorithm is a local search procedure which proved itself effective for several classes of problems. This chapter introduces Adaptive Search and its parallel adaptations. We present a new parallel version of Adaptive Search based on GPI and its different variants. Our experimental evaluation presents an in-depth analysis of its behaviour when executed on up to 512 cores.

8.1 Introduction

Adaptive search [20, 21] is a local search method for solving Constraint Satisfaction Problems (CSPs.) The key idea of the approach is to take into account the structure of the problem given by the CSP description, and to use in particular variable-based information to structure applicable meta-heuristics. Because of this, Adaptive Search may be characterised as a Constraint-Based Local Search (CBLS) method.

The input to this method is a problem in the CSP format. Again, this means a set of variables with their respective finite domain of possible values and a set of constraints over these variables.

The method is not limited to any specific type of constraint. However, it does require an explicit indication of how much a constraint is being violated, in the form of an *error function*. For example: an arithmetic constraint $X - Y < C$ will have as error function $max(0, |X - Y| - C)$.

The basic idea of this method can be described by 3 steps:

1. compute the error function for each constraint
2. combine, for each variable, the errors of all constraints in which it appears
3. the variable with the high largest error will be chosen and thus have its value modified. In this step it uses the well-known Min-Conflict [122] heuristics and select the value

in the variable domain that has the most tempting value, that is, the value for which the total error in the next configuration is minimal.

To avoid being trapped in local minima and loops the method uses a short-term adaptive memory similar to Tabu Search.

In this thesis, we concentrate on the Adaptive Search method as our Local Search method to investigate due to its efficiency and the existing work on parallel execution with MPI. This provides us with a basis for comparison in our own experimentation.

Another important point is the availability of models for different problems with different characteristics. The models are available with optimised parameters, which we use as a starting point.

8.2 The Algorithm

For instance, consider an n-ary constraint $c(X_1, \cdots X_n)$ and associated variable domains $D_1, \cdots D_n$. An error function f_c associated to the constraint c is a positive-valued function from $D_1 \times \cdots \times D_n$ such that $f_c(X_1, \cdots X_n)$ has value zero iff $c(X_1, \cdots X_n)$ is satisfied. The error function is used as a heuristic to represent the degree of satisfaction of a constraint and thus gives an indication on how much the constraint is violated. This is very similar to the notion of "penalty functions" used in continuous global optimisation. This error function can be seen as (an approximation of) the distance of the current configuration to the closest satisfiable region of the constraint domain. Since the error is only used to heuristically guide the search, we can use any approximation when the exact distance is difficult (or even impossible) to compute. Adaptive Search is a simple algorithm (see Algorithm 9) but it turns out to be quite efficient in practise [21]. Considering the complexity/efficiency ratio, this can be a very effective way to implement constraint solving techniques, especially for "anytime" algorithms where (approximate) solutions have to be computed within a bounded amount of time.

Algorithm 9 Adaptive Search Base Algorithm

Input: problem given in CSP format:
- set of variables X_i with their domains
- set of constraints C_j with error functions
- function to project constraint errors on vars
- (positive) cost function to minimise

some tuning parameters:
- TT: number of iterations a variable is frozen
- RL: number of frozen variables triggering a reset
- RP: percentage of variables to reset
- MI: maximal number of iterations before restart
- MR: maximal number of restarts

Output: a solution if the CSP is satisfied or a quasi-solution of minimal cost otherwise.

Algorithm:

```
1:  $Restart \leftarrow 0$
2:  repeat
3:      $Restart \leftarrow Restart + 1$
4:      $Iteration \leftarrow Tabu\_Nb \leftarrow 0$
5:      Compute a random assignment $A$ of variables in $V$
6:      $Opt\_Sol \leftarrow A$
7:      $Opt\_Cost \leftarrow cost(A)$
8:      repeat
9:          $Iteration \leftarrow Iteration + 1$
10:         Compute errors of all constraints in $C$ and combine errors on each variable
11:                      ▷ (by considering only the constraints in which a variable appears)
12:         Select the variable $X$ (not marked Tabu) with highest error
13:         Evaluate costs of possible moves from $X$
14:         if no improvement move exists then
15:             mark $X$ as Tabu until iteration number: $Iteration + TT$
16:             $Tabu\_Nb \leftarrow Tabu\_Nb + 1$
17:             if $Tabu\_Nb \geq RL$ then
18:                 randomly reset $RP$ % variables in $V$ (and unmark those Tabu)
19:             end if
20:         else
21:             Select the best move
22:             Update the value of $X$, yielding the next configuration $A'$
23:             if $cost(A') < Opt\_Cost$ then
24:                 $Opt\_Sol \leftarrow A \leftarrow A'$
25:                 $Opt\_Cost \leftarrow cost(A')$
26:             end if
27:         end if
28:     until a solution is found or $Iteration \geq MI$
29: until a solution is found or $Restart \geq MR$
30: $output(Opt\_Sol, Opt\_Cost)$
```

8.3 Parallel Adaptive Search

When parallelising an algorithm one aims at identifying hot-spots and sources of parallelism. In Adaptive Search these sources of parallelism are essentially the inner loop of the algorithm and the search space of the problem.

By inner loop we refer to the loop of the Algorithm 9 starting at line 8. Within this loop there are several possibilities to exploit parallelism:

i. On line 10, the computation of errors on each variable could be done in parallel by partitioning the current configuration in n different portions, exploiting data parallelism.

ii. On line 12, the algorithm selects the variable with the highest error. It is possible that more than one such variable exist *i.e.,* it may be the case that several variables have the same highest error. In this case, the sequential execution of Adaptive Search chooses a random variable from the set of variables with highest error. A parallel implementation could proceed with a set of n variables instead of just one to the next step and pick the best improving move from all the possible choices.

iii. On line 13, the same as in the previous point could be argued. The number of possible moves such that there is an improvement of cost is not necessarily one *i.e.,* several moves are possible that yield the same improvement of the cost and the sequential implementation chooses a random one in this case. A parallel version could proceed with n different but equally improving moves, concurrently.

Each of the possibilities can be exploited separately or together as a series of steps in a fork-join model.

The problem with exploiting the inner loop of the algorithm is its granularity: it is too fine-grained. The overhead associated with synchronisation and dispatching of tasks comes at too high a cost.

The other main source of parallelism is the search space (domain) of the problem itself. Theoretically, this domain could be decomposed in several disjoint partitions to be explored in parallel and without dependencies. However, in practise, several issues arise with this. Each partition is in general still too large for a sequential execution and more importantly, not the whole search space is equally valid and the exploration should avoid areas of it that lead to poor solutions. Moreover, it is hard and expensive to control and maintain the search conducted in the different partitions since a Local Search algorithm only has a local view of the search space. One example is the case of problems that have the best solutions clustered in a particular region of the search space. In this case, the algorithm should converge to that zone but in case of parallel execution avoid redundant work, as much as possible.

The Adaptive Search method has already been subject to some research on its parallel behaviour. Previous work on parallel implementations of the Adaptive Search algorithm have mostly focused on independent multiple-walks, requiring neither communication nor shared memory between processing units.

In [33], the authors present a parallel implementation of the Adaptive Search algorithm for the Cell/BE, a heterogeneous multicore architecture. The system includes 16 processors (the SPEs) where each one starts with a different random configuration. The PPE acts as the master processor, waiting for the message of a found solution. For such number of processing units, the results were very promising, achieving for some problems linear speedup.

Further work with Parallel Adaptive Search continued to follow the same approach with no communication between workers but, more interestingly, concentrating on cluster systems with a larger number of cores.

In [14], the authors experiment and investigate the performance of a multiple independent-walk on a system with up to 256 cores. The parallelisation was done with MPI and involves the introduction of a "communication step" which tests if termination was detected (*i.e.,* a solution was found) and terminates the execution properly, in case it was.

The presented performance results are relatively modest in terms of parallel efficiency and still far away for the ideal speedup. This contrasts with the results obtained at a smaller scale (*i.e.,* . up to 16 cores) in previous work with the Cell/BE. This points out the need for better alternative strategies in order to better exploit large-scale parallelism.

In [31, 32], the authors focus on a particular problem, the Costas Array Problem (CAP), and its parallel execution on a large number of cores. The CAP is a hard combinatorial problem with practical applications and the authors present its modelling and parameter tuning. Using an independent multiple-walk, they observed linear speedups on different systems and up to 4096 cores (IBM Blue Gene/P).

Since the independent multiple-walk approach still leaves space for improvement in terms of parallel efficiency and scalability for some problems, new ways to take full advantage of parallel systems must be found. This means that some form of communication or more generally said, cooperation, must be introduced.

In [13], the authors experiment with more complex strategies, where processes exchange messages resembling branch-and-bound methods where the bound is exchanged between all participants. In their work, two alternatives are attempted: i) exchanging the cost of the current solution of each search process and ii) the current cost plus the number of iterations needed to achieve that cost. Unfortunately, neither approach achieves better results than an independent multiple-walk.

8.4 AS/GPI

Previous work with parallel Adaptive Search provides some groundwork to build upon and has showed that some problems exhibit scalability issues when run on a large number of cores. Specifically, some problems were identified and some reasons were conjectured [13]:

- Restarting a search process seems to reinforce the intrinsic sequential duration in the Adaptive Search method, thus producing a smaller parallel speedup. As a consequence, the number of restarts should be minimised as much as possible;
- Taking the decision to restart every c iterations may not be the most effective one. Each process spends a different amount of time on each of the c iterations. It is thus very difficult to automatically tune this parameter, which clearly depends on the problem size (at least.) Furthermore, taking a decision here can only trigger more restarts than it should: if this can improve the search on some executions, it appears not to be the case on average;
- The choice of the metric, *i.e.,* the cost value, is not precise enough and may be misleading in some cases. Other information should also be shared.

GPI seems *à priori* and from a high-level point of view, to be an interesting match to the problem of parallelisation. Local search methods work with local information, trying to progress and converge to solutions in a global search space, requiring little global information. As demonstrated by previous work, some problems exhibit low parallel efficiency. Hence, communication and cooperation become a possible avenue to further improve the scalability. The communication with GPI is based on one-sided primitives that ought to benefit the local view on a global search space, as it allows threads to cooperate asynchronously. Moreover, communication is very efficient as GPI exploits the full performance of the interconnect with little or no CPU intervention. Hence, we aim at exploring ways to further improve the parallelisation of the Adaptive Search algorithm, exploiting the efficiency of GPI and its programming model, with the objective of getting further benefits. But more importantly, we aim to find mechanisms, concepts or limitations that are general.

This study can be of benefit not only for Adaptive Search but also to find indicators that might be helpful for local search in general or even other methods.

In general, we can define the following objectives:

- Further investigate and **understand the behaviour** of parallel Adaptive Search on different problems and at large-scale.
- Investigate the **possibilities offered by GPI** and devise more sophisticated mechanisms for the parallel execution of Adaptive Search, improving its performance.
- Identify the – possibly new – **problems** generated by the previous point.
- Identify the consequences of using a different programming model such as GPI on the development and parallel execution of a local search method.

- Investigate the suitability of GPI for these problem domains.

The new parallel version of Adaptive Search based on GPI (AS/GPI) includes two variants which we name TDO (Termination Detection Only) and PoC (Propagation of Configuration).

The TDO variant implements the simple independent multiple-walk, serves mostly as our basis for comparison with two major goals: first, with the existing MPI version, to make sure that the implementation is correct and the performance is as expected. Second, to allow us to measure the improvement (if any) obtained with the more complex PoC variant. The PoC variant introduces more communication and sharing between working threads but it is our expectation that the overhead will be offset by the performance gain.

The following sections present the two different variants in more detail.

8.4.1 Termination Detection Only

The variant with Termination Detection Only (TDO) is rather straightforward and implements the idea of an independent multiple-walk: all available cores execute the sequential version of the Adaptive Search algorithm.

We name this variant as Termination Detection Only because it amounts to a termination detection problem *i.e.,* detecting the termination of a distributed computation. Termination Detection is in itself a subject of much research and several algorithms have been and continue to be proposed ([34, 90, 91, 89, 123]).

In the case of the Parallel Adaptive Search method, we are interested in detecting termination as soon as any of the participating threads has found a solution, instead of waiting for all threads to finish their search. Some of them can potentially require too many steps in order to find a solution (it is enough to be trapped in a 'zone' of the search space with no possible solutions).

The implementation of this variant is rather straightforward. Our TDO variant, following the SPMD model of GPI, starts the first instance of a search thread. This first search thread is responsible for spawning its peers on the same node. Hence, the total number of search threads on each node equals the number of available cores on a node.

Adaptive Search starts, by default, with a randomly generated solution. The TDO variant follows the same line and each search thread starts with its own randomly generated initial solution - n search threads perform, ideally, n different walks. This choice is based on the premise that having several threads perform different walks allows, in principle, a good coverage of the search space. Moreover, we diversify initially over the search space instead of relying solely on the diversification mechanisms intrinsic to Adaptive Search, as would be the case if one had started all threads from a single identical initial solution.

In terms of parallelisation, the TDO variant only requires the implementation of a mechanism for triggering and detecting termination. Whenever a search thread finds a solution, it triggers termination by *writing* to its peers that it has found one. Thus, the wall-clock time of the parallel execution should be the time taken by this fastest thread.

All search threads, except the one which found a solution, must detect termination. This entails introducing a communication step in the internal loop of the Adaptive Search algorithm. This means checking whether termination was emitted by any search thread. In case it was, the search thread simply stops its search. Otherwise, it continues with its walk.

The introduction of such a communication step becomes a (small) source of overhead and thus it is only executed every k iteration.

8.4.2 Propagation of Configuration

The experiments in previous work have found that the simple approach to parallelisation, namely, the independent multiple-walk, is insufficient to obtain parallel efficiency on some problems especially when experimenting with a large number of cores. Moreover, exchanging simple information such as the cost leads to no improvement. This result goes to show that this is not a useful metric, at least not by itself: it just says that cost C (better than the current cost) can be achieved but says nothing about when and how to reach it.

Hence, we aim at exchanging more, and more meaningful information, introducing cooperation. By cooperation we mean mechanisms that allow threads to share information about their state and thus benefit from the collective search. Also, we would like to exploit the potential and benefits of GPI and its programming model (one-sided communication, no wait for communication, global access to data, threaded model, etc.) This can be achieved, for instance, by moving towards algorithms which resort to significantly more communication than in previous cases.

One of the most powerful aspects of Local Search is its simplicity. Because of this, it is not obvious what could be considered as the *meaningful* information to be shared and communicated to other threads. One promising candidate which has not yet been tried is the whole current configuration.[1] The final configuration represents the solution when the algorithm stops.

The implementation of the Adaptive Search method which we use deals only with permutation problems – *i.e.,* there is an implicit *all-different* global constraint over all the variables – and thus, a configuration is the permutation vector of the variables in the problem.

We previously observed that introducing cooperation in local search/meta-heuristics implies answering some important questions. Following that approach, we formulated similar questions which we aim at answering more specifically, for our combination of Adaptive Search and GPI.

In short, we posed the following questions:

[1]Because the term *solution* is sometimes misleading, we refer to the current solution as a *configuration.*

1. What to communicate?
2. Who does the communication?
3. When to do the communication?
4. How to do the communication?
5. What to do with the communicated information?

It is impossible to answer some of these questions without carrying out actual experiments. In general, we consider different options although not all of the possible ones. In some cases, we consider a single option.

Our answers to the important questions establish the basis of our approach which we name Propagation of Configuration (PoC).

What to communicate?

The Adaptive Search method (as many other methods) is very simple and includes very few elements that can be communicated.

The proposed option is to communicate a full configuration. To this, we only add the cost of the configuration as it is the evaluation metric. Plus, computing the cost every time a configuration is exchanged is a source of extra (unnecessary) overhead specially if a problem has a large number of variables.

Still, the question remains of which configuration to communicate. In our design we consider the best configuration *i.e.,* the configuration with better cost and that necessarily translates to the current configuration held by a search thread.

Communicating configurations can be of advantage because it implicitly includes more information about the state of the search since it, in a sense, provides a positioning within the whole search space. As the best configurations are being exchanged, other threads that are currently on poorer neighbourhoods might benefit from moving to a better one. Together with the stochastic behaviour of Adaptive Search and enough diversification, the whole search procedure can be performed on the best neighbourhoods and possibly, converge faster onto good solutions.

Who does the communication?

Answering the question of who does the communication involves deciding whether a single thread or all threads actually perform communication. Note that on each node, we consider as many threads as the number of available cores. Communication is performed between nodes, by reading or writing the global memory of GPI. Hence, to answer this question we consider the limit cases, if one or all threads actually take the role of communicating with the other nodes.

There are potential advantages and disadvantages with both options. If all threads perform communication, any shared resources must be protected by a mutual exclusion mechanism, which might suffer from high contention. Moreover, when all threads perform communication, a lot more pressure on the interconnect follows, increasing the parallel overhead and with possibly a lot of redundant communication happening (the same configuration being passed around several times). On the other hand, there will be a rapid progress towards the best promising neighbourhood, intensifying the search. Of course, this can be positive but can also become dangerous since most of threads might get trapped in a local minimum or poor quality neighbourhood. A good trade-off between intensification and diversification needs to be achieved.

If a single master thread communicates, the effects are potentially the opposite: less intensification but also less contention, less pressure on the interconnect and less redundant work. It is important to emphasise that, in this case, although only a single thread is communicating with other nodes, all other search threads are benefiting from the cooperation. On each node and given the GPI programming model, all search threads have access to the global memory and are able to use what is being exchanged between the master threads on all nodes.

When to do the communication?

The first possible answer to this question is to follow the same strategy as with the Termination Detection Only variant: introduce a communication step and perform communication every k iteration. The value of k becomes fundamental on how well this option might perform. With a low value of k (*e.g.,* $k = 10$), a strong intensification of the search is achieved but with the vulnerability that threads might give up too soon on a promising neighbourhood. With a high value of k, the process becomes less vulnerable but less intensification is achieved as less information will be propagated.

The other option is to not interrupt the normal flow of the algorithm for communication, letting the search progress normally and independently until a local minimum is achieved. Only at this point the communication step is entered and the configuration is propagated and possibly used. For problems with a very low number of local minima, we can still consider to have a progress step every k iteration that ensures communication is reaching all search threads.

In summary, we can distinguish six combinations:

1. propagation every k iteration with a master thread doing communication
2. propagation every k iteration with all threads doing communication
3. propagation at local minima **only** with a master thread doing communication
4. propagation at local minima **only** with all threads doing communication
5. propagation at local minima, with progress every k iteration and a master thread doing communication

6. propagation at local minima, with progress every k iteration and all threads doing communication

In principle the options where communication is only done at local minima (options 3 and 4) seem more promising as no disturbance is caused when the algorithm is making good progress. However, there is the possible danger that if threads rarely hit local minima, the propagation of configurations will not progress enough and some threads might never see an up-to-date configuration. One solution to this problem is to still have communication every k iteration, where search threads simply keep the communication progressing but only use the propagated information when they reach a local minimum (options 5 and 6). On the other hand, if a problem has too many local minima, the communication step will be executed much more often, probably redundantly with less real value. In both cases the overhead of communication grows considerably and has to be substantially offset by the gain resulting from it.

Besides communicating configurations, detecting termination still has to be done. The presence of a communication step every k iteration is still existing and can be extended with the communication of configurations. This fact supports both pairs of options (1, 2) and (5, 6).

Ultimately, this question exposes several options and issues which can only be effectively answered empirically.

How to do the communication?

Search threads can be organised differently and exchange information using various methods. To answer this question, we consider a single alternative. Since we aim at large scale executions, we need an efficient approach. Communication is done along a tree-based topology, where each node only communicates with its parent and children (if any). Currently, a binary tree is used but this can be parametrised at initialisation. At each communication step, the propagation of the configuration is done either up (to the parent) or down (to the children) the tree. This only happens if a configuration was propagated from the children (in case of the up direction) or from the parent (down direction). The propagation of the communication behaves then like a wave, up and down the tree, with possibly different configurations being propagated at different points of the tree and contributing to some diversification.

Consider Figure 8.4.1 as an example and visualisation of the mechanism. Each node of the tree corresponds to one node of the parallel system. The colour of each node corresponds to a configuration. For instance, at the root of the tree (node 0), the configuration that will be propagated down to the children is green. On the other hand, the children (nodes 1 and 2) act differently on the propagated configuration: node 2 propagates the same configuration it received from its parent (green) further down to its children but node 1 decides to propagate a different configuration (red). And this mechanism continues down the tree,

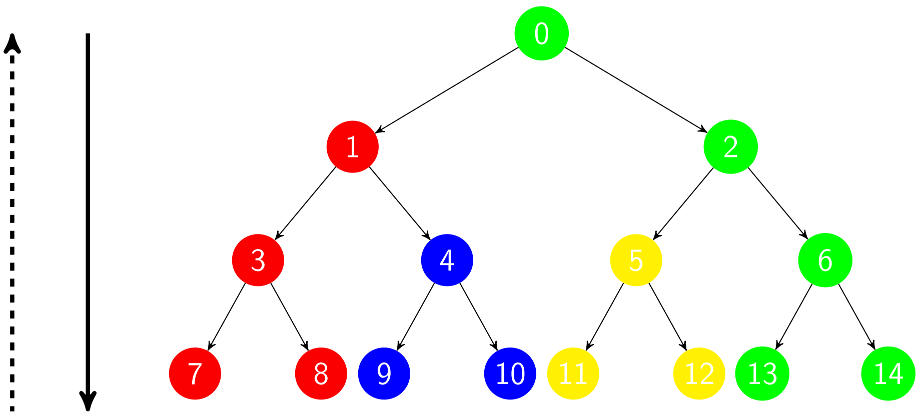


Figure 8.4.1: Communication topology

until the leaves are reached. In the example, four different colours (configurations) are being propagated. The leaves, however, change the direction of the wave of propagation, towards the root and the process continues, up and down the tree.

Communication is performed by using GPI one-sided primitives. A thread posts a write operation and returns immediately to work. The configuration to be propagated will be directly written to the memory of the remote node, asynchronously, without any acknowledgement and overlapped with the algorithm's computation. The remote, on its communication step, checks whether a valid configuration was written to its memory, decides how to act on it and propagates its decision further, in the same direction.

We consider this single alternative aiming at a good balance between intensification and diversification and because having a tree-based topology provides an efficient pattern to achieve communication scalability.

What to do with the information?

When a search thread enters a communication step, it must decide what to do. Obviously, if there is no "incoming" configuration propagated, the search thread immediately resumes its work. In case there is a propagated configuration, the search thread evaluates it and decides: in our PoC variant, if the propagated configuration is better than its current one then that same configuration is further propagated, keeping the "wave" of configurations moving along the topology. If not, the search thread propagates its current configuration instead.

Whether the search thread actually uses the propagated configuration is a different decision. As we have mentioned before, the search thread might keep the propagation wave in progress but, according to its current search state, opt for not using it because its walk is progressing well. On the other hand, if for example, the communication step is entered as a local minimum was reached, then the search thread may take the propagated configuration and use it to continue. In both cases, we only consider the question of using the configuration or not. It would be possible though, to make this decision given a certain probability, adding one more parameter to the algorithm. Other option would be to use the propagated configuration and just to extract some "useful" information and update the current walk.

In our PoC variant when a search thread reaches that point, of deciding if it uses the propagated configuration, it implements a *greedy* approach: if the configuration is better, the search thread simply uses it.

8.5 Experimental Evaluation

In this section we present the results obtained with AS/GPI on different problems. We compare both GPI variants (TDO and PoC) to the MPI and sequential implementation of Adaptive Search.

The experiments were conducted on a cluster system where each node has dual Intel Xeon 5148LV ("Woodcrest") processors (*i.e.,* 4 cores per node) with 8 GB of RAM. The full system is composed of 620 cores connected with Infiniband (DDR). We performed our experiments on the system using up to 256 cores on some problems and 512 cores on others.

8.5.1 The Problems

Our experimental evaluation focuses on known problems, often used as benchmarks. The problems we evaluated are the following:

- **all-interval**: the All Interval Series problem (prob007 in CSPLib),
- **costas-array**: the Costas Array problem,
- **magic-square**: the Magic Square problem (prob019 in CSPLib).

Such problems have different characteristics, with respect to their execution and parametrisation, allowing us to assess AS/GPI on different behaviour and attributes. Moreover, from the previous work with parallel adaptive search, it became evident that different problems behave considerably differently in terms of parallel execution. It is therefore important to understand and try to characterise them in order to be able to draw conclusions from the experimentation with the new parallel variants.

Although Adaptive Search, as any meta-heuristic, has several parameters and each problem is subject to different approaches (models), in our evaluation we decided not to modify either the parameters or the model, as our aim was to keep an "as-is" approach, not interfering with what the domain expert would model and thereby obtain directly comparable results.

All Interval Series

Although looking like a pure combinatorial search problem, this benchmark is in fact a well-known exercise in music composition [130]. The idea is to compose a sequence of N notes such that they are all different and tonal intervals between consecutive notes are also distinct (see Figure 8.5.1).

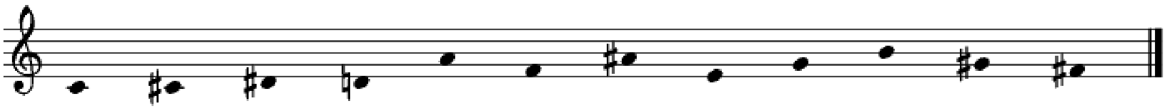


Figure 8.5.1: Example of all-interval in music

This problem is described as `prob007` in the CSPLib [48] and is equivalent to finding a permutation of the N first integers such that the absolute difference between two consecutive pairs of numbers are all different. This amounts to finding a permutation $(X1, \ldots X_N)$ of $\{0, \ldots N-1\}$ such that the list $(abs(X_1 - X_2), abs(X_2 - X_3) \ldots abs(X_{N-1} - x_N))$ is a permutation of $1, \ldots, N-1$.

A possible solution for $N = 8$ is $(3, 6, 0, 7, 2, 4, 5, 1)$ since all consecutive distances are different:

$$3 \overset{3}{—} 6 \overset{6}{—} 0 \overset{7}{—} 7 \overset{5}{—} 2 \overset{2}{—} 4 \overset{1}{—} 5 \overset{4}{—} 1$$

The Adaptive Search model only maintains the list of N variables X_i and ensures it forms a permutation by swapping values inside the list. It is worth noticing that the constraint on the distances (absolute values between each $X_i - X_{i+1}$) is not encoded as a data-structure but ensured via the cost function (this further reduces the amount of data in the local storage). The cost function of a configuration is the largest missing distance (largest distances are hardest to place and thus should be privileged). Obviously, a solution is found when this cost is zero.

Costas Array

A Costas array is an $n \times n$ grid containing n marks such that there is exactly one mark per row and per column and the $n(n-1)/2$ vectors joining every pair of marks are all different. Figure 8.5.2 depicts an example of Costas array of size 5. It is convenient to see the Costas Array Problem (CAP) as a permutation problem by considering an array of n variables $(V_1, \ldots, V_n)$ which forms a permutation of $\{1, 2, \ldots, n\}$. The Costas array above can thus be represented by the array $[3, 4, 2, 1, 5]$.

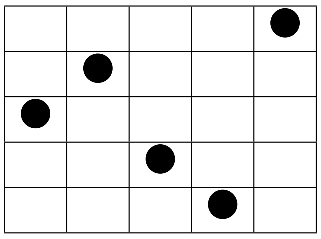


Figure 8.5.2: Costas Array example

Magic Square

The magic square problem is listed as `prob019` in CSPLib [48] and consists in placing the numbers $\{1, 2 \cdots N^2\}$ on an $N \times N$ square, such that the sum of the numbers in all rows, columns and the two diagonal is the same. The constant value that should be the sum of all rows, columns and the two diagonals can be easily computed to be $N(N^2 + 1)/2$.

For instance, here is a solution for $N = 4$:

$$\begin{array}{rrrrl}
 & & & & \quad 34 \\
 & & & & \nearrow \\
\hline
\multicolumn{1}{|r|}{2} & \multicolumn{1}{r|}{13} & \multicolumn{1}{r|}{8} & \multicolumn{1}{r|}{11} & \rightarrow 34 \\ \hline
\multicolumn{1}{|r|}{12} & \multicolumn{1}{r|}{7} & \multicolumn{1}{r|}{14} & \multicolumn{1}{r|}{1} & \rightarrow 34 \\ \hline
\multicolumn{1}{|r|}{15} & \multicolumn{1}{r|}{4} & \multicolumn{1}{r|}{9} & \multicolumn{1}{r|}{6} & \rightarrow 34 \\ \hline
\multicolumn{1}{|r|}{5} & \multicolumn{1}{r|}{10} & \multicolumn{1}{r|}{3} & \multicolumn{1}{r|}{16} & \rightarrow 34 \\ \hline
\downarrow & \downarrow & \downarrow & \downarrow & \searrow \\
34 & 34 & 34 & 34 & \quad 34
\end{array}$$

The modelling for Adaptive Search involves N^2 variables $X_1, \ldots, X_{N^2}$. The error function of an equation $X_1 + X_2 + \ldots + X_k = b$ is defined as the value of $X_1 + X_2 + \ldots + X_k - b$. The combination operation is the sum of the absolute values of the errors. The overall cost function is the addition of absolute values of the errors of all constraints. As with the other problems, a configuration with zero cost is a solution.

8.5.2 Measuring Performance

As mentioned before, the goals of parallel local search and meta-heuristics go beyond the traditional view of performance as just reduction of the execution time. Thus, an experimental evaluation should provide further measurements of some of these characteristics. More than just the execution time, it should allow to assess the robustness and analyse the behaviour of the algorithm.

Adaptive Search has a stochastic nature mostly due to non-deterministic (randomised) decisions taken to achieve diversity on the search. Commonly, this requires an experimental evaluation to measure several statistics over a considerable number of runs. These include the mean, median, standard deviation and the minimum and maximum value of the collected data. In our experiments we executed each problem 100 times in order to obtain meaningful results.

We want to extract more characteristics and aspects of the behaviour of programs, than just the empirical analysis of the execution time. It is important to characterise the benchmark problems from different perspectives: for instance, the performance of Adaptive Search depends on the search space under consideration and some of its properties such as the number of local minima or the presence of *plateaus*. We analyse the problems using different information such as the number of resets and local minima. In general, such analyses give us a view on different properties about the search space structure and neighbourhood graph, which are aspects fundamental to understanding performance and execution behaviour of the algorithm.

To complete our empirical analysis, we observe the run-time distribution (RTD) and run-time length (RTL) as advocated by [66, 65]. Both characterise the run-time behaviour and support more comprehensively the other statistical criteria (mean, median, etc.). Run-time distributions are useful when comparing algorithms, to perform observations that otherwise would be left out and indicate possible improvements such as optimal cut-off for restarts. One often used approach is their approximation by probability distributions. Recall that this is interesting for parallelisation, since if the run-time distribution can be approximated by an exponential distribution, one can expect linear speedups with the simple independent multiple-walk. We observe both the run-time distribution (RTD) and run-time length (RTL). Both are obtained with the cumulative distribution of n observations (in our case $n = 100$). Whereas the RTD uses the wall-clock time (execution time), the RTL uses the number of iterations required for the algorithm to find a solution.

Notwithstanding the need for some of these specific analyses, a reduction of the execution time is still a primary objective. After all, in this thesis we aim at scalability, large scale execution and performance. From the execution time point of view, it is a focus item. In fact, our performance evaluation started in that direction and is therefore here so presented. We measured the obtained speedups with the different variants and compared the results with the MPI implementation (multiple-walk) of Adaptive Search.

8.5.3 Results

The following sections present the results of our experimental evaluation on the previously mentioned benchmark problems. We start by observing the obtained results in terms of speedup and then delve into a more detailed analysis to possibly understand and explain the results we obtained.

For clarity, in some cases we only depict or show the most significant results, pertinent for the presentation and discussion around the experimental evaluation.

As we are interested on the behaviour with larger number of cores, we present the results starting from 16 cores (in our system this corresponds to four nodes). However, the results of the speedup are always relative to the sequential execution.

Magic Square

The Magic Square benchmark was one of the problems that exhibited disappointing scalability when using the simple independent multiple-walk and therefore a major target for improvement with more sophisticated approaches.

The Figure 8.5.3 depicts the obtained speedup results for this problem on up to 512 cores. It depicts the TDO variant, two PoC combinations and the MPI reference implementation.

As expected, our TDO variant performs similarly to the reference MPI implementation, achieving a speedup of $\approx$ 45 on 512 cores. In a sense, we confirm the disappointing scalability results using a simple multiple-walk. On the other hand , for this problem, our PoC variant improves the performance and scales better as we increase the number of cores used.

One question was, *at which point in time should we perform cooperation?* That is, when to do communication: in our experiments it turned out that the best approach is to have a communication step every k iterations where the value of k is decisive. Surprisingly, for this benchmark, a lower value of k (k=10 in contrast to k=1000) improves scalability by a factor of 2, achieving a speedup of 97 with 512 cores. Still a low parallel efficiency but a major improvement over the other options.

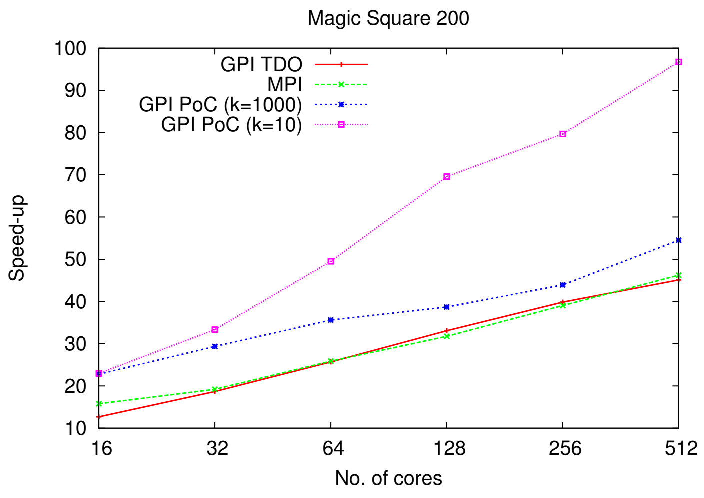


Figure 8.5.3: Magic Square 200 on 512 cores (128 nodes)

Note that for the PoC variant, we only depict two different values for the communication interval k since these two values represent the range of possibilities where the PoC improves over the TDO variant. For example, with k = 100, the obtained speedup lies between the both depicted values (10 and 1000), supporting the observation that a lower value for the interval is better. Furthermore, we also considered other options for when to perform communication. Instead of a communication step every k iterations, performing communication when the search finds a local minimum, with or without progress of the wave of communication, was also evaluated. The outcome was similar in that effectively there is an improvement over the TDO variant but again, lying between both depicted approaches. Finally, introducing cooperation at the moment a local minimum is found yields better results if the wave of propagation of configurations is kept progressing, every few iterations.

Table 8.5.1 presents the results we obtained, in terms of speedup in more detail. We present the speedup based on the average time, as well as when considering only the minimum time (fastest execution) and the maximum time (slowest execution). To consider both extreme cases (fastest and slowest) allows us to have a sense on the degree of dispersion among runs and how it influences the obtained speedup. Considering the slowest execution (*Max*) represents the worst-case while the fastest execution *Min* indicates the best scenario.

The average case represents the already depicted results (Figure 8.5.3) where our PoC variant with $k = 10$ yields the best results. In the best-case (*Min*) we observe a minimal improvement as we increase the number of cores. Here too, the PoC variant ($k = 10$) obtains the best results but only marginally.

On the other hand, when we observe the worst-case (*Max*), we can see that the parallel execution of Adaptive Search results in substantial improvements. This happens in all variants, including the MPI reference implementation. In this case, our PoC variant also obtains the best result with a speedup of 1188. That means the parallel executions narrow the range of execution times and according to this criteria, we can say the parallel version is more robust.

Variant		Cores					
		16	32	64	128	256	512
TDO	Avg	12.68	18.66	25.68	33.07	39.84	45.10
	Min	1.68	2.67	1.70	1.87	2.76	2.85
	Max	88.17	97.50	224.10	350.34	513.22	559.88
PoC (k=10)	Avg	**22.95**	**33.33**	**49.51**	**69.58**	**74.53**	**96.72**
	Min	2.62	3.18	3.34	3.52	3.72	3.86
	Max	90.99	127.86	119.71	342.48	425.25	1187.97
PoC (k=1000)	Avg	22.78	29.36	35.60	38.69	43.92	54.50
	Min	1.63	2.23	2.18	2.44	2.76	2.58
	Max	235.92	297.50	438.53	413.48	574.07	744.53
MPI	Avg	15.78	19.20	25.88	31.75	39.05	46.22
	Min	2.31	2.21	2.08	2.83	2.10	2.85
	Max	137.77	141.41	235.44	335.77	400.05	645.90

Table 8.5.1: Speedup for Magic Square (n=200).

To go further we analysed each problem and its execution more thoroughly. It provides us with more elements about the robustness of the algorithm, how it changes as the number of cores used is increased and on the distribution of execution times. Table 8.5.2 presents basic statistics obtained from our experimental runs. For all the runs, we present the mean (*Avg*), median (*Med*), minimum (*Min*), maximum (*Max*) and standard deviation (*Stdev*) of all run times.

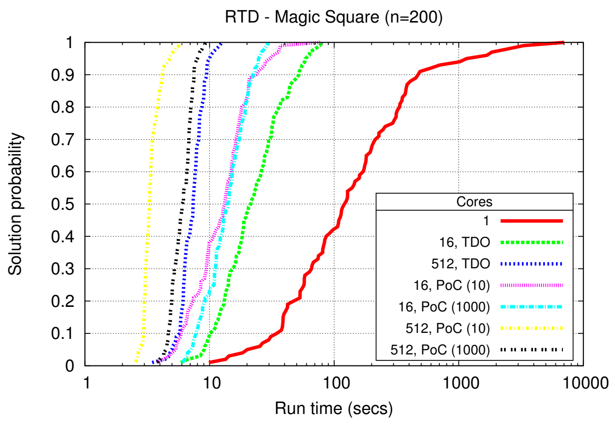
The first observation gathered from Table 8.5.2 is about the standard deviation. For this problem, the sequential execution exhibits a high dispersion, with a very high standard deviation. This dispersion is greatly reduced with the parallel execution of Adaptive Search. The PoC variants improve on the simple TDO variant and according to this dispersion aspect based on the standard deviation, they are more robust.

We included both the average (*Avg*) and the median (*Med*) to have more than one location parameter and thus a better view of the distribution of values. If the distribution is asymmetrically skewed, the median provides a more reliable measure of location as the mean is more sensitive to outliers. In fact, in the sequential execution the value of the median differs greatly from that of the average. This difference is significantly reduced with our parallel variants and the distribution of values becomes more symmetrical and the execution more predictable.

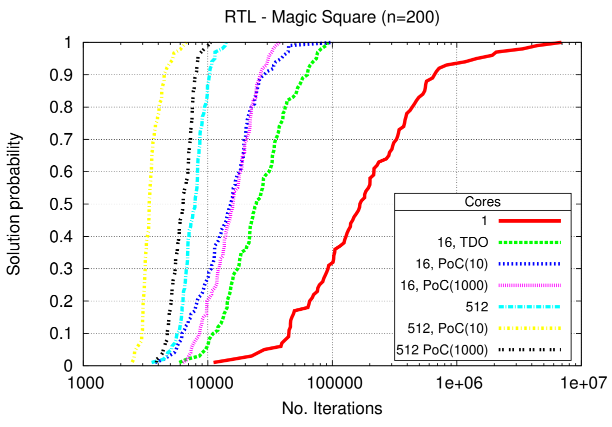
Variant		Cores						
		Seq	16	32	64	128	256	512
TDO	Avg	334.66	26.40	17.93	13.03	10.12	8.40	7.42
	Med	119.03	20.99	16.07	12.19	9.74	8.23	7.39
	Min	9.89	5.88	3.71	5.82	5.28	3.58	3.47
	Max	7020.91	79.63	72.01	31.33	20.04	13.68	12.54
	Stdev	825.37	16.25	10.51	4.89	2.87	2.12	1.60
PoC (k=10)	Avg	-	14.58	10.04	6.76	4.81	4.49	3.46
	Med	-	13.10	8.35	5.72	4.29	3.77	3.29
	Min	-	3.78	3.11	2.96	2.81	2.66	2.56
	Max	-	77.16	54.91	58.65	20.50	16.51	5.91
	Stdev	-	9.56	7.63	5.73	2.15	2.23	0.62
PoC (k=1000)	Avg	-	14.69	11.40	9.40	8.65	7.62	6.14
	Med	-	14.14	10.65	9.41	8.29	7.43	6.04
	Min	-	6.07	4.44	4.54	4.05	3.58	3.83
	Max	-	29.76	23.60	16.01	16.98	12.23	9.43
	Stdev	-	5.55	4.15	2.67	2.25	1.73	1.16

Table 8.5.2: Statistics for Magic Square (n=200).

A more visual perspective is given by Figure 8.5.4, where the RTD and RTL for this problem are depicted. Both allows to examine the behaviour of all our executions, in terms of execution time and the number of total of required iterations. Note that we plotted on semi-log plot to have a view on the full range of variation.



(a) Run Time Distribution (RTD)



(b) Run Time Length (RTL)

Figure 8.5.4: Run time and length distributions - Magic Square (n=200)

Figure 8.5.4 shows the behaviour of the sequential execution (1 core) together with our parallel variants at small scale (16 cores) and large scale (512 cores). In it, we can observe the large variation of runs for the sequential case, both in terms of execution time and iterations (both curves are similar). The curves in the parallel variants become steeper and we can see how each variant positions itself with respect to the others.

From the depicted RTD and RTL, we can do the following observations: theoretically, a local search algorithm can require zero, one or a very large number of steps (or CPU time, respectively) in order to find a solution. From Figure 8.5.4, we can visualise that there is always a minimum number of steps that Adaptive Search needs to perform. This minimum number of steps decreases but not drastically, even though the number of cores is being doubled. At large scale, the RTD and RTL become much more steep (towards an almost vertical line) hinting that the results are very close to the possible limit obtainable with each variant. In fact, these observations proceed on the same direction as when we mentioned that in the best case there is low improvement in terms of speedup, independently of which variants or how many cores are used. In other words, there seems to be a limit, a minimum number of steps required to find a solution for this instance of the problem.

Up to this point, we analysed the Magic Square problem from the point of view of performance, scalability and robustness. Our results indicate that the PoC variant increases the performance and robustness over the TDO variant and MPI reference, both of which are based on an independent multiple-walk. In other words, there is clearly a benefit from cooperation where search threads cooperate by exchanging complete (best) configurations. Surprising though, is the fact there is a larger benefit from performing cooperation every few iterations and that this interval between iterations (k) turns out small. This is somewhat counter-intuitive to what we would imagine as the best solution.

To better understand the results we obtained, we need to observe the characteristics of the problem and the behaviour of its execution. Table 8.5.3 presents what we have collected and shows the average values of different information. The collected information is the following:

Iterations The number of iterations required to find a solution.
Local Minima The number of local minima hit.
Swaps The number of variable swaps performed during execution.
Resets The number of partial resets performed (not full restart).
Same var The number of times in which there existed more than one candidate variable (highest error value) to be selected, per iteration.

Cores	Iterations	Local Min	Swaps	Resets	Same var	Variant
1	418039.51	26123.40	275733.92	3.04	23.60	Sequential
16	17107.45	974.74	7432.41	0.19	39.43	PoC (k=1000)
	17426.16	974.82	8645.67	0.41	44.45	PoC (k=10)
	30828.83	1757.83	11908.05	0.21	36.79	TDO
32	12914.61	692.12	5868.80	0.09	39.06	PoC (k=1000)
	11812.03	644.33	4234.71	0.07	54.90	PoC (k=10)
	20693.44	1149.54	7827.31	0.06	41.72	TDO
64	10313.15	528.91	4866.86	0.00	47.71	PoC (k=1000)
	7612.85	400.48	3267.00	0.03	63.30	PoC (k=10)
	14718.16	800.15	5841.08	0.00	46.27	TDO
128	9423.25	474.16	4791.41	0.02	46.89	PoC (k=1000)
	5298.71	263.45	2530.89	0.01	76.97	PoC (k=10)
	11121.46	586.45	5177.12	0.00	51.72	TDO
256	8175.29	394.47	4319.50	0.00	49.85	PoC (k=1000)
	4888.92	242.30	2442.40	0.01	86.90	PoC (k=10)
	8979.32	464.08	4737.72	0.00	59.90	TDO
512	6397.75	290.12	3852.56	0.00	57.37	PoC (k=1000)
	3605.80	167.99	2165.27	0.00	94.66	PoC (k=10)
	7936.46	398.38	4270.30	0.00	62.95	TDO

Table 8.5.3: Problem characteristics - Magic Square (n=200).

From the sequential execution, we can see that this problem performs a very low number of partial resets when compared to the total number of iterations or even to the number of identified local minima. The number of local minima is also relatively small when compared to the total number of iterations. On the other hand, the number of candidate variables per iteration (Same var) is relatively high, meaning that at each iteration there are several possible moves towards the next configuration.

From the parallel executions, we can see how the number of iterations as well as the number of local minima and variable swaps performed decrease, as the number of cores grows. The number of resets performed is also reduced, converging to zero as the number of cores is increased. This applies to the different parallel variants and is more substantial with the best variant, PoC. Oppositely, the number of moves (Same var) grows together with the number of search threads.

From the collected information we can deduce that each configuration has a dense neighbourhood (several moves) and may benefit from the parallel exploration of different moves. Thus, the PoC variant improves the performance and scalability of the algorithm. When a search thread adopts a propagated configuration, it will define its own path from that configuration and (possibly) differently from some other thread that receives that same promising configuration. Moreover, this problem has a low number of local minima and resets meaning that paths from one (initial) configuration towards an optimal solution are a series of "rarely-interrupted" transitions from neighbour configurations. In other words, there is a relatively dense neighbourhood graph in a smoother landscape.

All-Interval

The All-Interval was another problem that showed scalability issues when run in parallel. Moreover, this problem possesses different characteristics.

As before, we start by showing graphically the obtained speedup using the different variants and the reference MPI implementation. The obtained results for the instance where $n = 400$, are shown in Fig. 8.5.5.

In the case of the PoC variant, we only depict the case where $k = 1000$. With a lower value for the interval of communication the observed speedup is not comparable with the depicted variants and thus we opted to leave it out.

In this problem, we can see that in general the obtained speedup at large scale considering the number of total cores (here up to 256 cores given the smaller problem size) is rather low - maximum of ≈ 30. This is the case for all the different variants. The GPI TDO variant behaves similarly, although slightly better, as the MPI version. Both reach a modest speedup factor of around 26 and 21, respectively. On the other hand, the PoC variant has an alternating behaviour: at small scale, it behaves much worse than the TDO and MPI variants but as the number of cores grows, the difference to both is reduced (even surpassing the MPI version). The speedup curve is steeper and ends very close to the TDO variant.

Table 8.5.4 summarises the obtained speedups and as before, together with the best and worst cases (*Min* and *Max*). Like in the Magic Square problem, the best case does not see significant improvement from employing more search threads. Here too, there seems to be a minimum number of performed moves which cannot be significantly reduced. However, in the worst case, we observe a small difference: the speedup in this case is larger than the average case but not as substantial as in the Magic Square problem and stays considerably below the ideal case. If on one hand it points at a lower level of digression among the different run times, it also hints at the difficulty to obtain high speedups, close to the ideal (linear) case.

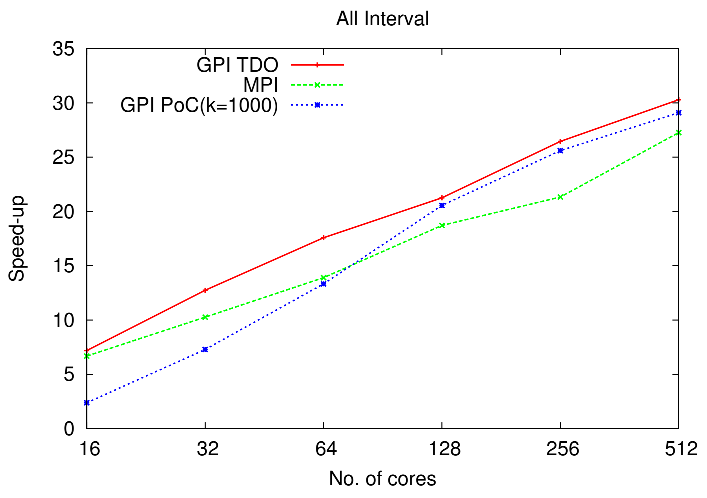


Figure 8.5.5: All Interval 400 on 256 cores (64 nodes)

Variant		Cores					
		16	32	64	128	256	512
TDO	Avg	**7.19**	**12.74**	**17.58**	**21.25**	**26.45**	**30.29**
	Min	2.57	3.62	2.69	2.70	3.84	3.89
	Max	8.04	16.17	33.00	57.80	62.59	89.95
PoC (k=1000)	Avg	2.38	7.29	13.33	20.55	25.60	29.09
	Min	2.14	2.80	3.41	2.63	3.74	3.41
	Max	1.91	5.92	13.06	46.10	59.66	69.30
MPI	Avg	6.68	10.26	13.90	18.71	21.35	27.36
	Min	2.10	2.18	2.25	3.41	3.47	3.45
	Max	7.00	17.84	26.83	44.49	53.48	69.78

Table 8.5.4: Speedup for All Interval (n=400).

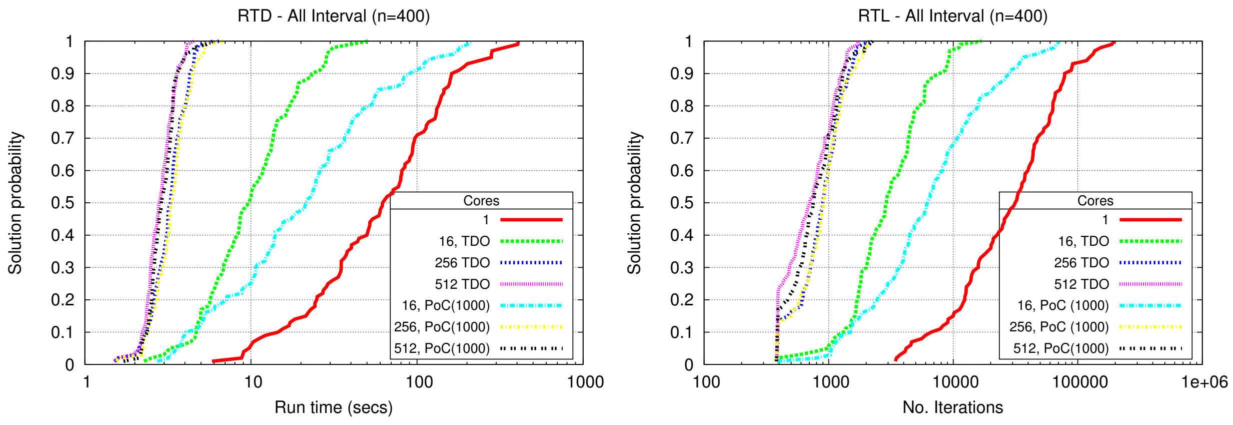
In order to further analyse the results obtained with our GPI-based variants, Table 8.5.5 presents the collected statistics.

Although the obtained speedup appears to be rather disappointing given the large number of cores employed, we can see from Table 8.5.5 that the robustness of the algorithm increases. The standard deviation is gradually reduced as the number of cores grows. For this problem, the difference between the mean (average) and median is smaller as in the Magic Square problem case and both values become closer at large scale. The distribution of values is more symmetrical.

Figure 8.5.6 presents the RTD and RTL of our runs. Here too, we can see that there is a minimum number of required iterations by the sequential execution, which is reduced by the parallel variants. At large scale, both parallel variants revel much more robust as the probability to find a solution grows very quickly.

Variant		Cores						
		Seq	16	32	64	128	256	512
TDO	Avg	87.55	12.17	6.87	4.98	4.12	3.31	2.89
	Med	62.97	9.79	6.16	4.61	4.07	3.25	2.83
	Min	5.83	2.27	1.61	2.17	2.16	1.52	1.50
	Max	407.49	50.70	25.20	12.35	7.05	6.51	4.53
	Stdev	80.96	8.39	3.49	2.12	1.20	0.80	0.57
PoC (k=1000)	Avg	-	36.74	12.01	6.57	4.26	3.42	3.01
	Med	-	21.28	6.17	4.84	3.94	3.31	2.89
	Min	-	2.73	2.08	1.71	2.22	1.56	1.71
	Max	-	212.97	68.80	31.21	8.84	6.83	5.88
	Stdev	-	44.36	13.22	4.77	1.36	0.86	0.63

Table 8.5.5: Statistics for All Interval (n=400).



(a) Run Time Distribution (RTD) (b) Run Time Length (RTL)

Figure 8.5.6: Run time and length distributions - All Interval (n=400)

Comparing the TDO and PoC variants at small scale, we see that although there is some intersection on shorter runs and their probability, the PoC variants requires longer runs to find a solution, having more dispersion on its runs. At large scale (256 and 512 cores), both variants have almost coincident behaviour.

From the previous analysis, we can see that it is hard to expect good speedups from the parallelisation. Moreover, the PoC variant does not perform any better than the TDO variant. This is in high contrast with the previous problem (Magic Square). The All-Interval benchmark problem has different characteristics which do not benefit from the cooperation scheme.

Table 8.5.6 presents the information collected from the runs of the All-Interval problem.

In the All Interval problem the number of resets is equal to the number of local minima. Both are low compared to the number of total iterations. The number of possible variable

Cores	Iterations	Local Min	Swaps	Resets	Same var	Variant
1	41533.63	1436.37	42976.50	1436.37	9.28	Sequential
	11404.14	364.37	11770.16	364.37	6.56	PoC (k=1000)
16	3786.74	78.87	3865.84	78.87	4.60	TDO
	3746.91	98.48	3845.75	98.48	5.30	PoC (k=1000)
32	2179.91	42.57	2222.61	42.57	5.31	TDO
	2075.46	45.01	2120.62	45.01	4.50	PoC (k=1000)
64	1546.34	21.82	1568.23	21.82	3.32	TDO
	1314.42	18.30	1332.76	18.30	4.03	PoC (k=1000)
128	1244.26	18.63	1262.92	18.63	3.50	TDO
	958.06	10.07	968.14	10.07	3.04	PoC (k=1000)
256	934.61	9.82	944.48	9.82	2.77	TDO
	845.53	12.61	858.17	12.61	3.65	PoC (k=1000)
512	776.24	8.91	785.16	8.91	2.58	TDO

Table 8.5.6: Problem characteristics - All Interval (n=400).

choices or moves is higher than one, meaning that some diversification could be achieved but is much lower as in the case of Magic Square problem.

On the parallel executions, the number of iterations is reduced as expected. The number of local minima and resets are also reduced when compared with the total number of iterations. In contrast with the Magic Square problem, however, the number of possible moves shrinks as the number of cores grows.

From the Magic Square problem we have learned that the PoC strategy can benefit from the low number of local minima and resets as well as from the possibility of moves to neighbours. In general, a smooth landscape with a relatively dense neighbourhood graph. The All-Interval problem fulfils this general aspect, at least partially: it also has a small number of local minima and resets and some moves seem to be possible (although in smaller number). However, PoC does not result in better scalability although at large scale it performs as well as TDO.

Only by combining all previous different views can we understand the behaviour of this problem. The RTD and RTL (Figure 8.5.6) draw attention to the almost coincident behaviour of the TDO and PoC variants. Particularly, the RTL makes this noticeable as the behaviour of both variants for short lengths is practically overlapping, which raises the hypothesis that the PoC variant may converge with the TDO variant. The similarity of results becomes more evident after this observation (for example Tables 8.5.5 and 8.5.4) and from the problem characteristics (Table 8.5.6.) We can effectively understand the scalability difference: at large scale, the execution requires, on average, less than 1000 iterations which is the interval value for number of iterations using the PoC variant. In other words, at large scale, PoC behaves as the TDO variant and performs similarly to a independent multiple-walk. This justifies the low performance with a small number of nodes and a much worse performance if the interval k is set to a low value (e.g. $k = 10$).

An in-depth analysis of the execution for this problem confirmed our hypothesis and provided more elements to our analysis: although, on average, there are some possible moves (as we observed from Table 8.5.6), the great majority of the walks are a trajectory with a single move. That is, the neighbourhood graph is not very dense. Hence, there is less benefit from taking a propagated configuration since there is only one trajectory, possibly redundantly taken.

Costas Array

As already observed, the Costas Array Problem (CAP) already shows an almost optimal scalability using an independent multiple-walk with no cooperation. Introducing cooperation to improve on the optimal scalability may seem unnecessary and less significant. Yet, the CAP is another kind of problem with different characteristics and to examine its behaviour might provide more insight on our parallel variants and the results of the other problems.

In fact, our experiments showed that the PoC variant performs much worse than the simple TDO variant. Hence, we only present the speedup obtained using the TDO variant and will analyse this problem with this variant only.

Figure 8.5.7 depicts the speedup results we obtained for the CAP where $n = 20$, using the GPI TDO variant and the MPI reference implementation.

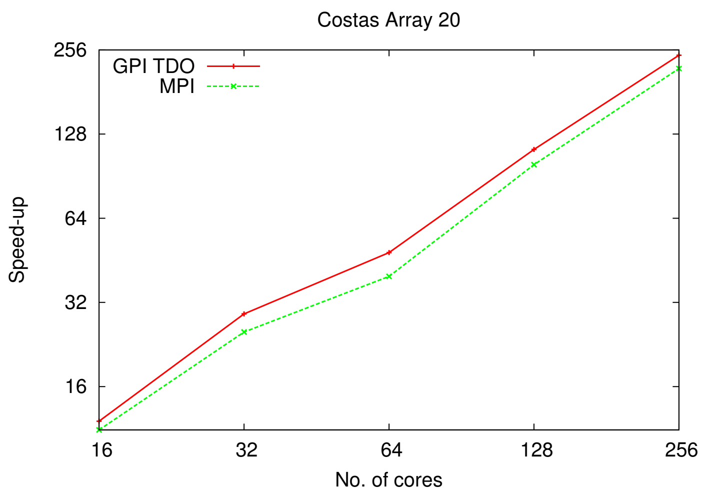


Figure 8.5.7: Costas Array (n=20) on 256 cores (64 nodes)

The CAP obtains almost linear speedup up to 256 cores, using both implementations of an independent multiple-walk.

Table 8.5.7 presents the resulting speedups on average together with the best and worst cases (*Min, Max*). Compared to the previous problems, the CAP shows a different behaviour. The average case, as Figure 8.5.7 already depicted, produces almost linear speedups. However, the best and worst cases result in results opposite to the previous problems. It is the best case (*Min*) which sees higher improvements whereas the worst case (*Max*) obtains speedups also close to linear.

Variant		Cores				
		16	32	64	128	256
TDO	Avg	12.03	29.13	48.24	112.77	245.01
	Min	53.07	743.00	371.50	123.83	743.00
	Max	11.92	35.93	53.13	163.86	266.61
MPI	Avg	11.20	25.08	39.65	99.58	219.72
	Min	14.86	41.28	247.67	185.75	743.00
	Max	10.67	30.74	31.98	133.22	186.65

Table 8.5.7: Speedup for Costas Array (n=20).

When we look at the statistics of this problem (Table 8.5.8), a pattern similar to the previous problems appears. The standard deviation is high on the sequential execution but decreases as more cores are used. In the same line, the difference between the mean and median also shrinks and the distribution of execution times becomes more symmetrical and the algorithm more robust.

Noteworthy though, is the extremely fast execution in the best case. In all parallel executions, the fastest executions corresponds to a few iterations only and a solution is found after just a few milliseconds. This helps to explain the high speedups in the fastest case.

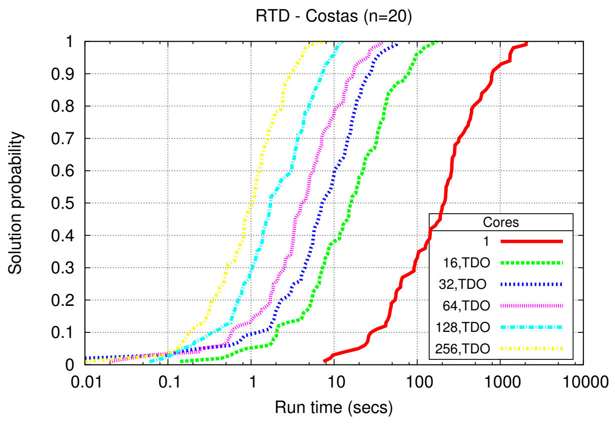
Variant		Cores					
		Seq	16	32	64	128	256
TDO	Avg	340.57	28.32	11.69	7.06	3.02	1.39
	Med	214.63	15.80	7.38	4.00	1.73	1.02
	Min	7.43	0.14	0.01	0.02	0.06	0.01
	Max	2066.23	173.37	57.50	38.89	12.61	7.75
	Stdev	410.83	32.86	11.53	7.71	2.88	1.30

Table 8.5.8: Statistics for Costas Array (n=20).

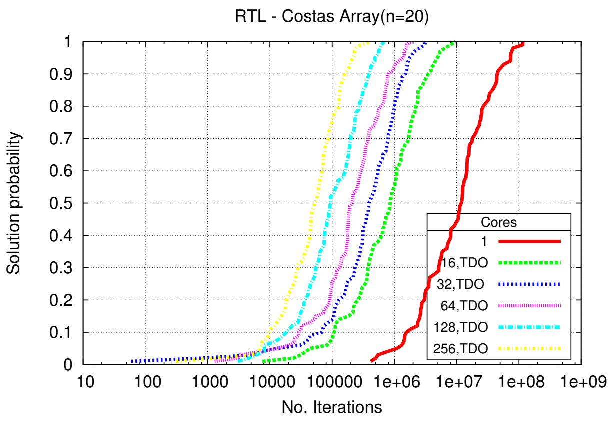
The RTD and RTL of this problem (Figure 8.5.8) also show clearly the good scalability as the number of cores is increased. Both distribution curves move constantly to the left, towards lower execution time and lower number of iterations, respectively.

The characteristics of the CAP are different from the previous problems (Table 8.5.9). In this case, the number of identified local minima is very large (almost every second iteration finds a local minimum) and the number of partial resets is also very high, coincident with the number of local minima *i.e.,* at each local minimum found, a partial reset is performed. Also, the number of possible moves at each iteration is close to 1.

The Costas Array Problem exhibits optimal scalability with the independent multiple-walk MPI version or with our TDO variant and this is already satisfactory *per se*. On the other



(a) Run Time Distribution (RTD)



(b) Run Time Length (RTL)

Figure 8.5.8: Run time and length distributions - Costas Array (n=20)

Cores	Iterations	Local Min	Swaps	Resets	Same var	Variant
1	19080849.03	9549817.37	9246393.04	9549817.37	1.00	Sequential
16	1446313.72	723931.31	700820.89	723931.31	1.00	TDO
32	633831.78	317150.45	307211.78	317150.45	1.00	TDO
64	368505.78	184475.44	178511.19	184475.44	1.00	TDO
128	163640.23	81906.15	79298.64	81906.15	1.00	TDO
256	74990.24	37537.36	36340.40	37537.36	1.00	TDO

Table 8.5.9: Problem characteristics - Costas Array (n=20).

hand, it performs worse with the PoC variant: propagating a configuration turns out to be only a source of parallel overhead and will limit the search allowing less diversification. A propagated configuration will allow, on average, a single move and two threads taking the same configuration results in redundant work which is also likely to be unfruitful since the CAP is one of the problems with a high number of local minima and resets. This also explains the good scalability using the TDO variant, where increasing the number of cores covers more of the total search space together with the fact that solutions for this problem are well spread over it.

Notwithstanding, the analysis of behaviour and characteristics of the CAP allowed us to improve our understanding on how, in general, Adaptive Search may benefit from parallelisation and how our PoC variant may contribute to increase its performance.

8.6 Discussion

The results we obtained in the experimental evaluation presented large differences in how the different problems benefit from parallelism and from the implemented variants. To be able to draw some conclusions on our experiments, we characterised the chosen problems using different perspectives. One of the main objectives was to investigate and understand this discrepancy of results, in as general way as possible.

By relating the characterisation of problems with the obtained experimental results, we can come up with insights which help us in better understanding the parallelisation of these algorithms. Furthermore, we are concerned with the degree to which Adaptive Search may benefit from a communication scheme similar to the one we designed, how to improve on it and which directions should further research focus on.

We argue that one critical aspect is the neighbourhood of a configuration – the set of possible moves – which define transitions between configurations. In other words, the neighbourhood graph of the problem. If a problem has a dense neighbourhood, each of these moves can be explored in parallel. Thus, when a promising configuration (with lower cost) is propagated and several moves are possible, they can be explored in parallel and the probability that one of these moves will lead to a faster path towards an optimal solution increases. On the contrary, if only one move is possible, there is little benefit from using a configuration which was propagated by another search thread and here the algorithm has to rely more on the stochastic behaviour of Adaptive Search to achieve diversification.

Another important aspect is the number of local minima and resets and how they both relate. A problem that finds a large number of local minima (skewed landscape) before encountering an optimal solution benefits less from continuing with a configuration which seems promising. This configuration is heuristically promising but in reality this information is less meaningful than it should. Similarly, a problem with a high number of partial resets suffers from the same issue. To improve on such problems, the configuration should be used differently, as a guide from which other kinds of information can be computed.

There are some aspects that we have not analysed but which are also important to benefit from parallelism. The solutions of a problem (if known and possible) ought to be analysed: their number, density and distribution are important characteristics, with a significant impact on the success of a walk. A problem with solutions uniformly spread over the search space may only require a independent multiple-walk (as we have seen in the case of the CAP) whereas a problem in which solutions are clustered in some part may benefit from more complex mechanisms of cooperation which tend to intensify the search.

We characterised the problem of how to do parallelisation of Adaptive Search and local search as answering structuring questions.

The first question was "**what to communicate?**" and our answer was to *communicate a full configuration.* The configuration used at the moment of communication was the current best configuration. What we have observed is that this a good choice for the case where the configuration can be improved in several ways that is, when several moves are possible. Otherwise, redundant steps will be performed. Moreover, the *goodness* and meaningfulness of the configuration must be high and allow further transitions.

The next question we asked was "**who does the communication?**," considering that each node of the system runs several search threads. Between having a single thread or all threads – on each node – doing communication, our answer is that only a single thread (master) should perform communication and, given the GPI programming model, the other threads can also benefit and contribute.

The answer to "**when to do communication?**" resulted in a slightly counter-intuitive answer. For the case where the PoC variant results in better performance, it was better to have a communication every 10 steps. However, we believe this value is also problem dependent and there is no definitive answer. In our case, only a more careful analysis of the problem will provide more insight into this. Moreover, in a scheme such as the TDO variant, a communication every 10 steps is clearly overkill.

We also only considered one option to "**how to do the communication?**," using a tree structure over which a *wave* of configurations propagates up and down. This choice proved good for at least one the problem (Magic Square), given the fact that more frequent communication resulted in better results and thus an efficient structure was required. Other structures for communication are possible but it is hard to state whether there is more or less benefit from such change.

In our PoC variant, a search thread considers a propagated configuration according to its cost and proceeds with the configuration with the best cost. This is only a good option to the question of "**what to do with the communicated information?**" in case the search thread makes further and significant progress in a different direction than any other thread that used that same configuration. In case the problem has a skewed landscape and/or requires many resets, the propagated configuration should be used differently. Perhaps, by establishing some kind of distance to its own configuration or extract more useful information.

Our answers to the questions are not definitive or final. In fact, they raise other questions and new issues. But our experimental evaluation increased the understanding of how Adaptive Search can benefit from parallelism. Only with careful attention to each problem and its characteristics, could one design strategies to improve the execution in large-scale parallel systems. Such knowledge is valuable for local search methods in general and for the development of new algorithms or implementations of the evaluated problems.

Chapter 9

Conclusion

In this second part of the thesis, we turned our attention to another method of solving hard problem with constraints namely Constraint-Based Local Search. We focused on one particular method, Adaptive Search, for which we developed a new version based on GPI and took different approaches in order to understand its behaviour at large scale.

Local Search methods are incomplete and do not guarantee finding an solution (optimal or otherwise). But they are extremely effective at finding good quality solutions for hard problems and sometimes they are the only viable approach. Constraint-Based Local Search uses constraints to describe and control local search, combining the conceptual clarity of constraints and the effectiveness of local search.

There are numerous different methods for local search but in our work we opted for and focused on Adaptive Search, given its effectiveness and the existing previous work on parallelism. One of our main objectives was to investigate its behaviour in a massively parallel setting, focusing on different benchmark problems, particularly those which, in previous work, showed scalability difficulties when targeting a larger number of cores. The other main objective was to investigate the possibilities afforded by GPI and its programming model to devise more sophisticated mechanisms which would allows us to improve the parallel execution of Adaptive Search.

We tackled the problem of parallelisation in a general way, exploiting GPI for improved results. But local search does not exhibit, at least in a first instance, the common issues when exploiting parallelism such as communication latency or load balancing. We started with scalability as our main target, increasing the speedup at large scale. However, our research evolved to the point of considering other aspects which turned out to be significant.

We presented two variants, TDO and PoC, which aim at answering critical questions when considering parallel execution for local search methods. These questions include issues such as what to communicate or what to do with the communicated information. The TDO variant implements a independent multiple-walk and yields the best results for types of problems such as the Costas Arrays. The PoC variant introduces cooperation between search threads which exchange a full configuration between them, in order to speed the search up. For other types of problem, such as the Magic Square, this variant is an improvement over the other options.

Our experimental evaluation made it clear to what extent different problems benefit from the different variants, presenting results from different perspectives. Besides the initial focus on scalability, we analysed the problems in terms of robustness, their characteristics and how these influence the ability to exploit parallelism. From our results and their discussion, we drew some conclusions and put forth interpretations of the success rate of each variant. In the end, our study provides a general understanding of how Adaptive Search benefits from cooperation that communicates the full configuration and how it could be improved by parallel execution. This knowledge could easily be adapted for local search in general.

We have seen that the characteristics of each problem play a major role on the success of the parallel strategy and there is less room for improvement from the programming model or run-time system point of view. It can be observed that the results obtained with GPI do not yield a very significant improvement over an MPI implementation. Whether a MPI implementation would benefit equally, with the same speedup factors, from a design similar to the one we implemented with GPI, remains an open question. In other words, it would definitely benefit although is not clear whether in the same measure w.r.t. increased scalability. Nevertheless, it is important to emphasise that GPI molded the thinking and design approach to the problem. First, relying on the efficient communication primitives of GPI, one of our initial aims was to *communicate more*. Second, its threaded model allowed us to consider the hierarchy of systems from the beginning, looking at different options such as how each search thread handles cooperation with local and remote peers. Oppositely, the MPI process-based view does not expose that hierarchy. As an example, previous work on parallelising Adaptive Search considered – at least initially – search threads as MPI processes which communicate through messages, independently of their locality (local or remote). Again, not that this considerably changes the picture (hybrid approaches with MPI are certainly possible), but initial presumptions can steer the design and development and here GPI allowed us to design with cooperation seen as a more asynchronous event.

Much can be done to further extend this work. One possibility previously raised, is a more profound study of the benchmark problems concerning the search space and characterisations of the solutions (density, number, distribution). The PoC variant could be extended to include the observations and conclusions we have reached. For example, we could propagate the configuration only when there is more than one move possible or use the configuration differently, extracting other information instead of simply using it (e.g. *no-goods*).

Different but also large problems could be examined to further verify our conclusions. One potential final goal could be the design of a new Local Search or Meta-Heuristic more amenable to parallelisation that builds upon these experiences (based on Adaptive Search or otherwise).

Chapter 10

Concluding Remarks and Directions for Future Work

Given the general availability of multiprocessor systems and the need to modify software to take advantage of such systems, we set to study declarative programming models and their parallelisation using a recent technology from the HPC domain called GPI. We focused on scalability and large scale execution, with the rationale that the number of processing units of a system will continue to increase, while memory access will continue to be hierarchical.

Parallel programming is a difficult task and declarative models are attractive due to the raised programmer productivity ensuing from the increased degree of abstraction with which control is formulated. We focus on models based on Constraint Programming since they have been successfully applied to a wide class of non-trivial and useful problems while having a significant potential for parallelisation.

We divided our work into two major parts, corresponding to two different declarative computational models based on constraints: in Part I, we dealt with complete constraint solving and developed MaCS, a parallel constraint solver based on GPI. In Part II we focused on Constraint-Based Local Search by exploring the Adaptive Search algorithm. Both approaches are very distinct and exhibit different characteristics when it comes to parallelism and its utilisation. We analysed and proposed techniques that allow us to better understand the behaviour at large scale, while trying to take advantage of the GPI programming model.

Part I

We focused on complete constraint solving. We identified the search component as the one with most potential for parallelisation and used the UTS benchmark as a representative of such tree-shaped computations. The GPI implementation of the UTS benchmarks allowed us to address the one of the most daunting challenges for this type of computations:

namely dynamic load balancing. Our results improved over the performance of existing implementations.

We leveraged that work and experience with UTS, together with some ideas from PaCCS, and designed MaCS, a parallel complete constraint solver based on GPI. The self-contained representation of constraint stores from PaCCS, the dynamic load balancing mechanism based on work stealing from the GPI UTS implementation and the architecture of MaCS designed to take advantage of the GPI programming model resulted in a highly scalable parallel constraint solver, effective for a wide range of problems with very diverse characteristics.

A claim which was clearly demonstrated is that the declarative nature of MaCS allows a transparent and effective use of parallel systems, leaving the user out of the low-level implementation details related to parallel execution, while retaining a very effective use of the parallel resources.

Part II

We focused on a different approach: Constraint-based Local Search (CBLS). There are several local search and meta-heuristics exist and we picked a particular one, Adaptive Search. These approaches are incomplete as they do not guarantee to find an (optimal) solution but are often necessary to cope with NP-hard problems.

From the parallelisation point of view, local search is very distinct given its simplicity and sources of parallelism. Our goal was, taking advantage of GPI, to improve (if possible) on the scalability of Adaptive Search and understand its behaviour at large scale. We implemented AS/GPI – Adaptive Search based on GPI – including two variants, TDO and PoC. The TDO variant implements the classic independent multiple-walk approach while the PoC variant introduces cooperation between search threads using a novel communication structure.

CBLS approaches pose unique and difficult challenges for parallel execution. Focusing on the empirical analysis and evaluation of AS/GPI, we enhanced the large differences on how the different problems benefit from the implemented variants, presenting results from different points of view. We analysed the problems in terms of scalability, robustness, their characteristics and how these influence the ability to exploit parallelism. Ultimately, our study provides a general understanding of how Adaptive Search benefits from cooperation and how it could be improved by parallel execution. This knowledge could easily be adapted for local search in general.

One important aspect, related to both approaches, was our goal to evaluate the suitability of GPI to such computational models, whether GPI is of advantage and where it could be improved for such computations. For tree-shaped computations in general and parallel constraint solving in particular, GPI proved a very useful tool and brought about more efficient implementations. In CBLS in general and Adaptive Search in particular, we argued that the characteristics of each problem play a much greater role than the underlying implementation and programming model. Nevertheless, GPI molded the thinking and design approach and allowed us to immediately think of cooperation as a more asynchronous process and was clearly a more scalable approach for some problems.

We believe we covered important aspects related to large scale execution of approaches based on constraints, using a recent programming model. In summary, the main contributions of this thesis are the following:

- the implementation of a scalable dynamic load balancing scheme based on work stealing, appropriate for parallel tree search and tree-shaped computations. It takes advantage of the GPI programming model to implement an hierarchical approach to work stealing (local and remote) and a novel work pre-fetch step, in order to reduce the various sources of overhead. We used the UTS benchmark as inspiration and evaluated it on top of our load balancing implementation on a recent system with up to 3072 cores. The evaluation revealed good performance and greater scalability when compared with other version of UTS.
- a new parallel constraint solver, MaCS, based on some ideas of PaCCS, leveraging from our dynamic load balancing implementation with GPI and a new architecture where only the worker is the driving force. We performed a detailed analysis of MaCS' behaviour on a variety of problems, focusing on the effectiveness of our work stealing implementation. The experimental evaluation on different problems showed good scalability and considerable parallel efficiency on a system with up to 512 cores.
- a new parallel version of Adaptive Search based on GPI, AS/GPI. AS/GPI includes different variants, TDO and PoC. The TDO variant corresponds to the usually used independent multiple-walk. PoC is a novel cooperation approach that communicates a full configuration, using a tree-based structure for communication. Since different problems behave differently, we present a in-depth study, considering different points of view beyond speedup. The study provides a deeper understanding of some scalability problems exhibited by the algorithm when executed at large scale which are easily applied to other methods.

Future Work

The work in this thesis can be extended in many ways. In both Part I and Part II, we pointed directions for future research work on each approach. Moreover, both approaches could be unified and merged onto a hybrid approach. In other words, a local search approach such as Adaptive Search could be integrated in MaCS and provide a combination that would allow quick high-quality solutions with local search and completeness and optimality proving of complete solving.

Bibliography

[1] U.A. Acar, G.E. Blelloch, and R.D. Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12. ACM, 2000.

[2] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] Lourdes Araujo and José J. Ruz. A Parallel Prolog System for Distributed Memory. *Journal of Logic Programming*, 33:49–79, 1997.

[4] A. Arbelaez and Y. Hamadi. Improving parallel local search for SAT. *Learning and intelligent optimization*, pages 46–60, 2011.

[5] R. Bariuso and A. Knies Infiniband Trade Association. *SHMEM User's Guide*, 1994.

[6] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, 2006.

[7] P. Berenbrink, T. Friedetzky, and L.A. Goldberg. The natural work-stealing algorithm is stable. *SIAM Journal on Computing*, 32(5):1260–1279, 2003.

[8] G.S. Bloom and S.W. Golomb. Applications of numbered undirected graphs. *Proceedings of the IEEE*, 65(4):562–570, 1977.

[9] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. 35th Ann. Symp. Found. Comp. Sci.*, pages 356–368, 1994.

[10] D. Bonachea and J. Jeong. GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages. *CS258 Parallel Computer Architecture Project*, Spring 2002.

[11] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Performance Computing and Networking*, 1:91–99, 2004.

[12] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *HIPS*, pages 52–60. IEEE Computer Society, 2004.

[13] Yves Caniou and Philippe Codognet. Communication in Parallel Algorithms for Constraint-Based Local Search. In *IPDPS Workshops*, pages 1961–1970, 2011.

[14] Yves Caniou, Philippe Codognet, Daniel Diaz, and Salvador Abreu. Experiments in Parallel Constraint-Based Local Search. In *EvoCOP*, pages 96–107, 2011.

[15] V. Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[16] Lei Chai, Albert Hartono, and Dhabaleswar K Panda. Designing high performance and scalable MPI intra-node communication support for clusters. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10. IEEE, 2006.

[17] S. Chakrabarti and K. Yelick. Randomized load-balancing for tree-structured computation. In *IEEE Scalable High Performance Computing Conf.*, pages 666–673, 1994.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V.Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. Conf. on Object Oriented Prog. Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005.

[19] G. Chu, C. Schulte, and P.J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, pages 226–241. Springer-Verlag, 2009.

[20] Philippe Codognet and Daniel Diaz. Yet Another Local Search Method for Constraint Solving. In *SAGA*, pages 73–90, 2001.

[21] Philippe Codognet and Daniel Diaz. An Efficient Library for Solving CSP with Local Search. In T. Ibaraki, editor, *MIC'03, 5th International Conference on Metaheuristics*, 2003. `http://pauillac.inria.fr/~diaz/adaptive/`.

[22] MPI Committee. MPI3-RMA: A flexible, high-performance RMA interface for MPI (draft2), December 2008.

[23] G. Cong, S. Kodali, S. Krishnamoorty, D. Lea, V. Saraswat, and T. Wen. Solving irregular graph problems using adaptive work-stealing. In *In Proc. 37th Int Conf. on Parallel Processing (ICPP)*, Portland, OR, Sept. 2008.

[24] UPC Consortium. UPC language specifications, v1.2. Technical report, Lawrence Berkeley National Lab, 2005.

[25] Teodor Gabriel Crainic, Michel Toulouse, and Michel Gendreau. Toward a Taxonomy of Parallel Tabu Search Heuristics. *INFORMS Journal on Computing*, 9(1):61–72, 1997.

[26] TeodorGabriel Crainic and Michel Toulouse. Parallel meta-heuristics. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 497–541. Springer US, 2010.

[27] T.G. Crainic, T. Davidović, and D. Ramljak. Designing Parallel Meta-Heuristic Methods. Technical report, CIRRELT, June 2012.

[28] Mauro Dell'Amico and Marco Trubian. Applying tabu search to the job-shop scheduling problem. *Annals OR*, 41(3):231–252, 1993.

[29] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challanges in dynamic load balancing. *J. Appl. Numer. Math.*, 52(2-3):133–152, 2005.

[30] Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St. John, and Courtenay Vaughan. Design of dynamic load-balancing tools for parallel applications. In *Proc. of the 14th international conference on Supercomputing (ICS '00).*, pages 110–118, ACM, New York, NY, USA, 2000.

[31] D. Diaz, F. Richoux, Y. Caniou, P. Codognet, and S. Abreu. Parallel local search for the Costas Array Problem. In *Parallel Computing and Optimization (PCO)*, 2012.

[32] D. Diaz, F. Richoux, P. Codognet, Y. Caniou, and S. Abreu. Constraint-Based Local Search for the Costas Array Problem. In *Learning and Intelligent OptimizatioN Conference (LION 6)*, 2012.

[33] Daniel Diaz, Salvador Abreu, and Philippe Codognet. Parallel Constraint-Based Local Search on the Cell/BE Multicore Architecture. In *IDC*, pages 265–274, 2010.

[34] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Inf. Process. Lett.*, 16(5):217–219, 1983.

[35] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *Proc. of 6th Intl. Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS)*, pages 1–8, 2007.

[36] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. A message passing benchmark for unbalanced applications. *J. Simulation, Modelling Practice and Theory*, pages 1177–1189, 2008.

[37] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. Scalable Work Stealing. In *Proc. 21st Intl. Conference on Supercomputing (SC)*, November 2009.

[38] M. Dorigo and G. Di Caro. Ant colony optimization: a new meta-heuristic. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2. IEEE, 1999.

[39] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[40] Lígia Ferreira. *Programação por Restrições Distribuídas em Java.* PhD thesis, Universidade de Évora, 2004.

[41] Lígia Ferreira and Salvador Abreu. Design for AJACS, yet another Java constraint programming framework. In *Electronic Notes in Theoretical Computer Science*, pages 167–178, June 2001.

[42] C. Fleurent and F. Glover. Improved constructive multistart strategies for the quadratic assignment problem using adaptive memory. *INFORMS Journal on Computing*, 11(2):198–204, 1999.

[43] T. Frühwirth and S. Abdennadher. *Essentials of constraint programming.* Springer, 2003.

[44] Edgar Gabriel, Graham Fagg, George Bosilca, Thara Angskun, Jack Dongarra, Jeffrey Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 353–377, 2004.

[45] P. Galinier and J.K. Hao. A general approach for constraint solving by local search. *Journal of Mathematical Modelling and Algorithms*, 3(1):73–88, 2004.

[46] Michel Gendreau, Alain Hertz, and Gilbert Laporte. A Tabu Search Heuristic for the Vehicle Routing Problem. *Management Science*, 40:1276–1290, 1994.

[47] Michel Gendreau and Jean-Yves Potvin. Tabu search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 41–59. Springer US, 2010.

[48] Ian P. Gent and Toby Walsh. CSPLIB: A Benchmark Library for Constraints. In *CP*, pages 480–481, 1999. `http://www.csplib.org`.

[49] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.

[50] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and cybernetics*, 39(3):653–684, 2000.

[51] Fred Glover. Tabu Search - Part I. *INFORMS Journal on Computing*, 1(3):190–206, 1989.

[52] Fred Glover. Tabu Search - Part II. *INFORMS Journal on Computing*, 2(1):4–32, 1990.

[53] Fred Glover and Claude McMillan. The general employee scheduling problem. An integration of MS and AI. *Computers & OR*, 13(5):563–573, 1986.

[54] T. Gonzalez, editor. *Handbook of Approximation Algorithms and Metaheuristics.* Chapman and Hall / CRC, 2007.

[55] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *Knowledge and Data Engineering, IEEE Transactions on*, 11(1):28–35, 1999.

[56] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[57] Daniel Grunewald. BQCD with GPI: A case study. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 388–394. IEEE, 2012.

[58] John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[59] Youssef Hamadi. Optimal Distributed Arc-Consistency. *Constraints - An International Journal*, 7:367–385, 2002.

[60] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.

[61] Pascal Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In *International Conference on Logic Programming/Joint International Conference and Symposium on Logic Programming*, pages 165–180, 1989.

[62] P.V. Hentenryck and L. Michel. *Constraint-based local search.* The MIT Press, 2009.

[63] Alain Hertz and D de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

[64] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1992.

[65] Holger Hoos. On the Empirical Evaluation of Las Vegas Algorithms - Position Paper. Technical report, University of British Columbia, 1999.

[66] Holger H. Hoos and Thomas Stützle. Evaluating Las Vegas Algorithms: Pitfalls and Remedies. In *UAI*, pages 238–245, 1998.

[67] Wei Huang, Gopalakrishnan Santhanaraman, H-W Jin, Qi Gao, and Dhabaleswar K Panda. Design of high performance MVAPICH2: MPI2 over InfiniBand. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 43–48. IEEE, 2006.

[68] T. Ibaraki, K. Nonobe, and M. Yagiura, editors. *Metaheuristics: Progress as Real Problem Solvers.* Springer Verlag, 2005.

[69] M. Krishnan J. Nieplocha, V. Tipparaju and D. Panda. High Performance Remote Memory Access Comunications: The ARMCI Approach. *International Journal of High Performance Computing and Applications*, 20(2):233–253, 2006.

[70] R.J. Harrison J. Nieplocha and R.J Littlefield. Global Arrays: A portable shared memory programming model. *Proceedings Supercomputing 94*, 1994.

[71] Joxan Jaffar, Andrew E. Santosa, Roland H. C. Yap, and Kenny Qili Zhu. Scalable Distributed Depth-First Search with Greedy Work Stealing. In *International Conference on Tools with Artificial Intelligence*, pages 98–103, 2004.

[72] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang, and Barbara Chapman. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*, 37(9):562–575, 2011.

[73] Guy L. Steele Jr. Parallel Programming and Parallel Abstractions in Fortress. In *IEEE PACT*, page 157. IEEE Computer Society, 2005.

[74] J. Nieplocha K. Parzyszek and R.A. Kendall. A generalized portable SHMEM library for high performance computing. *Proc. Parallel and Distributed Computing and Systems*, 2000.

[75] Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275 – 286, 1990.

[76] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter*, pages 115–132, 1994.

[77] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

[78] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[79] Martin Kühn. Parallelization of an Edge-and Coherence-Enhancing Anisotropic Diffusion Filter with a Distributed Memory Approach Based on GPI. *Competence in High Performance Computing 2010*, pages 99–110, 2012.

[80] Vipin Kumar and Anshul Gupta. Analysis of scalability of parallel algorithms and architectures: a survey. In *Proceedings of the 5th international conference on Supercomputing*, pages 396–405. ACM, 1991.

[81] Grama A.Y. Kumar V. and Vempaty N.R. Scalable load balancing techniques for parallel computers. *J. Par. Dist. Comp.*, 22(1):60–79, 1994.

[82] Y.K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[83] A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):pp. 497–520, 1960.

[84] Youssed Hamadi Lucas Bordeaux and Horst Samulowitz. Experiments with massively parallel constraint solving. In *Twenty-First International Joint Conference on Artificial Intelligent IJCAI-09*, pages 443–48, July 2009.

[85] TV Luong, N. Melab, and EG Talbi. Parallel Local search on GPU. *INRIA, Lille, France, Tech. Rep. RR*, 6915, 2009.

[86] C. E. Leiserson M. Frigo and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. Conf. on Prog. Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[87] Rui Machado and Carsten Lojewski. The Fraunhofer Virtual Machine: a communication library and runtime system based on the RDMA model. In *Computer Science-Research and Development*, volume 23(3), pages 125–132, 2009.

[88] Rui Machado, Carsten Lojewski, Salvador Abreu, and Franz-Josef Pfreundt. Unbalanced tree search on a manycore system using the GPI programming model. *Computer Science-Research and Development*, 26(3):229–236, 2011.

[89] Jeff Matocha and Tracy Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, 1998.

[90] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987. 10.1007/BF01782776.

[91] Friedemann Mattern. Global Quiescence Detection Based on Credit Distribution and Recovery. *Inf. Process. Lett.*, 30(4):195–200, 1989.

[92] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. *SIGPLAN Not.*, 44(4):45–54, February 2009.

[93] L. Michel and P.V. Hentenryck. Localizer. *Constraints*, 5(1):43–84, 2000.

[94] L. Michel and P.V. Hentenryck. A constraint-based architecture for local search. In *ACM SIGPLAN Notices*, volume 37, pages 83–100. ACM, 2002.

[95] L. Michel and P.V. Hentenryck. Comet in context. In *Proceedings of the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge: Paris C. Kanellakis memorial workshop on the occasion of his 50th birthday*, pages 95–107. ACM, 2003.

[96] L. Michel, A. See, and P. Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS Journal on Computing*, 21(3):363–382, 2009.

[97] Nenad Mladenovic and Pierre Hansen. Variable neighborhood search. *Computers & OR*, 24(11):1097–1100, 1997.

[98] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009. available at: `http://www.mpi-forum.org`.

[99] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0, September 21th 2012. available at: `http://www.mpi-forum.org`.

[100] Shyam Mudambi and Joachim Schimpf. Parallel CLP on Heterogeneous Networks. In *International Conference on Logic Programming/Joint International Conference and Symposium on Logic Programming*, pages 124–141, 1994.

[101] Ankur Narang, Abhinav Srivastava, Ramnik Jain, and R. K. Shyamasundar. Dynamic Distributed Scheduling Algorithm for State Space Search. In Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par*, volume 7484 of *Lecture Notes in Computer Science*, pages 141–154. Springer, 2012.

[102] M. Newton, D. Pham, A. Sattar, and M. Maher. Kangaroo: An efficient constraint-based local search system using lazy propagation. *Principles and Practice of Constraint Programming–CP 2011*, pages 645–659, 2011.

[103] T. Nguyen and Yves Deville. A Distributed Arc-Consistency Algorithm. *Science of Computer Programming*, 30:227–250, 1998.

[104] R. Numrich and J.K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 1998.

[105] Kei Ohmura and Kazunori Ueda. c-sat: A Parallel SAT Solver for Clusters. In *SAT*, pages 524–537. Springer Verlag, 2009.

[106] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An Unbalanced Tree Search Benchmark. In *Proc. 19th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2006.

[107] Stephen Olivier and Jan Prins. Scalable Dynamic Load Balancing Using UPC. In *Proc. of 37th International Conference on Parallel Processing (ICPP-08). Portland, OR*, 2008.

[108] D. Bonachea P. Hilfinger. *Titanium language reference manual*, 2005.

[109] Vasco Pedro. *Constraint Programming on Hierarchical Multiprocessor Systems.* PhD thesis, Universidade de Évora, 2012.

[110] L. Perron. Practical parallelism in constraint programming. In *Proceedings of CP-AI-OR 2002*, pages 261–276, 2002.

[111] Laurent Perron. Search Procedures and Parallelism in Constraint Programming. In Joxan Jaffar, editor, *CP*, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1999.

[112] M. Prais and C.C. Ribeiro. Reactive GRASP: An application to a matrix decomposition problem in TDMA traffic assignment. *INFORMS Journal on Computing*, 12(3):164–176, 2000.

[113] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[114] V. Rao and V. Kumar. Superlinear speedup in parallel state-space search. In *Foundations of Software Technology and Theoretical Computer Science*, pages 161–174. Springer, 1988.

[115] V. Nageshwara Rao and Vipin Kumar. On the Efficiency of Parallel Backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4:427–437, 1993.

[116] Kaushik Ravichandran, Sangho Lee, and Santosh Pande. Work Stealing for Multi-core HPC Clusters. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par (1)*, volume 6852 of *Lecture Notes in Computer Science*, pages 205–217. Springer, 2011.

[117] Carl Christian Rolf and Krzysztof Kuchcinski. State-copying and Recomputation in Parallel Constraint Programming with Global Constraints. In *Parallel, Distributed and Network-Based Processing*, pages 311–317, 2008.

[118] Carl Christian Rolf and Krzysztof Kuchcinski. Parallel Consistency in Constraint Programming. *PDPTA '09: The 2009 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2:638–644, July 2009.

[119] Carl Christian Rolf and Krzysztof Kuchcinski. Combining Parallel Search and Parallel Consistency in Constraint Programming. In *International Conference on Principles and Practice of Constraint Programming: TRICS workshop*, September 2010.

[120] A. Ruiz-Andino, L. Araujo, F. Sáenz, and J. Ruz. Parallel execution models for constraint programming over finite domains. *Principles and Practice of Declarative Programming*, pages 134–151, 1999.

[121] Alvaro Ruiz-andino, Lourdes Araujo, Fernando Sáenz, and José J. Ruz. Parallel Arc-Consistency for Functional Constraints. In *International Conference on Logic Programming/Joint International Conference and Symposium on Logic Programming*, pages 86–100, 1998.

[122] A. Philips S. Minton, M. Johnston and P. Laird. Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[123] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar B. Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In *PPOPP*, pages 201–212, 2011.

[124] Christian Simmendinger, Jens Jägersküpper, Rui Machado, and Carsten Lojewski. A PGAS-based Implementation for the Unstructured CFD Solver TAU. In *5th Conference on Partitioned Global Address Space Programming Models*, 2011.

[125] Jadranka Skorin-Kapov. Tabu Search Applied to the Quadratic Assignment Problem. *INFORMS Journal on Computing*, 2(1):33–45, 1990.

[126] Erich Strohmaier and Hongzhang Shan. Apex-Map: A Global Data Access Benchmark to Analyze HPC Systems and Parallel Programming Paradigms. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 49–, Washington, DC, USA, 2005. IEEE Computer Society.

[127] Thomas G. Stützle. *Local Search Algorithms for Combinatorial Problems - Analysis, Improvements and Applications.* PhD thesis, Technischen Universität Darmstadt, 1997.

[128] D. Teodorovic, P. Lucic, G. Markovic, and M.D. Orco. Bee colony optimization: principles and applications. In *Neural Network Applications in Electrical Engineering, 2006. NEUREL 2006. 8th Seminar on*, pages 151–156. IEEE, 2006.

[129] M. Toulouse, T.G. Crainic, and M. Gendreau. Communication issues in designing cooperative multi-thread parallel searches. Technical report, Citeseer, 1995.

[130] Charlotte Truchet and Philippe Codognet. Musical constraint satisfaction problems solved with adaptive search. *Soft Comput.*, 8(9):633–640, 2004.

[131] T. Van Luong, E. Taillard, N. Melab, and E.G. Talbi. Parallelization Strategies for Hybrid Metaheuristics Using a Single GPU and Multi-core Resources. *Parallel Problem Solving from Nature-PPSN XII*, pages 368–377, 2012.

[132] M. Verhoeven and E. Aarts. Parallel local search. *Journal of Heuristics*, 1(1):43–65, 1995.

[133] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[134] Feng Xie and Andrew Daveport. Solving scheduling problems using parallel message-passing based constraint programming. In *Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, pages 53–58, September 2009.

[135] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32(1):565–606, 2008.

[136] Jigang Yang and Scott D. Goodwin. High Performance Constraint Satisfaction Problem Solving: State-Recomputation versus State-Copying. In *International Symposium on High Performance Computing Systems*, pages 117–123, 2005.

[137] X.S. Yang. Firefly algorithms for multimodal optimization. *Stochastic algorithms: foundations and applications*, pages 169–178, 2009.

[138] X.S. Yang. A new metaheuristic bat-inspired algorithm. *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, pages 65–74, 2010.

[139] X.S. Yang and S. Deb. Cuckoo search via Lévy flights. In *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 210–214. IEEE, 2009.

[140] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering, IEEE Transactions on*, 10(5):673–685, 1998.

[141] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

[142] Ying Zhang and Alan K. Mackworth. Parallel and Distributed Finite Constraint Satisfaction: Complexity, Algorithms and Experiments. In *Third IEEE Symposium on Parallel and Distributed Processing*, pages 394–397, 1994.

Instituto de Investigação e Formação Avançada - IIFA

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