On the Scalability of Constraint Programming on Hierarchical Multiprocessor Systems

Rui Machado
CC-HPC - Fraunhofer ITWM
Kaiserslautern, Germany
rui.machado@itwm.fhg.de

Vasco Pedro
Universidade de Évora
Évora, Portugal
vp@di.uevora.pt

Salvador Abreu
Universidade de Évora
Évora, Portugal
spa@di.uevora.pt

***Abstract*—Recent developments in computer architecture progress towards systems with large core count, which expose more parallelism to applications, creating a hierarchical setup at the node and cluster levels. To take advantage of all this parallelism, applications must carefully exploit the different levels of the system which, if ignored, may yield surprising results. This aggravates the already difficult task of parallel programming.**

Declarative approaches such as those based on constraints are attractive to parallel programming because they concentrate on the logic of the problem. They have been successfully applied to hard problems, which usually involve searching through large problem spaces, a computationally intensive task but with potential for parallelization. Tree search algorithms play an important role in research areas such as constraint satisfaction or optimisation, and artificial intelligence. Tree search lends itself naturally to parallelization by exploiting different branches of the tree but scalability may be harder to achieve due to the high dynamic load balancing requirements.

In this paper we present a high-level declarative approach based on constraints and show how it benefits from an efficient dynamic load balancing based on work stealing targeted at large-scale. We focus on the implementation of a hierarchical work stealing scheme using a different programming model, GPI. Experimentation brought encouraging results on up to 512 cores on large instances of satisfaction and optimisation problems.

I. Introduction

Current computer architectures 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.

Declarative computational models can simplify the modeling 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 constraints programming have been successfully applied to hard combinatorial problems, which usually entail exploring large search spaces, a computationally intensive task, but one with significant potential for parallelization. This situation has been recognised as witness several recent efforts to automatically exploit the inherent parallelism found in constraint solving problems, be it with local search methods [4], [5], [6] or propagation-based complete solvers [8], [7].

Many applications solve a problem which is similar to the traversal of a large implicitly defined tree. Tree search lends itself naturally to parallelization by independently exploring different branches of the tree. This is an interesting problem, from the parallelization point of view, not only because high scalability is possible but given the challenges posed to the load balancing scheme. The load balancing scheme must ensure that all processing elements are active without prior knowledge concerning the shape of the tree, with a dynamic and irregular generation/granularity of work and communication.

In this paper, we present a system – MaCS – that provides the declarative model of constraint programming and that copes well with the increasing parallelism of current systems. The system is implemented on top of GPI, a PGAS API that focuses on one-sided and asynchronous communication.

This paper is organised as follows: in section II we shortly present Constraint Programming (CP). In section III we introduce GPI, the framework used to address the problem. We then describe MaCS, a novel parallel complete constraint solver in section IV and present the results of our performance evaluation in section VI. Finally, we conclude our work in section VIII and discuss some future work.

II. Constraint Programming

The idea of constraint programming is to solve problems by stating constraints (properties, conditions) involving variables, which must be satisfied by the solution(s) of the problem. One then lets a *solver* figure out adequate values for the variables.

Constraints can be viewed as 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* is used to find solutions for the problem.

The first step in solving a given problem, 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 and indexed set of finite sets of values, where each D_i is 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 Traveling 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)

After correctly modelling the problem at hand, a *constraint solver* is used to get solutions for the CSP. 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*.

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

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.

A. Parallel Constraint Solving

Due to the declarative nature of Constraint Programming, taking advantage of parallelism should be possible to achieve in the solving step. The modeling step should remain unchanged, as well as the user's awareness of the underlying implementation.

Parallelizing 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 parallelization of constraint solving has been handled in various ways, where the different parts of the process have been experimented with, as subjects of parallelization.

An obvious candidate for parallelization is *propagation*, where several propagators are evaluated in parallel. For problems where propagation consumes most of the computation time, good speedups may be achieved in this way. On the other hand, the overhead 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.

Another natural and common candidate, as found in the literature, is to parallelize the search. Search consists in walking 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 concerned with doing so at large scale.

III. GPI

GPI[1] (Global address space Programming Interface) is a PGAS API for parallel applications running on clusters [2].

An important idea behind GPI is the use of one-sided communication in which the programmer is encouraged 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 computation to overlap with communication then, only a small amount of time is potentially spent waiting for transfers to finish. As only one side of the communication

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

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 evolving traffic patterns, this can be very useful.

The thin communication layer in GPI 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 with the computation. While latency could and should be hidden by overlapping it with useful computation, data movement gets reduced since no intermediate buffers are 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 constitutes a better mapping to current systems with hierarchical memory levels and heterogeneous components, than offered by, say, MPI. 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 common cluster systems. Figure 1 depicts the architecture of GPI.

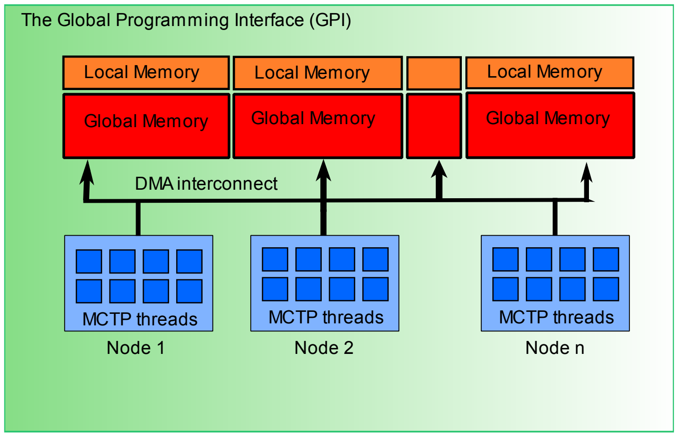


Fig. 1. GPI architecture

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 regular 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 is 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 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 by means of *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.

In the context of this work, the most important functionality is the read/write of the global memory. 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 (read or write), it needs to call a wait operation that blocks until the transfer has finished, asserting that the data is usable.

IV. MaCS

MaCS is a parallel complete constraint solver based on GPI. It is a fork from PaCCS (*Parallel Complete Constraint Solver*) [7]. Both PaCCS and MaCS are parallel constraint programming libraries that aim at exploiting the features of each programming model.

PaCCS was designed from the ground up, with parallel execution on a network of multiprocessors in mind and exhibits good scalability on the different systems. PaCCS is implemented with MPI where a distinguished process initiates the search, collects solutions, detects termination and returns answers. In PaCCS, load-balancing is achieved by means of *work-stealing*. By this we mean that when a search agent has covered its assigned search space, it then tries to obtain another search space from the other agents in the system. The idle agent first tries to obtain work from an agent in its immediate neighbourhood, constituted by the agents in the same shared-memory system. Failing that, it then expands the considered neighbourhood until it encompasses the whole parallel search system. Work-stealing outside the agent's immediate neighbourhood is done by proxy, with work requests having a commom origin becoming aggregated.

MaCS on its hand, although building upon PaCCS, is a more recent implementation. Its main objective is to provide an efficient and scalable constraint solver that takes advantage of parallel systems, using GPI and its programming model. It aims at large scale, 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 if it can be of advantage when implementing parallel constraint solvers targeted at large scale.

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 and compact representation is essential in a distributed setting and definitely 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.

Worker

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.

Worker Pool

A central aspect of the architecture of MaCS is the worker 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.

We wanted to leverage our previous work with UTS and general parallel tree search [1]. Since that solution has proved scalable to implement dynamic load balancing with work stealing, the worker pool uses the same data structure used in that work.

Figure 2 depicts the worker pool again, now in a closer view.

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

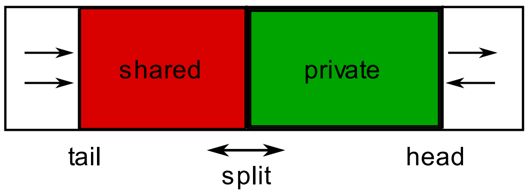


Fig. 2. Worker pool

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 worker 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 worker pools are placed in the GPI global memory, for global availability.

V. Dynamic Load Balancing

Our implementation is based on the observation that 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, MaCS leverages our previous work with the UTS benchmark [1], which aims exactly at characterising such mechanism.

The current search path may be invalid. A worker therefore invokes a 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 worker 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 worker pool (Figure 2) is divided in two regions: private and shared. The restore procedure tries to first retrieve a new store from the private region of the worker 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 worker 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.

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 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 worker 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 worker 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 worker 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 use 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 (possibly) their negative effect; if a poll succeeds, the opposite happens and the polling interval becomes more frequent.

VI. Experimental Evaluation

In this section, we present the results obtained with MaCS and PaCCS for different problems.

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). Given the machine availability, we performed our experiments on the system using up to 512 cores.

Problems

In our experimental evalution we used different problems, both representative of satisfaction and optimisation problems.

The first used problem is the well-know N-Queens problem. 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 so that it's not possible for a queen to attack one other one on the board. This means no pair of queens can share a row, a column or a diagonal and that these are our constraints.

The Quadratic Assignment Problem (QAP) is a 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.

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 3 depicts how much time, on average, is spent by workers on each of the major states.

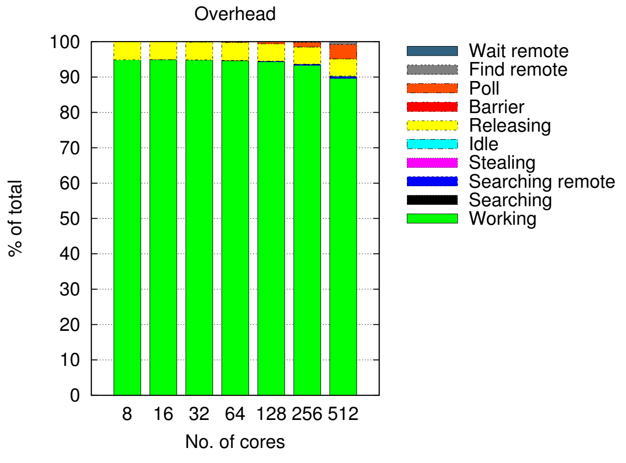


Fig. 3. Working time and Overhead

From Figure 3 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 worker 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 4c 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.

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 problems 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 number of nodes generated and consumed 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 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 I 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).

Unsurprisingly, the number of steals (local and remote) increases as more cores are used although at different rates as the number of remotes 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 are incur in high overhead and a very detrimental to parallel efficiency.

The obtained results 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.

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 I
WORK STEALING INFORMATION - QUEENS (N=17).

Figure 4 depicts the parallel speed-up (Figure 4a) and parallel efficiency (Figure 4b) graphs of MaCS and PaCCS.

For the case of MaCS, Figure 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 worker 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 4b - that the default settings of MaCS are not as efficient as it could. With eight cores (two nodes), the parallel efficiency drops considerably (to 91%), although less steeply after that. The overhead (Figure 3) is limiting better scalability. After improving MaCS execution based on the interpretation of previous data on overhead and load balancing, we observed almost linear speed-ups with a parallel efficiency of 96% with 512 cores. The improvement is simply based on the reduction of the number of (extraneous) release operations.

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) as other optimisation problems (not presented) 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 6c depicts the obtained performance with MaCS when compared with the ideal case. The results are near optimal, with scaling performance up to 512 cores.

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 II 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, the rate at which the number of steals increases is not constant.

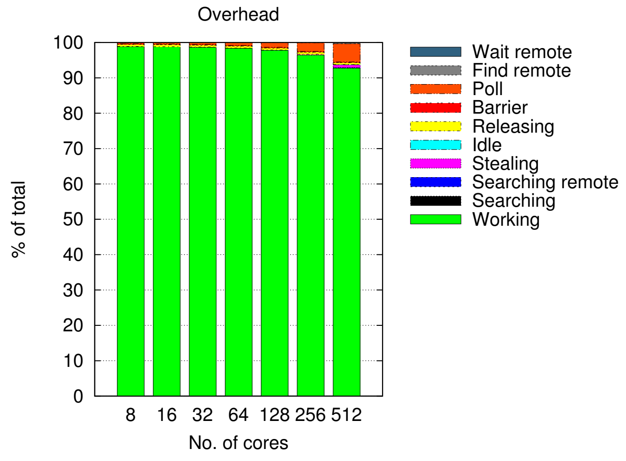


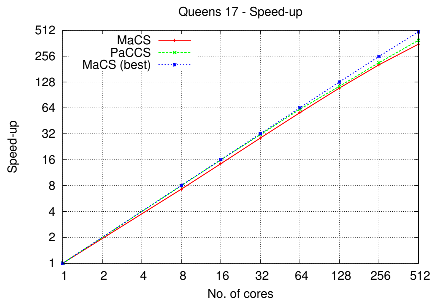
Fig. 5. Working Time and Overhead

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 512 cores. This tendency is valid for both local and remote steals.

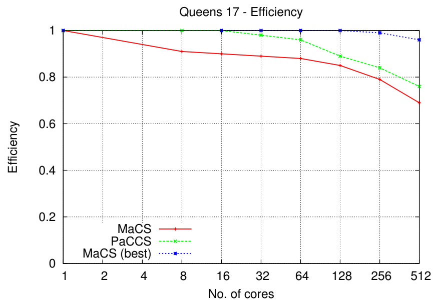
Another aspect relates to the number of failed steals: their number increases as the number of cores used is increased but remain very low - ideal up to 32 cores with zero failures - when compared with the total number of attempts to steal and when compared with the results obtained with the other problems.

The QAP is an optimisation problem and as such, the most hindering factor is the growth on the number of nodes to process. 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.

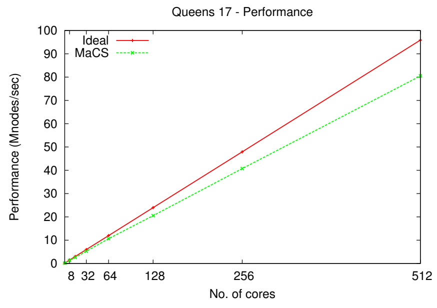
With the QAP, number of nodes (stores) processed by each



(a) Speed-up



(b) Efficiency



(c) Performance

Fig. 4. Queens (17) - Scalability

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 II
WORK STEALING INFORMATION - QAP (ESC16E).

run is not always the same, usually incrementing as we grow the number of cores. However, from Table II we can see that the growth is not substantial.

Figure 6 depicts the scalability obtained for the QAP problem for up to 512 cores. The obtained speed-ups are almost linear (Figure 6a) with a parallel efficiency above 90% (Figure 6b).

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 low increment on the total number of nodes than other optimisation problems and therefore does not suffer as much in terms of scalability. Moreover, the work stealing mechanism is more efficient, with much less failures.

A. Discussion

For compactness and space limitation, we only presented the results of two different problems as instances of satisfaction and optimisation problem. It is noteworthy to add that the behaviour observed in these two example is well transported for other problem of the same classes of problems.

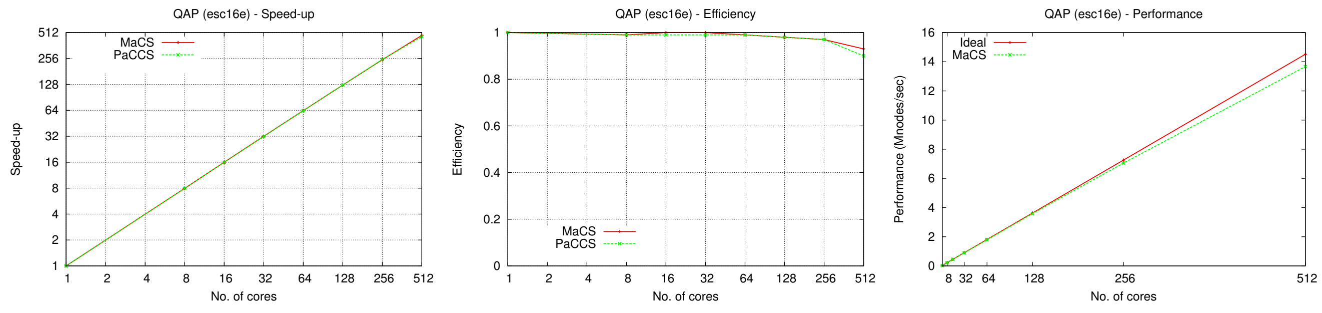
The performance evaluation from the previous sections showed that it is possible to obtain good scalability with MaCS on a large number of cores and with different kinds of problems.

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 CSP, N-Queens, exhibits a high node throughput since processing a single store is a fast operation. As a consequence, the accesses to and maintenance of the worker pool play an important role on scalability. On one hand the overhead incurred from *releasing* work on 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 problems, namely the N-Queens, 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 COP, processing a single store is more costly and hence there is a lower node throughput. Moreover, COPs suffer from an increasing problem size as more cores are added which has effects on their scalability. On the QAP, the growth on the number of nodes processed is not substantial and thus the parallel efficiency stays high (above



(a) Speed-up (b) Efficiency (c) Performance

Fig. 6. QAP - Scalability

90% up to 512 cores). Moreover, the QAP is a problem is the problem with a lower rate of stealing failures (local and remote).

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.

VII. Related Work

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([13]). Often cited work in parallel search integrated in a constraint solver is that of Perron [15], [14]. 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 [16] by parallelizing the search procedure [17]. Workers work 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. Several problems were tested resulting in good speedups but only up to 4 workers.

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 [18], 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 [19], 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 the N-Queens problem 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.

Large-scale parallel constraint solving is investigated in [20]. 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 UTS benchmark is often seen as a representative of unbalanced computations that require dynamic load balancing. An UPC implementation of the UTS benchmark is presented and evaluated in [10]. Dinan et. al [9] study 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. More recently, UTS and state space search problems in general, implemented with work stealing, remain a topic of intensive research and researchers continue to improve methods to deal with large scale computations and the hierarchical setup of current systems [22], [21].

VIII. Conclusion

Given the increasing number of processing units on current systems, with a hierarchical setup, a declarative programming approach definitely increases the productivity of the programmer, who may concentrate on the problem at hand and less on aspects related to parallelism.

In this paper we presented MaCS, a new parallel constraint solver which provides a high-level declarative approach targeted at large-scale parallelism.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 a compact and self-contained representation of a store. 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 as well as controlling the parallel execution. Although the amount of work for each worker increases, it is possible to keep the overhead low and achieve good scalability.

The approach taken and based on the observation that, from the parallelization point of view, parallel constraint solving could benefit from a general framework proved fruitful. The experimental evaluation of MaCS showed a scalable parallel constraint solver on different problems with different characteristics. 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.

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.

The encouraging results obtained with MaCS support the goal of further extending this work in many ways. One is to extend the study of MaCS' behaviour to 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). 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.

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

References

[1] Machado, R., Lojewski, C., Abreu, S., and Pfreundt, F. J. (2011). Unbalanced tree search on a manycore system using the GPI programming model. Computer Science-Research and Development, 26(3), 229-236.

[2] Machado, R., Lojewski, C.: The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. Computer Science-Research and Development 23(3), 125132 (2009)

[3] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P Sadayappan and Chau-Wen Tseng. UTS: An Unbalanced Tree Search Benchmark Proc. 19th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC). New Orleans, LA, November 2-4, 2006.

[4] Diaz, D., Abreu, S., Codognet, P.: Targeting the cell broadband engine for constraint-based local search. Concurrency and Computation: Practice and Experience **24**(6), 647–660 (2012)

[5] Martins, R., Manquinho, V., Lynce, I.: An overview of parallel SAT solving. Constraints **17**, 304–347 (2012)

[6] Michel, L., See, A., Van Hentenryck, P.: Distributed constraint-based local search. In: F. Benhamou (ed.) CP'06, 12th Int. Conf. on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, pp. 344–358. Springer Verlag (2006)

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

[8] Michel, L., See, A., Van Hentenryck, P.: Parallelizing constraint programs transparently. In: C. Bessiere (ed.) CP'07, 13th Int. Conf. on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, pp. 514–528. Springer Verlag (2007)

[9] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, P. Sadayappan Scalable Work Stealing Proc. 21st Intl. Conference on Supercomputing (SC). Portland, OR, Nov. 14-20, 2009.

[10] Stephen Olivier, Jan Prins. Scalable Dynamic Load Balancing Using UPC. Proc. of 37th International Conference on Parallel Processing (ICPP-08). Portland, OR, September 2008.

[11] Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proc. 35th Ann. Symp. Found. Comp. Sci. (1994) 356368

[12] Alba, E.: Special issue on new advances on parallel meta-heuristics for complex problems. Journal of Heuristics **10**(3), 239–380 (2004)

[13] Van Hentenryck, Pascal. Parallel constraint satisfaction in logic programming: Preliminary results of Chip within PEPSys. In Proc. 6th International Conference on Logic Programming, pp. 165-180. 1989.

[14] Perron, Laurent. "Practical parallelism in constraint programming." In Proceedings of CP-AI-OR 2002, pp. 261-276. 2002

[15] Perron, Laurent. "Search procedures and parallelism in constraint programming." In Principles and Practice of Constraint ProgrammingCP99, pp. 346-360. Springer Berlin/Heidelberg, 1999.

[16] Michel, Laurent, and Pascal Van Hentenryck. "A constraint-based architecture for local search." In ACM SIGPLAN Notices, vol. 37, no. 11, pp. 83-100. ACM, 2002.

[17] Michel, Laurent, Andrew See, and Pascal Van Hentenryck. "Transparent parallelization of constraint programming." INFORMS Journal on Computing 21, no. 3 (2009): 363-382.

[18] Jaffar, Joxan, Andrew E. Santosa, Roland HC Yap, and Kenny Q. Zhu. "Scalable distributed depth-first search with greedy work stealing." In Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on, pp. 98-103. IEEE, 2004.

[19] Bordeaux, Lucas, Youssef Hamadi, and Horst Samulowitz. "Experiments with massively parallel constraint solving." In Proceedings of the 21st international jont conference on Artifical intelligence, pp. 443-448. Morgan Kaufmann Publishers Inc., 2009.

[20] Xie, Feng, and Andrew Davenport. "Solving scheduling problems using parallel message-passing based constraint programming." In Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems COPLAS, pp. 53-58. 2009.

[21] Ravichandran, Kaushik, Sangho Lee, and Santosh Pande. "Work stealing for multi-core hpc clusters." Euro-Par 2011 Parallel Processing (2011): 205-217.

[22] Narang, Ankur, Abhinav Srivastava, Ramnik Jain, and R. Shyamasundar. "Dynamic Distributed Scheduling Algorithm for State Space Search." Euro-Par 2012 Parallel Processing (2012): 141-154.