Departamento de Informática

# DSM-PM2 Adequacy for Distributed Constraint Programming

Luís Pedro Parreira Galito Pimenta Almas
(luis.almas@deimos.com.pt)

Supervisor: Salvador Abreu (Universidade de Évora, DI)

Évora

2007

Departamento de Informática

# DSM-PM2 Adequacy for Distributed Constraint Programming

Luís Pedro Parreira Galito Pimenta Almas
(luis.almas@deimos.com.pt)

Supervisor: Salvador Abreu (Universidade de Évora, DI)

Évora

2007

# Abstract

High-speed networks and rapidly improving microprocessor performance make networks of workstations an increasingly appealing vehicle for parallel computing. No special hardware is required to use this solution as a parallel computer, and the resulting system can be easily maintained, extended and upgraded. Constraint programming is a programming paradigm where relations between variables can be stated in the form of constraints. Constraints differ from the common primitives of other programming languages in that they do not specify a step or sequence of steps to execute but rather the properties of a solution to be found. Constraint programming libraries are useful as they do not require the developers to acquire skills for a new language, providing instead declarative programming tools for use within conventional systems. Distributed Shared Memory presents itself as a tool for parallel application in which individual shared data items can be accessed directly. In systems that support Distributed Shared Memory, data moves between main memories of different nodes. The Distributed Shared Memory spares the programmer the concerns of message passing, where he would have to put allot of effort to control the distributed system behavior. We propose an architecture aimed for Distributed Constraint Programming Solving that relies on propagation and local search over a CC-NUMA distributed environment using Distributed Shared Memory.

The main objectives of this thesis can be summarized as:

- Develop a Constraint Solving System, based on the AJACS [3] system, in the C language, the native language of the experimented Parallel library - PM2 [4];

- Adapt, experiment and evaluate the developed constraint solving system distributed suitability by using DSM-PM2 [1] over a CC-NUMA architecture distributed environment;

Keywords: Distributed, Constraint Programming, Parallel, Distributed Shared Memory, PM2, AJACS

# Sumário

## Título: Adequação da biblioteca PM2-DSM para Programação por Restrições Distribuídas.

As Redes de alta velocidade e o melhoramento rápido da performance dos micro-processadores fazem das redes de computadores um veículo apelativo para computação paralela. Não é preciso hardware especial para usar computadores paralelos e o sistema resultante é extensível e facilmente alterável. A programação por restrições é um paradigma de programação em que as relações entre as variáveis pode ser representada por restrições. As restrições diferem das primitivas comuns das outras linguagens de programação porque, ao contrário destas, não especifica uma sequência de passos a executar mas antes a definição das propriedades para encontrar as soluções de um problema específico. As bibliotecas de programação por restrições são úteis visto elas não requerem que os programadores tenham que aprender novos *skills* para uma nova linguagem mas antes proporcionam ferramentas de programação declarativa para uso em sistemas convencionais. A tecnologia de Memória Partilhada Distribuída (Distributed Shared Memory) apresenta-se como uma ferramenta para uso em aplicações distribuídas em que a informação individual partilhada pode ser acedida directamente. Nos sistemas que suportam esta tecnologia os dados movem-se entre as memórias principais dos diversos nós de um cluster. Esta tecnologia poupa o programador às preocupações de passagem de mensagens onde ele teria que ter muito trabalho de controlo do comportamento do sistema distribuído. Propomos uma arquitectura orientada para a distribuição de Programação por Restrições que tenha os mecanismos da propagação e da procura local como base sobre um ambiente CC-NUMA distribuído usando memória partilhada distribuída.

Os principais objectivos desta dissertação podem ser sumarizados em:

- Desenvolver um sistema resolvedor de restrições, baseado no sistema AJACS [3], usando a linguagem 'C', linguagem nativa da biblioteca de desenvolvimento paralelo experimentada: o PM2 [4].
- Adaptar, experimentar e avaliar a adequação deste sistema resolvedor de restrições usando DSM-PM2 [1] a um ambiente distribuído assente numa arquitectura CC-NUMA;

# Acknowledgements

Thanks to my family and Fátima for all the support; to my Supervisor Salvador Abreu for all the attention and availability provided; to Deimos Engenharia for all the cooperation and understanding.

# Contents

I Introduction 9

1 Introduction and Motivation 10
1.1 Introduction . . . . . 10
1.2 Constraint Programming . . . . . 10
1.3 Distributed Shared Memory . . . . . 11
1.3.1 DSM Design . . . . . 12
1.3.2 DSM Algorithms . . . . . 13
1.3.3 Consistency Models . . . . . 15
1.3.4 Coherence Protocol Policy . . . . . 19
1.3.5 Other Design Issues . . . . . 20
1.4 Distributed Constraint Programming using DSM . . . . . 21
1.5 Objectives . . . . . 22
1.6 RoadMap to this Thesis . . . . . 23

II State of the Art 24

2 State of the Art - Distributed Shared Memory 25
2.1 The CC-NUMA Architecture . . . . . 25
2.2 DSM on Hardware / Software . . . . . 26
2.3 DSM Systems . . . . . 27
2.3.1 Page-Based DSM . . . . . 27
2.3.2 Shared Variable DSM . . . . . 28
2.3.3 Object-Based DSM . . . . . 29
2.3.4 DSM Systems Overview . . . . . 30
2.4 DSM-PM2: An Overview . . . . . 35
2.4.1 The PM2 runtime system . . . . . 35
2.4.2 DSM-PM2: The illusion of common address space . . 36
2.5 Conclusion . . . . . 37

3 State of the Art - Distributed Constraint Programming 39
3.1 Distributed Constraint Programming - from GC to Distributed AJACS . . . . . 39

3.1.1 GC Concept . . . . . . . . . . . . . . . . . . . . . . 39
3.1.2 AJACS Concept . . . . . . . . . . . . . . . . . . . . . 42
3.1.3 AJACS over Hyperion for Distributed Execution . . . 45
3.2 Other DCP Systems . . . . . . . . . . . . . . . . . . . . . . 46
3.2.1 DisChoco: A Platform for distributed constraint programming . . . . . . . . . . . . . . . . . . . . . . . . 46
3.2.2 The Mozart Programming System . . . . . . . . . . . 46
3.2.3 Disolver: The Distributed Constraint Solver . . . . . . 47
3.2.4 Alice System . . . . . . . . . . . . . . . . . . . . . . 47

**III Description** **51**

**4 AJACS/C Description - Distributed Patterns and Implementation** **52**
4.1 AJACS/C Model Description . . . . . . . . . . . . . . . . . 52
4.1.1 AJACS/C Static Model . . . . . . . . . . . . . . . . . 53
4.1.2 AJACS/C Dynamic Model . . . . . . . . . . . . . . . 55
4.1.3 AJACS/C Extensability . . . . . . . . . . . . . . . . . 57
4.2 Distributed Sharing Patterns with AJACS/C and DSM-PM2 57
4.2.1 Centralized Distribution Pattern . . . . . . . . . . . . 58
4.2.2 Local Distributed Pattern . . . . . . . . . . . . . . . . 59
4.3 AJACS/C Implementation Details - API . . . . . . . . . . . . 61
4.3.1 bittarr.h . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.3.2 constraints.h . . . . . . . . . . . . . . . . . . . . . . . 69
4.3.3 fdd_value.h . . . . . . . . . . . . . . . . . . . . . . . 72
4.3.4 problem.h . . . . . . . . . . . . . . . . . . . . . . . . 78
4.3.5 search.h . . . . . . . . . . . . . . . . . . . . . . . . . 79
4.3.6 store.h . . . . . . . . . . . . . . . . . . . . . . . . . . 81
4.3.7 strategy.h . . . . . . . . . . . . . . . . . . . . . . . . 84

**IV Interpretation of Results** **86**

**5 Examples and Interpretation of Results** **87**
5.1 PM2 Configuration . . . . . . . . . . . . . . . . . . . . . . . 88
5.2 Cluster Configuration . . . . . . . . . . . . . . . . . . . . . 88
5.3 How to create a DSM-PM2 AJACS/C Example . . . . . . . . 89
5.4 Queens Example . . . . . . . . . . . . . . . . . . . . . . . . 90
5.4.1 Specification . . . . . . . . . . . . . . . . . . . . . . . 90
5.4.2 Problem Description . . . . . . . . . . . . . . . . . . . 90
5.4.3 Interpretation of Results . . . . . . . . . . . . . . . . . 92
5.4.4 Conclusions . . . . . . . . . . . . . . . . . . . . . . . 98
5.5 n-Fractions Example . . . . . . . . . . . . . . . . . . . . . . 98

5.5.1 Specification . . . . . . . . . . . . . . . . . . . . . . . . 98
5.5.2 Problem Description . . . . . . . . . . . . . . . . . . . . 99
5.5.3 Interpretation of Results . . . . . . . . . . . . . . . . . . 101
5.5.4 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . 101

**V Conclusion** **104**

**6 Conclusions and Future Work** **105**
6.1 Work performed on this thesis . . . . . . . . . . . . . . . . . . . 105
6.1.1 AJACS/C Development . . . . . . . . . . . . . . . . . . 105
6.1.2 DSM-PM2 AJACS/C Integration . . . . . . . . . . . . 105
6.1.3 Distribution Models Design . . . . . . . . . . . . . . . . 106
6.1.4 AJACS/C with DSM-PM2 Experimentation . . . . . . 106
6.2 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
6.3 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

# List of Figures

1.1 DSM Concept . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
1.2 DCP using DSM Concept . . . . . . . . . . . . . . . . . . . . 22

2.1 Object Based DSM . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.2 Overview of the DSM-PM2 Software Architecture . . . . . . . 38

3.1 All Classes of AJACS . . . . . . . . . . . . . . . . . . . . . . . 44
3.2 DFS search applied to 4-Queens problem . . . . . . . . . . . . 45
3.3 Disolver General Architecture . . . . . . . . . . . . . . . . . . 48

4.1 AJACS/C Static Model . . . . . . . . . . . . . . . . . . . . . 54
4.2 AJACS/C Dynamic Model . . . . . . . . . . . . . . . . . . . . 56
4.3 DSM-PM2 AJACS/C Architecture . . . . . . . . . . . . . . . 57
4.4 Centralized DSM structure Architecture . . . . . . . . . . . . 58
4.5 Centralized Distribution Pattern - Master Algorithm . . . . . 59
4.6 Centralized Distribution Pattern - Worker Algorithm . . . . . 59
4.7 Local DSM structure Architecture . . . . . . . . . . . . . . . 60
4.8 Local Distribution Pattern - Master Algorithm . . . . . . . . 60
4.9 Local Distribution Pattern - Worker Algorithm . . . . . . . . 61

5.1 Experimenting Cluster Configuration . . . . . . . . . . . . . . 89
5.2 Building a DSM-PM2 AJACS/C application . . . . . . . . . . 90
5.3 Queens example . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.4 Queens Centralized strategy time results . . . . . . . . . . . . 93
5.5 Queens Centralized strategy speedup results . . . . . . . . . . 94
5.6 Queens Local strategy time results . . . . . . . . . . . . . . . 96
5.7 Queens Local strategy speedup results . . . . . . . . . . . . . 97
5.8 n-Fraction (n=2) example . . . . . . . . . . . . . . . . . . . . 100
5.9 n-Fractions (n=2) Local strategy time results . . . . . . . . . 102
5.10 n-Fractions (n=2) Local strategy speedup results . . . . . . . 103

# Part I

# Introduction: Constraint Programming and Distributed Shared Memory

# Chapter 1

# Introduction and Motivation

On this chapter the main context, basis and goals of this thesis study are presented to the reader.

## 1.1 Introduction

High-speed networks and rapidly improving microprocessor performance make networks of workstations an increasingly appealing vehicle for parallel computing. No special hardware is required to use this solution as a parallel computer, and the resulting system can be easily maintained, extended and upgraded. In terms of performance, improvements in processor speed and network bandwidth and latency allow networked workstations to deliver performance approaching or exceeding supercomputer performance for an increasing class of applications.

## 1.2 Constraint Programming

Constraint programming is a programming paradigm where relations between variables can be stated in the form of constraints. Constraints differ from the common primitives of other programming languages in that they do not specify a step or sequence of steps to execute but rather the properties of a solution to be found. The constraints used in constraint programming are of various kinds: those used in constraint satisfaction problems, those solved by the simplex algorithm, and others. Constraints are usually embedded within a programming language or provided via separate software libraries.

Constraint programming began with constraint logic programming, which embeds constraints into a logic program. This variant of logic programming is due to Jaffar and Lassez, who extended in 1987 a specific class of constraints that were introduced in Prolog II [31]. The first implementations of constraint logic programming were Prolog III [32], CLP(R) [33], and

CHIP [34]. Several constraint logic programming interpreters exist today, for example GNU Prolog [35].

Other than logic programming, constraints can be mixed with functional programming, term rewriting, and imperative languages. Constraints in functional programming are implemented in the multi-paradigm programming language Oz [5]. Constraints are embedded in an imperative language in Kaleidoscope. However, constraints are implemented in imperative languages mostly via constraint solving toolkits, which are separate libraries for an existing imperative language. ILOG Solver [36] is an example of such a constraint programming library for C++.

Constraint programming libraries are useful when building applications developed mostly in mainstream programming languages: they do not require the developers to acquire skills for a new language, providing instead declarative programming tools for use within conventional systems.

## 1.3 Distributed Shared Memory

Distributed Shared Memory presents itself as a tool for parallel application or a group of applications in which individual shared data items can be accessed directly. In systems that support Distributed Shared Memory, data moves between main memories of different nodes (as illustrated by figure 1.1). Each node can own data stored in the shared address space, and the ownership can change when data moves from one node to another. When a process accesses data in the shared address space, a mapping manager maps the shared memory address to the physical memory. The mapping manager is a layer of software implemented either in the operating system kernel or as a runtime library routine. Note: Most of the content discussed in this section stems from a survey of publicly available materials, which are duly identified.

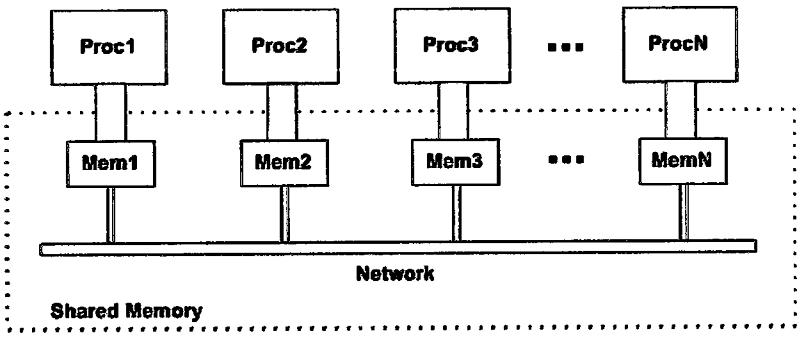


Figure 1.1: DSM Concept

The Distributed Shared Memory spares the programmer the concerns of message passing, where he would have to put allot of effort to control the distributed system behavior. The goal is to provide him the abstraction of shared memory. Shared memory provides the fastest possible communication, hence the greatest opportunity for concurrent execution.

### 1.3.1 DSM Design

The main problems that the DSM approach has to address are:

- The mapping of a logically shared address space onto the physically distributed memory modules;
- how to locate and access the needed data item(s);
- how to perserve the coherent view of the overall shared address space.

To responde to these problems we refer to the following text, that sintethizes the main considerations when designing DSM systems.

> [1] The crucial objective of solving those problems is the minimization of the average access time to the shared data. Having this goal in mind, two strategies for distribution of shared data are most frequently applied: replication and migration. Replication allows that multiple copies of the same data item reside in different local memories, in order to increase the parallelism in accessing logically shared data. Migration implies a single copy of data item which has to be moved to the accessing site, counting on the locality of reference in parallel applications. Besides that, systems with distributed shared memory have to deal with the consistency problem, when replicated copies of the same data exist. In order to preserve the coherent view of shared address space, according to the strict consistency semantics, a read operation must return the most recently written value. Therefore, when one of multiple copies of data is written, the others become stale, and have to be invalidated or updated, depending on the applied coherence policy. Although the strict coherence semantics provides the most natural view of shared address space, various weaker forms of memory consistency can be applied in order to reduce latency.
>
> As a consequence of applied strategies and distribution of shared address space across different memories, on some memory reference data item and its copies have to be located and managed according to a mechanism which is appropriate for such architecture. The solutions to the problems above are incorporated into the DSM algorithm, which can be implemented at the hardware and/or software level. Implementation level of a DSM mechanism is regarded as the basic design decision, since it profoundly affects system performance. The other important issues include: structure and granularity of shared data, memory consistency model that determines allowable memory access orderings and coherence policy (invalidate or update).

[1]From http://galeb.etf.bg.ac.yu/ vm/tutorial/multi/dsm/introduction/introduction.html

### 1.3.2 DSM Algorithms

The performance of a DSM system is dependent on the coupling between the applied DSM algorithm and the access patterns generated by the application. One of the possible classifications of DSM algorithms distinguishes between

> [2] (...)**SRSW** (Single Reader / Single Writer), **MRSW** (Multiple Reader / Single Writer), and **MRMW** (Multiple Reader / Multiple Writer). SRSW algorithms prohibit replication, and migration can be allowed if the distribution of shared address space over distributed memories is not static. MRSW algorithms are most often encountered in DSM systems. Based on a realistic assumption that the read operation is generally more frequent than the write operation, they allow multiple read-only copies of data items, in order to decrease read latency. Finally, the MRMW algorithms allow the existence of multiple copies of data items, even while being written to, but in some case, depending on the consistency semantics, write operations must be globally sequenced.

The algorithms to be described below [3] are categorized by whether they migrate and / or replicate data. Two of the algorithms (central-server algorithm and migration algorithm) migrate data to the site where it is accessed in an attempt to exploit local data accesses decreasing the number of remote accesses, thus avoiding communication overhead. The two other algorithms (Read-Replication algorithm and Full-Replication algorithm) replicate data so that multiple read accesses can take place at the same time using local accesses.

#### Central-server Algorithm

In the Central-Server Algorithm, a central-server maintains all the shared data. It services the read requests from other nodes or clients by returning the data items to them. It updates the data on write requests by clients and returns acknowledgment messages. A timeout can be employed to resend the requests in case of failed acknowledgments. Duplicate write requests can be detected by associating sequence numbers with write requests. A failure condition is returned to the application trying to access shared data after several retransmissions without a response.

Although, the central-server algorithm is simple to implement, the central-server can become a bottleneck. To overcome this problem, shared data can be distributed among several servers. In such a case, clients must be able to locate the appropriate server for every data access. Multicasting data access requests is undesirable as it does not reduce the load at the servers compared to the central-server scheme. A better way to distribute data is to partition the shared data by address and use a mapping function to locate the appropriate server.

[2]From http://galeb.etf.bg.ac.yu/ vm/tutorial/multi/dsm/introduction/introduction.html
[3]From http://www.niksula.cs.hut.fi/projects/ohtdsm/documents/kirjtutk/cmodels.html

## Migration Algorithm

In the Migration Algorithm, the data is shipped to the location of the data access request allowing subsequent accesses to the data to be performed locally. The migration algorithm allows only one node to access a shared data at a time. This is a single reader / single writer protocol, since only the threads executing on one host can read or write a given data item at any time.

Typically, the whole page or block containing the data item migrates instead of an individual item requested. This algorithm takes advantage of the locality of reference exhibited by programs by amortizing the cost of migration over multiple accesses to the migrated data. However, this approach is susceptible to thrashing, where pages frequently migrate between nodes while servicing only a few requests.

The migration algorithm provides an opportunity to integrate DSM with the virtual memory provided by the operating system running at individual nodes. When the page size used by DSM is a multiple of the virtual memory page size, a locally held shared memory page can be mapped to an application's virtual address space and accessed using normal machine instructions. On a memory access fault, if the memory address maps to a remote page, a fault-handler will migrate the page before mapping it to the process's address space. Upon migrating, the page is removed from all the address spaces it was mapped to at the previous node. Note that several processes can share a page at a node. To locate a data block, the migration algorithm can make use of a server that keeps track of the location of pages, or through hints maintained at nodes. These hints direct the search for a page toward the node currently holding the page. Alternatively, a query can be broadcasted to locate a page.

## Read-Replication Algorithm

One disadvantage of the migration algorithm is that only the threads on one host can access data contained in the same block at any given time. Replication can reduce the average cost of read operations, since it allows read operations to be simultaneously executed locally (with no communication overhead) at multiple hosts. However, some of the write operations may become more expensive, since the replicas may have to be invalidated or updated to maintain consistency. Nevertheless, if the ratio of reads over writes is large, the extra expense for the write operations may be more than offset by the lower average cost of the read operations.

Replication can be naturally added to the migration algorithm by allowing either one site a read/write copy of a particular block or multiple sites read-only copies of that block. This type of replication is referred to as multiple readers / single writer replication.

For a read operation on a data item in a block that is currently not local, it is necessary to communicate with remote sites to first acquire a read-only copy of that block and to change to read only the access rights to any writable copy if necessary before the read operation can complete. For a write operation to data in a block that is either not local or for which the local host has no write permission, all copies of the same block held at other sites must be invalidated before the write can proceed. The read-replication algorithm is consistent because a read access always returns the value of the most recent write to the same location.

In the read replication algorithm, DSM must keep track of the location of all the copies of data blocks. One way to do this is to have the owner node of a data block keep track of all the nodes that have a copy of the data block. Alternatively, a distributed linked list may be used to keep track of all the nodes that have a copy of the data block.

#### Full-Replication algorithm

The full replication algorithm is an extension of the read replication algorithm. It allows multiple nodes to have both read and write access to shared data blocks (the multiple readers / multiple writer protocol). Because many nodes can write shared data concurrently, the access to shared data must be controlled to maintain its consistency.

One possible way to keep the replicated data consistent is to globally sequence the write operations. A simple strategy based on sequencing uses a single global sequencer which is a process executing on a host participating in DSM. When a process attempts a write to shared memory, the intended modification is sent to the sequencer. This sequencer assigns the next sequence number to the modification with this sequence number to all sites. Each site processes broadcast write operations in sequence number order. When a modification arrives at a site, the sequence number is verified as the next expected one. If a gap in the sequence numbers is detected, either a modification was missed or a modification was received out of order, in which case a retransmission of the modification message is requested. In effect, this strategy implements a negative acknowledgment protocol.

### 1.3.3 Consistency Models

DSM systems, as we have seen, rely on replicating shared data items for allowing concurrent access at many nodes in order to improve performance. However, the concurrent accesses need to be carefully controlled, so that memory accesses may be executed in the order that which the programmer expects. In other words, the memory is coherent if the value returned by a read operation is always the value that the programmer would expect, that a read operation returns the value stored by the most recent write operation. Thus, to maintain the coherence of shared data, a mechanism to control and synchronize the accesses is necessary.

The term consistency is used to refer to a specific kind of coherence. The most natural semantics for memory coherence is strict consistency, defined as:

[4] "Strict consistency requires the ability to determine the latest write, which in turn implies a total ordering of requests. The total ordering of requests leads to inefficiency due to more data movement and synchronization requirements than what a program may really call for."

To counter this problem, DSM systems attempt to improve the performance by providing relaxed coherence semantics. Next follows several forms of memory coherence [4] (strict consistency, causal consistency, PRAM consistency & processor consistency, weak consistency, release consistency, entry consistency and scope consistency).

### Strict Consistency

The most stringent consistency model is called strict consistency. It is defined by the following condition: Any read to a memory location X returns the value stored by the most recent write operation to X.

This definition implicitly assumes the existence of absolute global time so that the determination of "most recent" is unambiguous. Uniprocessors have traditionally observed strict consistency.

In summary, when memory is strictly consistent, all writes are instantaneously visible to all processes and an absolute global time order is maintained. If a memory location is changed, all subsequent reads from that location see the new value, no matter how soon after the change the reads are done and no matter which processes are doing the reading and where they are located. Similarly, if a read is done, it gets the current value, no matter how quickly the next write is done.

### Causal Consistency

The causal consistency model represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not. Suppose that process **P1** writes a variable **X**. Then **P2** reads **X** and writes **Y**. Here the reading of **X** and the writing of **Y** are potentially causally related because the computation of **Y** may have depended on the value of **X** read by **P2** ( i.e., the value written by **P1** ). For a memory to be considered causally consistent, it is necessary that the memory obey the following condition: "Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines".

### PRAM Consistency & Processor Consistency

In causal consistency, it is permitted that concurrent writes be seen in a different order on different machines, although causally related ones must be seen in the same order by all machines. The next step in relaxing memory is to drop the latter requirement. Doing so gives

[4]From http://www.niksula.cs.hut.fi/projects/ohtdsm/documents/kirjtutk/cmodels.html

PRAM consistency, which is subject to the condition of all writes being done done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

### Weak Consistency

Although PRAM consistency and processor consistency can give better performance than the stronger models, they are still unnecessarily restrictive for many applications because they require that writes originating in a single process be seen everywhere in order. Not all applications require even seeing all writes, let alone seeing them in order. Considering the case of a process inside a critical section reading and writing some variables in a tight loop. Even though other processes are not supposed to touch the variables until the first process has left its critical section, the memory has no way of knowing when a process is in a critical section and when it is not, so it has to propagate all writes to all memories in the usual way.

Weak consistency, has three properties:

1. Accesses to critical (synchronization) variables are sequentially consistent.
2. No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
3. No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.

### Release Consistency

Weak consistency has the problem that when a synchronization variable is accessed, the memory does not know whether this is being done because the process is finished writing the shared variables or about to start reading them. Consequently, it must take the actions required in both cases, namely making sure that all locally initiated writes have been completed, as well as gathering in all writes from other machines. If the memory could tell the difference between entering a critical region and leaving one, a more efficient implementation might be possible. To provide this information, two kinds of synchronization variables or operations are needed instead of one.

Release consistency provides these two kinds. **Acquire** accesses are used to tell the memory system that a critical region is about to be entered. **Release** accesses say that a critical region has just been exited. These accesses can be implemented either as ordinary operations on special variables or as special operations.

**Lazy Release Consistency**: LRC is a refinement of RC that allows consistency action to be postponed until a synchronization variable released in a subsequent operation is acquired by another processor. Even then, the shared writes are made visible only to the acquiring processor. Synchronization transfers in an LRC system, therefore, involve only the synchronizing processors. This reduction in synchronization traffic can result in a significant decrease in the total amount of system communication, and a consequent increase in overall performance.

### Entry Consistency

Another consistency model that has been designed to be used with critical sections is entry consistency. Like release consistency, it requires the programmer to use acquire and release at the start and end of each critical section, respectively. However, unlike release consistency, entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier. If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks. When an acquire is done on a synchronization variable, only those ordinary shared variables guarded by that synchronization variable are made consistent. Entry consistency differs from lazy release consistency in that the latter does not associate shared variables with locks or barriers and at acquire time has to determine empirically which variables it needs.
Formally, a memory exhibits entry consistency if it meets all the following conditions:

1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
3. After an exclusive mode access to a synchronization variable has been performed, any other process next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

### Scope Consistency [5]

Scope Consistency was developed by the University of Princeton in 1996. In ScC, the concept of **scope** is defined as all the critical sections. This means the locks define the scopes implicitly, making the concept easy to understand. A scope is said to be **opened at an acquire**, and **closed at a release**. ScC is defined as: *When a processor Q opens a scope previously closed by another processor P, P propagates the updates made within the same scope to Q.* As ScC propagates less

[5]From http://www.srg.csis.hku.hk/srg/html/jump.htm

amount of data, the efficiency increases. And the updates not propagated are usually not needed, since it is usual practice for all accesses of the same shared variable to be guarded by the same lock.

### Summary

- Strict: Absolute time ordering of all shared accesses matters.
- Causal: All processes see all causally-related shared accesses in the same order.
- PRAM: All processes see writes from each processor in the order they were issued. Writes from different processors may not always be in the same order.
- Weak: Shared data can only be counted on to be consistent after synchronization is done.
- Release: Shared data are made consistent when a critical region is exited.
- Entry: Entry consistency improves lazy release consistency in that the latter does not associate shared variables with locks or barriers and at acquire time has to determine empirically which variables it needs.
- Scope: Locks define the scopes implicitly. It propagates less amount of data / the efficiency increases.

## 1.3.4 Coherence Protocol Policy

After a write operation, the subjacent replicated piece of data becomes out-of-date and must be refreshed across all replicated sites. Two strategies can be used to achieve this: write invalidate or write update.

#### Write Invalidate [6]

This protocol is commonly implemented in the form of *multiple-reader-single-writer sharing.* At any time, a data item may either be:

- accessed in read-only mode by one or more processes;
- read and written by a single process.

An item that is currently accessed in read-only mode can be copied indefinitely to other processes. When a process attempts to write to it, a multicast message is sent to all other copies to invalidate them, and this is acknowledged before the write can take place; the other processes are thereby prevented from reading stale data. Any processes attempting to access the data item are blocked if a writer exists.

[6]From http://www.cs.gmu.edu/cne/modules/dsm/purple/wr$_i$*nval.html*

Eventually, control is transferred from the writing process and other accesses may take place once the update has been sent. The effect is to process all accesses to the item on a first-come-first-served basis. This scheme achieves sequential consistency.

#### Write Update [7]

In the write update protocol, the updates made by a process are made locally and multicast to all other replica managers possessing a copy of the data item, which immediately modify the data read by local processes. Processes read the local copies of data items, without the need for communication. In addition to allowing multiple readers, several processes may write the same data item at the same time; this is known as **multiple-reader-multiple-writer sharing**.

Reads are cheap in the write-update option. However, ordered multicast protocols are relatively expensive to implement in software.

#### Eager VS Lazy protocols

In an eager protocol, modifications to shared data are made visible globally at the time of a release. With a lazy protocol, the propagation of modifications is postponed until the time of the acquire. At this time, the acquiring processor determines which modifications it needs to see. Both eager and lazy approaches can be applied to invalidate or update protocols. Lazy protocols are ideally suited for situations where communication has a high cost per message, because they send messages only when absolutely necessary, often resulting in far fewer messages overall.

### 1.3.5 Other Design Issues

Besides the important decisions on choosing the DSM algorithm (that can be implemented on HW and/or SW), the best suited memory consistency model that determines allowable memory access orderings or coherence policy (invalidate or update), other important issues like: **structure** and **granularity** of shared data or page replacement are also crucial [8].

#### Granularity

Granularity refers to the **size of the shared memory unit**. A page size of that is a multiple of the size provided by the underlying hardware or the memory management system allows for the integration of DSM and the memory management systems. By integrating DSM with the underlying memory management system, a DSM system can take advantage of the built in protection mechanism to detect incoherent memory references, and use built in fault handlers to prevent and recover from inappropriate references.

[7]From http://www.niksula.cs.hut.fi/projects/ohtdsm/documents/kirjtutk/cmodels.html
[8]From http://www.cs.gmu.edu/cne/modules/dsm/green/design.html

A large page size for the shared memory unit will take advantage of the locality of reference exhibited by processes. By transferring large pages, less overhead is incurred due to page size, but there is greater chance for contention to access a page by many processes. Smaller page sizes are less apt to cause contention as they reduce the likelihood of false sharing.

**False sharing** of a page occurs when two different data items, not shared but accessed by two different processes, are allocated to a single page. So the protocols that adapt to a granularity size that is appropriate to the sharing pattern will perform better than those protocols that make use of a static granularity size. False sharing results when the system can not distinguish between accesses to logically distinct pieces of data. False sharing occurs because the system tracks accesses at a granularity larger than the size of individual shared data items. Conventional protocols typically require processes to gain sole access to a page before it can be modified. Therefore, false sharing can lead to situations where multiple processes contest ownership of a page, even though the processes are modifying entirely disjoint sets of data.

### Page Replacement

A memory management system has to address the issue of page replacement because the size of physical memory is limited. In DSM systems that support data movement, traditional methods such as least recently used (LRU) cannot be used directly. Data may be accessed in different modes such as **shared, private, read-only, writable, etc.**, in DSM systems. To avoid degradation in the system performance, a page replacement policy would have to take the **page access modes** into consideration. For instance, private pages may be replaced before shared pages, as shared pages would have to be moved over the network, possibly to their owner. Read-only pages can simply be deleted as their owners will have a copy. Thus the LRU policy with classes is one possible strategy to handle page replacement. Once a page is selected for replacement, the DSM system must ensure that the page is not lost forever. One option is to swap the page onto disk memory. However, if the page is a replica and is not owned by the node, it can be sent to the owner node.

## 1.4 Distributed Constraint Programming using DSM

On of the inherent characteristics of Constraint Programming aims on solving problems that may have huge search spaces and as so often are very time consuming tasks. The natural choice is to try to paralelize the search space over different CPUs (as illustrated in figure 1.2) in using some paralelization technique, being one specific example the distribution of work over different independent nodes over some network.

Paralelization does not come without cost as applications have to be designed specifically for that purpose which means that some decisions have to be made before-hand such as:

- The feasability to partition the work to be distributed over different nodes;
- The needed synchronization, from the problem itself, shall be reduced to a minimum to not generate too much communication between the distributed nodes;
- Some paralelization model needs to be designed, supported by the use of some specific paralelization libraries (MPI [37] alike or other) to assure a proper distributed computation.

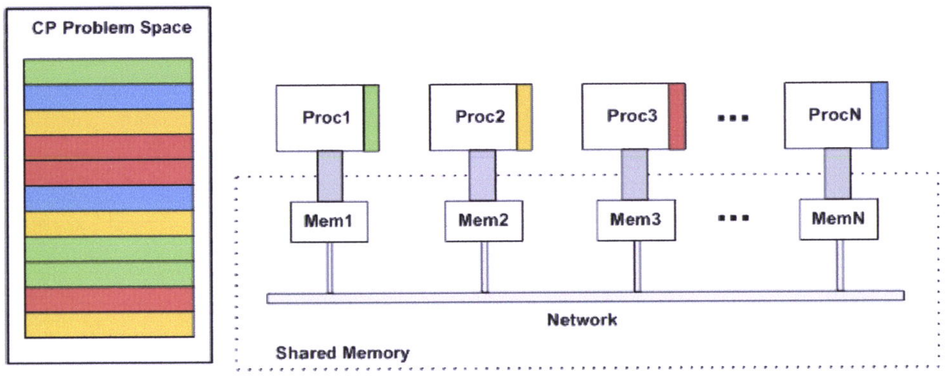


Figure 1.2: DCP using DSM Concept

As information must transit between the different nodes DSM comes as a natural hypothesis choice to ease the burden caused by the last identified need, from the items listed above.

The communication abstraction, that DSM sells to offer, is most apealing since theoretically it would eliminate the need for the programmer to waist energy on message passing management and data consistency across memory allocated on each node and allow him to concentrate / focus attention on the distribution model design and on the CSP distributed feasability itself.

## 1.5 Objectives

This thesis presents an adaptation of the AJACS [3] system to the 'C' language and an adequancy study of this constraint solving system to a distributed environment using distributed shared memory, hereby its proposed to:

- Develop a Constraint Solving System in the 'C' language, the native language of the experimented Parallel library - PM2 [4] (to be detailed further ahead on this thesis report);
- Adapt and Experiment the 'C' Constraint Solving System to a CC-NUMA architecture (refer to section 2.1 for details) distributed environment using DSM-PM2 [1];
- Evaluate the DSM-PM2 adequacy for distributed constraint programming by testing different distributed approaches and at the same time trying to obtain run-time performance speedups.

The work performed in this thesis will be presented in 7th International Colloquium on Implementation of Constraint and Logic Programming Systems [48].

## 1.6 RoadMap to this Thesis

The reminder chapters of this thesis report have the following presentation:

- On Chapters 2 and 3 is presented a *State of the Art* summary on the two involved technologies under subject analysis of this thesis: - Distributed Shared Memory (chapter 2) and Distributed Constraint Programming (chapter 3);
- On Chapter 4 the AJACS/C constraint library is described, including it's implementation details. The adopted Distribution Models proposed for experimentation are also explained;
- Chapter 5 deals with examples and interpretation of results;
- Chapter 6 presents conclusions and future lines of work.

# Part II

# State of the Art: Distributed Shared Memory and Distributed Constraint Programming

# Chapter 2

# State of the Art - Distributed Shared Memory

This chapter will present the reader a survey on the currently available DSM technology.

## 2.1 The CC-NUMA Architecture

The cluster configuration used by this thesis report study is based on a CC-NUMA model. Next follows a definition of the CC-NUMA architecture as for a comparison with other model architectures [1].

> The term **CC-NUMA** stands for Cache-Coherent Non Uniform Memory Access. In the CC-NUMA model, the system runs one operating system and shows only a single memory image to the user eventhough the memory is physically distributed over the processors. Since processors can access their own memory much faster than that of other processors, memory access is non uniform (NUMA). In this architecture the contents of the various processor caches should be coherent requiring extra hardware and a cache coherency protocol. A NUMA computer fulfilling these requirements is called a CC-NUMA machine. Refer to figure 1.1.
>
> Comparing a CC-NUMA computer to MPP (Massively Parallel Processing) / cluster-based systems and SMP (Symmetric Multi Processing)/PVP (Parallel Vector Processing) systems shows that CC-NUMA machines can be regarded as an attempt to get the best out of both worlds.
>
> SMP/PVP computers have multiple processors using the same memory which makes it easy to parallelize a code on loop level using compiler options and/or directives. However, due to the fact that processors have to perform data exchange with their memory over the same

[1] From http://www.sara.nl/userinfo/reservoir/ccnuma/index.html

bus having only a limited bandwidth, these systems will only scale to 10s of processors.

MPP/cluster-based systems do not have this drawback. The nodes in MPP/cluster-based machines have their own private memory and therefore each node possesses only a part of the data. Such machines will scale upto a very large number of processors if the computation to communication ratio of the program is high. The programming model for these computers is based on message passing and processors explicitly have to perform the communication with other processors by sending and receiving data. Programs have to take care of data distribution over the processors and have to be adapted for explicit communication. This implies that programs developed for single processor and PVP/SMP machines cannot be applied to these machines straight away. A substantial amount of redesigning and reprogramming of these codes is necessary.

CC-NUMA machines combine the benefits of MPP/cluster-based systems and SMP/PVP machines. The fact that CC-NUMA machines behave like shared memory computers from a user point of view simplifies the porting of programs developed for single processor or SMP/PVP machines. Moreover, CC-NUMA computers allow for loop-level parallelization by means of compiler options or compiler directives similar to SMP/PVP systems. The good scalability properties are inheritted from MPP/cluster-based systems since memory is distributed over the nodes.

## 2.2 DSM on Hardware / Software

DSM solutions come in two main flavors: Software and Hardware based. Hybrid solutions propose to be the third choice. These solutions are described below [2].

**Software** support for DSM is generally more flexible and convenient for experiments than hardware implementations, but in many cases can not compete with the hardware level DSM in performance. Nevertheless, majority of DSM systems described in the open literature were based on software mechanisms, since networks of workstations are getting more popular and powerful. Therefore, the use of DSM concept seems to be an appropriate and relatively low-cost solution for their use as parallel computers. Ideas and concepts that originally appeared in software-oriented systems often migrated to the hardware implementations. Software-based DSM mechanisms can be implemented on the level of programming language, since the compiler can detect shared accesses and insert calls to synchronization and coherence routines into executable code.

**Hardware** approach has two very important advantages: complete transparency to the programmer, and generally better performance

[2]From http://galeb.etf.bg.ac.yu/ vm/tutorial/multi/dsm/introduction/introduction.html

than other approaches. Since hardware implementations typically use a smaller unit of sharing (e.g., cache block), they are less susceptible to false sharing and thrashing effects. Hardware implementations are particularly superior for applications that have high level of fine-grain sharing. Hardware solutions are classified into three groups: **CC-NUMA** (Cache Coherent Non-Uniform Memory Architecture), **COMA** (Cache-Only Memory Architecture), and **RMS** (Reflective Memory System) architectures. In CC-NUMA systems, as explained in section 2.1, parts of shared address space are statically distributed among the local memories in the clusters of processors, where the improved locality of accesses is expected. In the COMA architectures, the distribution of data is dynamically adaptable to the application behavior, so the parts of overall workspace can freely migrate according to its usage. In reflective memory architectures, all write operations to the globally shared regions are immediately followed with updates of all other copies of the same data item. Hardware-oriented DSM mechanism appears to be very promising DSM approach, due to its superior performance and the transparency it offers to the programmer. It is expected to be more frequently used in the future.

**Hybrid solutions** may be in order to achieve the speed and transparency of hardware schemes, as well as the flexibility and sophistication of software solutions. Designers sometimes choose to implement a suitable combination of hardware and software methods. Some level of software support can be found even in the entirely hardware solutions, with a goal to better suit to the application behavior. As none of the design choices in the world of DSM has been proven to be absolutely superior, it seems that the integration of various approaches will be intensively pursued in future by system architects, in their strive to gain better performance.

## 2.3 DSM Systems

In this section several DSM systems are presented. Different types of systems are introduced and several of their main features are exposed and compared.

### 2.3.1 Page-Based DSM

The idea behind this type of DSM system is to emulate the cache of a multiprocessor by making use of the *memory management unit (MMU)* and the operating system software. In this DSM system, the address space is divided into **chunks**. These chunks are distributed over all the processors in the system. When a processor references an address that is not local (available in that node), a trap occurs, and the DSM software fetches (gets) the chunk containing the address and restarts the faulting instruction, which now completes successfully. Further characteristics of this type of DSM system are presented below. [3]

#### Replication

One improvement to the basic system that can improve performance considerably is to replicate chunks that are read only, read-only constants, or other read-only data structures. However, if a replicated chunk is suddenly modified, inconsistent copies are in existence. The inconsistency is prevented by using some consistency protocols.

#### Finding the Owner

The simplest solution for finding the owner is by doing a broadcast, asking for the owner of the specified page to respond. An optimization is not just to ask who the owner is, but also to tell whether the sender wants to read or write and say whether it needs a copy of the page. The owner can then send a single message transferring ownership and the page is well, if needed.

#### Finding the Copies

Another important detail is how all the copies are found when they must be invalidated. Again, two possibilities present themselves. The first is to broadcast a message giving the page number and ask all processors holding the page to invalidate it. This approach works only if broadcast messages are totally reliable and can never be lost.

The second possibility is to have the owner or page manager maintain a list or copyset telling which processors hold which pages. When a page must be invalidated, the old owner, new owner, or page manager sends a message to each processor holding the page and waits for an acknowledgment. When each message has been acknowledged, the invalidation is complete.

#### Page Replacement

As in any system using virtual memory, it can happen that a page is needed but that there is no free page frame in memory to hold it. When this situation occurs, a page must be evicted from memory to make room for the needed page. Two subproblems immediately arise: which page to evict and where to put it.

### 2.3.2 Shared Variable DSM

Page-based DSM takes a normal linear address space and allows pages to migrate dynamically over the network. Another, more structured approach, is to share only a determined set of variables and data structures that are needed by more than one process. This way, the problem changes from

[3]From http://cs.gmu.edu/cne/modules/dsm/yellow/page_dsm.html

how to do paging over the network to how to maintain a potentially replicated, distributed data base consisting of the set of shared variables. More considerations and an example below [4].

> Different techniques are applicable here, and these often lead to major performance improvements. Using shared variables that are individually managed also provides considerable opportunity to eliminate false sharing. If it is possible to update one variable without affecting other variables, then the physical layout of the variables on the pages is less important. One of the most important examples of such a system is *Munin* [9].

### 2.3.3 Object-Based DSM

In an **object-based distributed shared memory**, processes on multiple machines (as figure 2.1 illustrates) share an abstract space filled with shared objects. The location and management of the objects is handled automatically by the system. This model is more abstract and differs to the *page-based DSM systems*, in which the former provides, in contrast, a raw linear memory of bytes from 0 to some maximum. In object-based DSM any process can invoke any object's methods, regardless of where the process and object are located. The operating system and runtime system have the task to make the act of invoking work no matter where the processes and the objects are located. An Object-Based DSM feature is that processes cannot directly access the internal state of any of the shared objects, so various optimizations are claimed as possible where not possible with page-based DSM. For example, since access to the internal state is strictly controlled, it may be possible to relax the memory consistency protocols. Further detail on Object-Based DSM characteristics as for its advantages and disadvantages are described below [5].

> Once a decision has been made to structure a shared memory as a collection of separate objects instead of as a linear address space, there are many other choices to be made. Probably the most important issue is whether objects should be replicated or not. If replication is not used, all accesses to an object go through the one and only copy, which is simple, but may lead to poor performance. By allowing objects to migrate from machine to machine, as needed, it may be possible to reduce the performance loss by moving objects to where they are needed.
>
> On the other hand, if objects are replicated, what should be done when one copy is updated? One approach is to invalidate all the other copies so that only the up to date copy remains. Additional copies can be created later, on demand, as needed. An alternative choice is not to invalidate the copies, but to update them. Shared-variable DSM also

[4]From http://cs.gmu.edu/cne/modules/dsm/yellow/shared_dsm.html
[5]From http://cs.gmu.edu/cne/modules/dsm/yellow/object_dsm.html

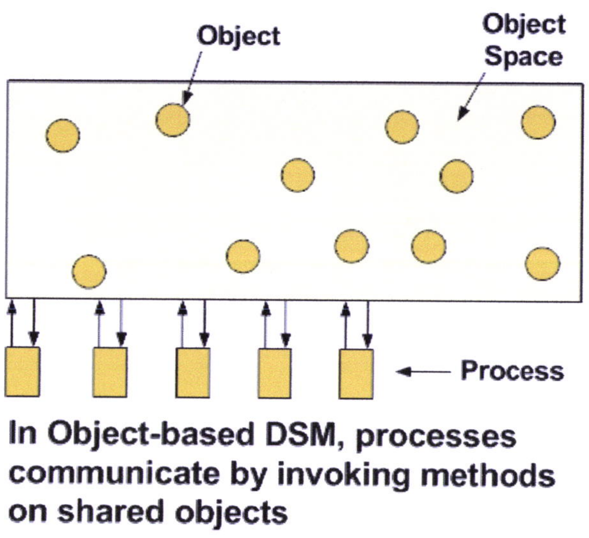

Object
Object Space
Process
In Object-based DSM, processes communicate by invoking methods on shared objects

Figure 2.1: Object Based DSM

has this choice, but for page-based DSM, invalidation is the only feasible choice. Similarly, object-based DSM, like shared-variable DSM, eliminates most false sharing.

Object-based DSM has **three advantages** over the other methods:

1. It is more modular than the other techniques.
2. The implementation is more flexible because accesses are controlled.
3. Synchronization and access can be integrated together cleanly.

Object-based DSM **also has disadvantages**. For one thing, it cannot be used to run old "dusty deck" multiprocessor programs that assume the existence of a shared linear address space that every process can read and write at random.

A second potential disadvantage is that since all accesses to shared objects must be done by invoking the objects' methods, extra overhead is incurred that is not present with shared pages that can be accessed directly.

### 2.3.4 DSM Systems Overview

Next a set of tables illustrates a set of DSM systems classified as Software, Hardware or Hybrid (Hardware / Software) [6].

[6]From http://galeb.etf.bg.ac.yu/ vm/tutorial/multi/dsm/introduction/introduction.html

**DSM Software Systems**

| Name | Type of Implementation | Type of Algorithm | Consistency Model | Granularity unit | Coherence policy |
|---|---|---|---|---|---|
| IVY [7] | user-level library + OS modification | MSRW | sequential | 1Kb | invalidate |
| Mermaid [8] | user-level library + OS modifications | MSRW | sequential | 1Kb, 8Kb | invalidate |
| Munin [9] | runtime system + linker + library + preprocessor + OS modifications | type-specific (SRSW, MRSW, MRMW) | weak release | variable size objects | type-specific (delayed update, invalidate) |
| Midway [10] | runtime system + compiler | MRMW | entry, release, processor | 4Kb | update |
| Tread Marks [11] | user-level | MRMW | lazy release | 4Kb | update, invalidate |
| Blizzard [12] | user-level + OS kernel modification | MRSW | sequential | 32-128b | invalidate |
| Mirage [13] | OS kernel | MRSW | sequential | 512b | invalidate |
| Clouds [14] | OS, out of kernel | MRSW | inconsistent, sequential | 8Kb | discard segment when unlocked |

| **Name** | **Type of Implementation** | **Type of Algorithm** | **Consistency Model** | **Granularity unit** | **Coherence policy** |
|---|---|---|---|---|---|
| Linda [15] | Language | MRSW | sequential | variable (tuple size) | implem. dependent |
| Orca [16] | Language | MRSW | synchro dependent | shared data object size | update |
| DSM-PM2 [1] | runtime system + library | type-specific (MRSW, MRMW) | sequential, eager release, java consistency | 4kb | invalidate |

**DSM Hardware Systems**

| **Name** | **Cluster Conf** | **Net work** | **Type of algorithm** | **Consist Model** | **Granular unit** | **Coherence policy** |
|---|---|---|---|---|---|---|
| Memnet [17] | single processor, Memnet device | token ring | MRSW | sequential | 32b | invalidate |
| Dash [18] | SGI 4D/340 (4 PEs, 2-L caches), loc. mem. | mesh | MRSW | release | 16b | invalidate |
| SCI [19] | arbitrary | arbitrary | MRSW | sequential | 16b | invalidate |

| **Name** | **Cluster Conf** | **Net work** | **Type of algo-rithm** | **Consist Model** | **Granular unit** | **Coherence policy** |
|---|---|---|---|---|---|---|
| KSR1 [20] | 64-bit custom PE, I+D caches, 32M loc.mem. | ring-based hierar-chy | MRSW | sequential | 128b | invalidate |
| DDM [21] | 4 MC88110s, 2 caches, 8-32M local memory | bus-based hierar-chy | MRSW | sequential | 16b | invalidate |
| Merlin [22] | 40-MIPS Com-puter | mesh | MRMW | processor | 8b | update |
| RMS [23] | 1-4 pro-cessors, caches, 256M local memory | RM bus | MRMW | processor | 4b | update |

**DSM Hybrid Systems**

| Name | Cluster Conf + Network | Type of algorithm | Consist. Model | Granular unit | Coherence pol |
|---|---|---|---|---|---|
| PLUS [24] | M88000, 32K cache, 8-32M local memory, mesh | MRMW | processor | 4Kb | update |
| Galactica [25] | 4 M88110s, 2-L caches 256M local memory, mesh | MRMW | multiple | 8Kb | update / invalidate |
| Alewife [26] | Sparcle PE, 64K cache, 4M local mem, CMMU, mesh | MRSW | sequential | 16b | invalidate |
| FLASH [27] | MIPS T5, I+D caches, MAGIC contoller, mesh | MRSW | release | 128b | invalidate |
| Typhoon [28] | SuperSPARC, 2-L caches, NP controller | MRSW | custom | 32b | invalidate custom |
| Hybrid DSM [29] | FLASH-like | MRSW | release | variable | invalidate |
| SHRIMP [30] | 16 Pentium PC nodes, Intel Paragon routing network | MRMW | AURC, scope | 4Kb | update / invalidate |

**DSM Systems Consistency Models Comparison**

DSM performance is always the major concern. The DSM system, IVY [7], uses SC but performance is poor due to excessive data communication in the network. This major performance bottleneck is relieved by later systems, which use other relaxed models to improve efficiency. For example, Munin [9] made use of the weak Eager Release Consistency (ERC) model.

TreadMarks [11] went a step further, using the weaker Lazy Release Consistency (LRC). The relatively good efficiency and simple programming interface helps TreadMarks remain as the most popular DSM system. On the other hand, Midway [10] adopted an even weaker model called Entry Consistency (EC), but it requires programs to insert explicit statements to state which variables should be guarded by a certain synchronization variable. This makes the programming effort more tedious. Scope Consistency claims to be weaker than LRC, approaching the efficiency of EC. As the programming interface is exactly the same as that used by LRC, good programmability can be ensured.

## 2.4 DSM-PM2: An Overview

### 2.4.1 The PM2 runtime system

PM2 (Parallel Multithreaded Machine) [4] is a multithreaded environment for distributed architectures. It provides a POSIX like interface to create, manipulate and synchronize lightweight threads in user space, in a distributed environment. Its basic mechanism for internode interaction is the Remote Procedure Call (RPC).

[7] Using RPCs, the PM2 threads can invoke the remote execution of userdefined services. Such invocations can either be handled by a preexisting thread, or they can involve the creation of a new thread. While threads running on the same node can freely share data, PM2 threads running on distan nodes may only interact through RPC. This mechanism can be used either to send/retrieve information to/from the remote node, or to have some remote action executed. The minimal latency of a RPC is 6 micro sec over SISCI/SCI [44] and 8 micro sec over BIP [46]/Myrinet [43] on our local Linux clusters.
PM2 includes two main components. For multithreading, it uses Marcel, an efficient, userlevel, POSIXlike thread package. To ensure network portability, PM2 uses an efficient communication library called Madeleine [6], which was ported across a wide range of communication interfaces, including highperformance ones such as BIP [46], SISCI [44], VIA [45], as well as more traditional ones such as TCP, and MPI [37].
An interesting feature of PM2 is its thread migration mechanism that allows threads to be transparently and preemptively moved from one node to another during their execution. Such a functionality is typically useful to implement generic policies for dynamic load balancing, independently of the applications: the load of each processing node can be evaluated according to some measure, and balanced using preemptive migration. The key feature enablin preemptiveness is the isoaddress approach to dynamic allocation featured by PM2. The isomalloc

[7]From DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols - Gabriel Antoniu and Luc Bougé.

allocation routine guarantees that the range of virtual addresses allocated by a thread on a node will be left free on any other node. Thus, threads can be safely migrated across nodes: their stacks and their dynamically allocated data are just copied on the destination node at the same virtual address as on the o iginal node. This guarantees the validity of all pointers without any further restriction [3]. Migrating a thread with a minimal stack and no attached data, takes 62 micro sec over SISCI/SCI [44] and 75 micro ses over BIP [46]/Myrinet [43] on our local Linux clusters.

### 2.4.2 DSM-PM2: The illusion of common address space

**DSM-PM2**[1] provides the illusion of a common address space shared by all PM2 threads irrespective of their location and thus implements the concept page-based Distributed Shared Memory on top of the distributed architecture of PM2. But DSM-PM2 is not only a DSM layer for PM2, its goal is to provide a portable implementation platform for multithreaded DSM consistency protocols (see figure 2.2).

[7] Given that all DSM communication primitives have been implemented using PM2's RPC mechanism based on Madeleine [6], DSM-PM2 inherits PM2's wide network portability. However, the most important feature of DSM-PM2 is it customizability: actually, the main design goal was to provide support for implementing, tuning and comparing several consistency models, and alternative protocols for a given consistency model. As a starting remark, we can notice that all DSM systems share a number of common features. Every DSM system, aimed for instance at illustrating a new version of some protocol, has to implement again a number of core functionalities.

It is therefore interesting to ask: What are the features that need to be present in any DSM system? And then: What are the features that are specific to a particular DSM system? By answering these questions, we become able to build a system where the core mechanisms shared by the existing DSM systems are provided as a generic, common layer, on top of which specific protocols can be easily built. In our study, we limit ourselves to page-based DSM systems.

#### Access detection

Most DSM systems use page faults to detect accesses to shared data, in order to carry out actions necessary to guarantee consistency. The generic core should provide routines to detect page faults, to extract information related to each fault (address, fault type, etc.) and to associate protocolspecific consistency actions to a pagefault event.

### Page manager

Pagebased DSM systems use a page table which stores information about the shared pages. Each memory page is handled individually. Some information fields are common to virtually all protocols: local access rights, current owner, etc. Other fields may be specific to some protocol. The generic core should provide the page table structure and a basic set of functions to manipulate page entries. Also, the page table structure should be designed so that new information fields could be added, as needed by the protocols of interest.

### DSM communication

We can notice that the known DSM protocols use a limited set of communication routines, like sending a page request, sending a page, sending diffs (for some protocols implementing weak consistency models, like release consistency). Such a set of routines should also be part of the generic core.

### Synchronization and consistency

Weaker consistency models, like release, entry, or scope consistency require that consistency actions be taken at synchronization points. In order to support these models, the generic core should provide synchronization objects (locks, barriers, etc.) and enable consistency actions to be associated to synchronization events.

### Thread-safety

Modern environments for parallel programming use multithreading. All the data structures and management routines provided by the generic core should be threadsafe: multiple concurrent threads should be able to safely call these routines.

## 2.5 Conclusion

After this DSM overview survey is little to say that a lot of effort has already been done on trying to find the best solutions for the several DSM implementation problems. Both software and hardware research fronts thrive searching for these best options and try to improve and prove themselves as the path to follow. Maybe the final solution will be something in between as a hybrid solution. Difficultly there will be consensus in what system, algorithm, coherence protocol or consistency model is better suited for each and every problem because each one requires a different approach.

Distributed Shared Memory presents itself as a good technological answer to the crescent demanding world of intensive distributed computation. Present research as proven this technology valid and useful as it aims to make the life

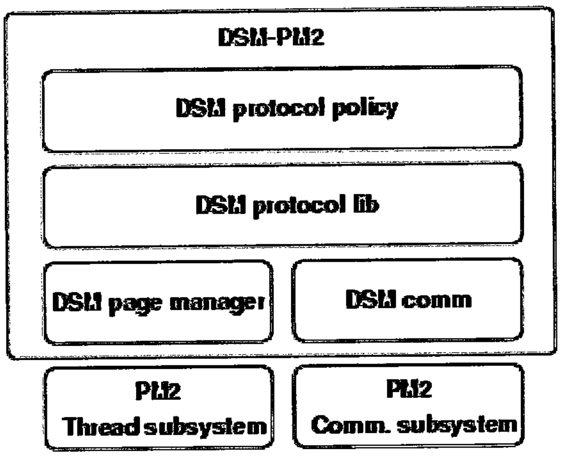


Figure 2.2: Overview of the DSM-PM2 Software Architecture

of programmers easier. The next step will be to try to leverage and assess an acceptable cost / efficiency tradeoff and finally consolidate this knowledge and try to come up with standards and proposals for real commercial future DSM systems.

# Chapter 3

# State of the Art - Distributed Constraint Programming

This chapter will present the reader a representative survey on the currently available DCP technology.

## 3.1 Distributed Constraint Programming - from GC to Distributed AJACS

### 3.1.1 GC Concept

The GC [39] (Generic Constraint) is a constraint propagation system, with three different implementations of finite variables domains (FD, FDD and FDIU). It is a system that uses the OO approach implemented in Java. The constraint propagation and variables domains are explained at the application and implementation level and some examples from the litterature are presented in order to clarify the construction of applications with GC.

#### Finite Domains

The GC (as for AJACS) will implement its contraint system over Finite Domains. A finite domain is a finite set of non-negative integers. A notation for a finite domain can be "n,...,m".

#### Variables

The domain of a variable is the set of values that it can assume. GC implements three distinct classes of variables over finite domains.

- FD (Finite Domain) variables: These are represented by a single positive numbers interval. Variables with this kind of representation are caracterized by the manipulation of the maximum and minimum values of its interval.

- FDD variables: These are FD variables where no longer the interval extremes are manipulated but where all values of the domain between its extremes) are considered.

- FDIU variables: They are also FD variables but where the values of the domain are unions of intervals.

Over these variable concept GC implements all basic operations like checking of the domain is empty or singleton do copies of variables, get the first/last element, to know which is the next element, construct singleton domains etc.

### Constraints

Constraints are in the core of GC in the end its what it is all about. A constraint is a relation between variables. The inclusion of a new constraint in the system will create new dependencies in the variable(s) that interveen in that restriction. The imposition of a constraint will frequently narrow the variables domain. Once that happens all the dependent variable's domains will be analyzed and updated too, if necessary. In GC, constraints are classes that follow an inheritance hierarchy. New constraints subclasses can be added with the definition of appropriate *localUpdate(n)*, method. The *localUpdate* method is responsible for the actualization of the nth variable domain for that constraint.

### Iterators and Strategies

Propagation by it self cannot construct the solutions of a constraint problem. There is the need to walk through the range of possibilities. The idea behind Iterators is to reduce variables to single value instantiations. By "iterating" through the variables and combining this with the propagation mechanism its then possible to obtain the solution(s) of the problem. The Iterators are associated with a specific search strategy. This can be "depth-first", "left-right", "breadth-first", "first-fail".

### Propagation

Propagation is the mechanism that allows the validation of the constraints after some change, or reduction, on the system variables has occurred. These changes will occur by the iteration of variables. Jointly with the GC Iterators these are the building blocks for finding the solutions of a finite domain constraint system

The root of the constraint class hierarchy defines the *localUpdate(n)* method, which is responsible for the propagation mechanism and is triggered by the installation of the constraint. If there is a change in the corresponding domain, all the constraints involving that variable will be re-evaluated, until a fixpoint is reached, i.e. no more changes exist, so this method will be repeatedly invoked until the provoked changes produce additional changes.

**Example**

**N Queens**

- The problem: Lay down N queens on a N x N chess-board so that there is no couple of queens threatening each other.
- The variables: We have N variables that take values from 1 to N. For instance, to N = 4, the solution [3,1,4,2] means that, in row 1 the queen must be placed at column 3, in row 2 the queen must be placed at column 1, and so on.
- The constraints: To avoid the queens from threatening each other we will implement a constraint. If Ci and Cj are queens placed in columns i and j respectively, to avoid them to attack each other we must have

$$\forall j > i \begin{cases} C_j \neq C_i \\ C_j \neq C_{i+(j-i)} \\ C_j \neq C_{i-(j-i)} \end{cases}$$

The constraint **NoAttack**, for 2 variables, implements this restrictions.

```
public class NoAttack extends gc.fdiu.VV.Constraint{
    private int c;
    public NoAttack(Variable VX, Variable VY, int V) {
        super(VX,VY);
        c=V;
    };
    public void localUpdate(int n) {
        int m=1-n;
        if (env[m].ground ()) {
            env[n].clear (env[m].min);
            env[n].clear (env[m].min + c);
```

```
                env[n].clear (env[m].min - c);
            }
            env[n].updateMinMax ();
        }
    }
```

Since this constraint is for two variables, a method to apply it at all variables in the list of queens, is required. Next an example piece of code that defines a new constraint *NoAttack* with its corresponding *localUpdate* method. Note: *env[]* is an array of variables that make part of that constraint, in this case (NoAttack) there are two (the two queens).

```
static void safe (Variable queen[], int n) {
    for (int i=0; i<n; ++i)
       for (int j=i+1; j<n; ++j)
           new NoAttack(queen[i], queen[j], j-i).tell();
}
```

### 3.1.2 AJACS Concept

The AJACS [3] (Another Java Constraint Programming System) is a toolkit for Concurrent Constraint programming implemented in the Java language. It comes as a successor of the Constraint Programming also in Java in that it represents an attempt to deal with some of GC [39] inadequacies in terms of performance whilst providing a setting which is adequate to express problems in a way that can be easily solved in a parallel execution environment, as provided by a concurrent programming setting.

As said before the AJACS implementation (figure 3.1) is founded on similar concepts as in its precusor GC refined now with the introduction of a "Store" structure that represents the set of values as in a state snapshot, and a "Problem" structure that holds the initial store as also all the constraints static information (e.g. information about which constraints are associated to each variable).

#### Values

In AJACS one "value" represents a subset of its variable domain. One value is considered basic (ground) when it contains a singleton value.

#### Variables

There is no explicit concept of "variable" in AJACS. It is an abstract notation to represent the set of values located in the same index in some store.

### Stores

A Store is an indexed collection of values. The objective behind it is to create successive similar states (in respect to the number of contained values) in which the values associated to some index represent a variable domain. This is why there is no explicit representation of Variable in AJACS since it is obtained by the concept of Store.

### Constraints

Constraints in AJACS follow the same principle as in GC, they are the relations between variables in a problem. AJACS constraints are the mechanism reponsible for the propagation of results to the other variables of the state.

### Problem

The Problem defines the set of variables with its associated initial domain, i.e. a state. Morever it holds the set of constraints over those same variables. The objective behind the fomulation of a Problem is to determine its solutions, i.e. the basic values for all the variables that are consistent with the imposed constraints.

### Search & Strategy

The order by which the variables are instatiated defines a different solution space configuration. The AJACS Search is a repetitive procedure upon the problem possibilitie space until a solution is found. The search procedure may rely upon a given search strategy. The strategy is applied on a given state (store) to specify its next state. Part of this strategy is to decide which of the non ground variables (the ones that are not singletons), will be selected; and for the chosen variable the way in which the domain reduction will be performed. Normally this is accomplished by the determination of the variable single value.

### Example

Note: Example taken from AJACS [3] for illustration purposes.

We will give an example of how to work with the AJACS classes system, with the classical N-Queens. Consider N=4, for simplicity. We have 4 Values: u0, u1, u2 and u3, defined by:

```
u0 = u1 = u2 = u3 = new Value(1,4).
```

We must implement a constraint, lets call it *NoAttack*. NoAttack is a subclass of Constraint, and assures that two queens do not threatening each other. The problem, and its initial store is defined by:

```
p = new Problem([u0,u1,u2,u3]).
```

In order to update the the lists C and Cv, we add the constraints to the problem.

```
for (i=0; i<=2; ++i)
   for (j=i+1; j<=3; ++j)
       p.add (new NoAttack (i,j))
```

Let C1 be the constraint NoAttack(0,1), C2 be the constraint NoAttack(0,2), etc. C1.env=[0,1], C2.env=[0,2], etc. Adding the constraints to the problem turns List C into C1, C2, C3, C4, C5, C6 and List Cv into Cv = Cv0=(C1,0), (C2,0), (C3,0), Cv1=(C1,1), (C4,0), (C5,0), Cv2=(C2,1), (C4,1), C6,0), Cv3=(C3,1), (C5,1), (C6,1).

Now we could apply a search, lets define it:

```
s = new Search(p, st=new StratFirst()).
```

The solution is given by s.solution(). Figure 3.2 shows the sequence of stores generated by solution.

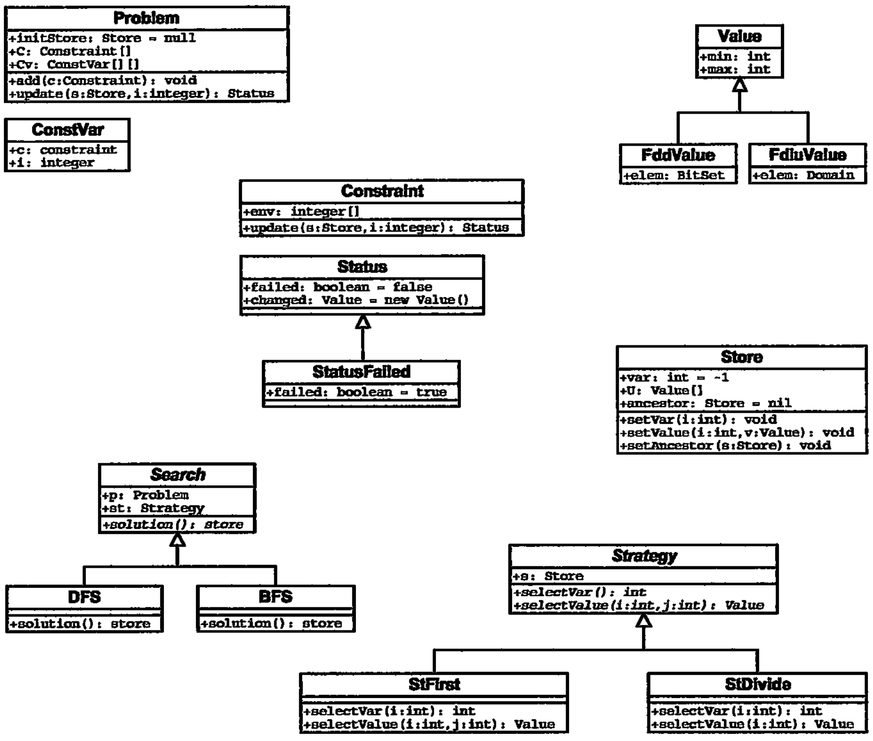


Figure 3.1: All Classes of AJACS

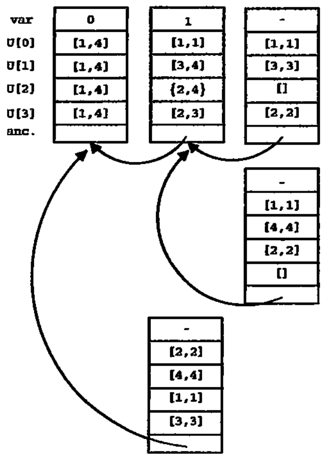


Figure 3.2: DFS search applied to 4-Queens problem

### 3.1.3 AJACS over Hyperion for Distributed Execution

#### The Hyperion System

Hyperion [2] is a Java system that aims for Java compilation to native code with a run-time library that executes Java threads in a distributed-memory environment. This allows a Java programmer to view a cluster of processors as executing a single Java virtual machine. The separate processors are simply resources for executing Java threads with true parallelism, and the run-time system provides the illusion of a shared memory on top of the private memories of the processors. The environment is available on top of several UNIX systems and can use a large variety of communication interfaces thanks to the high portability of its run-time system. Hyperion [2] was developed at the University of New Hampshire and comprises a Java-bytecode-to-C translator and a run-time library for the distributed execution of Java threads. Hyperion has been built using the PM2 distributed, multi-threaded run-time system from the cole Normale Suprieure de Lyon [12]. As well as providing lightweight threads and efficient inter-node communication, PM2 provides the generic distributed-shared-memory layer, DSM-PM2 [1]. Another important advantage of PM2 is its high portability on several UNIX platforms and on a large variety of communication interfaces and protocols (BIP [46], SCI [44], VIA [45], MPI [37], TCP). Thanks to this feature, Java programs compiled by Hyperion can be executed with true parallelism in all these environments.

### AJACS over Hyperion

The idea behind this other project (AJACS over Hyperion) was to implement and evaluate the AJACS [3] system over the Hyperion solution. For details refer to AJACS [3].

## 3.2 Other DCP Systems

### 3.2.1 DisChoco: A Platform for distributed constraint programming

DisChoco [40] is a Java library built on top of the Choco Java open-source solver. Communication is performed via the simple agent communication infrastructure (SACI) if the agents are implemented on distant machines. Otherwise (simulation) the communication is performed via a local communication simulator. The implementation of DisChoco was made to offer a modular software architecture which accepts extensions easily. DisChoco can be used for simulation of a multiagents environment on a single Java virtual machine, or performed in an environment physically distributed for a realistic use. Each agent in the environment is executed asynchronously in a separate execution thread, and communicates with its peers through message exchange. DisChoco takes into account an agent with a complex local problem, message loss, message corruption, and message delay.

### 3.2.2 The Mozart Programming System

The Mozart system [5] provides state-of-the-art support in two areas: open distributed computing and constraint-based inference. Mozart implements Oz, a concurrent object-oriented language with dataflow synchronization. Oz combines concurrent and distributed programming with logical constraint-based inference, making it a unique choice for developing multi-agent systems. Mozart is an ideal platform for both general-purpose distributed applications as well as for hard problems requiring sophisticated optimization and inferencing abilities.

#### Constraint Programming

Oz is a powerful constraint language with logic variables, finite domains, finite sets, rational trees and record constraints. The system is competitive in performance with state-of-the-art commercial solutions, but is much more expressive and flexible, providing first-class computation spaces, programmable search strategies, a GUI for the interactive exploration of search trees, parallel search engines exploiting computer networks, and a programming interface to implement new and efficient constraint systems.

**Open Distributed Computing**

The Mozart system [5] is an ideal platform for open distributed computing: it makes the network completely transparent. The illusion of a common store is extended across multiple sites and automatically supported by very efficient protocols. In addition, full control is retained over network communication patterns, permitting very efficient use of network resources. Furthermore, reliable, fault tolerant applications can easily be developed.

### 3.2.3 Disolver: The Distributed Constraint Solver

Disolver [38] is a constraint-based optimization engine. It relies on an extended **Constraint Programming** paradigm which seamlessly integrates local search. It is especially designed to run on **multi-core, parallel and distributed** architectures and comes out as a C++ library.

Disolver is the first suite devoted to combinatorial problem solving in distributed and Grid-like infrastructures. It initially came out as a research tool. However, it was also used to solve large industrial problems. So far, it has been used to address the following problems,

- resource allocation,
- publishing,
- e-contracts negotiation,
- scheduling,
- distributed scheduling,
- configuration,
- model-checking,
- capacity planning,
- packing.

Therefore, it claims to be robust enough to tackle very large problems (involving tens of thousands constraints over thousands of variables).

For the Disolver general architecture refer to figure 3.3.

### 3.2.4 Alice System

Alice ML [41] is a functional programming language based on Standard ML, extended with rich support for concurrent, distributed, and constraint programming. Alice ML extends Standard ML with several new features:

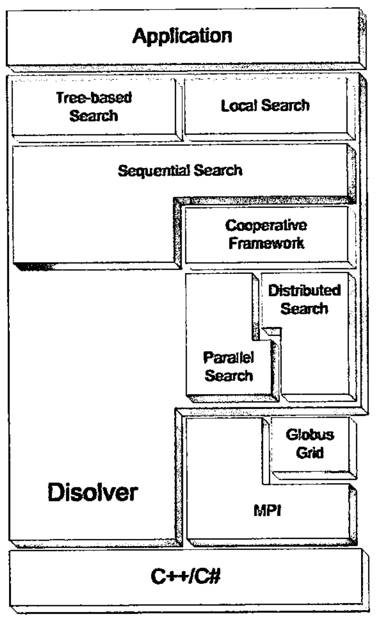


Figure 3.3: Disolver General Architecture

- **Futures**: laziness and light-weight concurrency with implicit data-flow synchronisation;
- **Higher-order modules**: higher-order functors and abstract signatures;
- **Packages**: integrating static with dynamic typing and first class modules;
- **Pickling**: higher-order type-safe, generic platform-independent persistence;
- **Components**: platform-independence and type-safe dynamic import export of modules;
- **Distribution**: type-safe cross-platform remote functions and network mobility;
- **Constraints**: solving combinatorical problems using constraint propagation and programmable search.

The Alice [41] System is a rich open-source programming system featuring the following tools:

- **Virtual machine**: a portable VM with support for just-in-time compilation;
- **Interactive system**: an interpreter-like interactive toplevel with easy graphical interface;
- **Batch compiler**: separate compilation;
- **Static linker**: type-safe bundling of components;
- **Inspector**: a tool for interactively inspecting data structures;
- **Explorer**: a tool for interactively investigating search problems;
- **Gtk+**: a binding for the Gnome toolkit GUI library;
- **SQL**: a library for accessing SQL databases;
- **XML**: a simple library for parsing XML documents .

## Distribution Concept in Alice

Alice also provides high-level means for processes at different sites to communicate directly.

**Tickets** The first mechanism that allows sites to establish peer-to-peer connections is offer and take. A process can create a package and make it available to other processes. Offering a package opens a communication port and returns an URI for that port. The URI is called a ticket. A ticket can be transferred to other sites, say by email or through some web document. Other sites can then obtain the available package using the ticket. Of course, the exporting site may also obtain it itself. In general, *take* establishes a connection to the communication port denoted by the ticket, and retrieves the offered package. Transfer of the package is defined by the pickling/unpickling semantics, i.e. the whole closure of the package is transferred to the client, including any code representing embedded functions.

**Proxies** Tickets are intended merely as a means to establish an initial connection between sites. All subsequent communication should be dealt with by the functions in the offered package. Alice provides a very simple feature to enable this idiom: proxies. A proxy is basically an RPC stub, a mobile reference to a stationary function that can be used in place of the function it references.

The library function

```
Remote.proxy : ('a -> 'b) -> ('a -> 'b)
```

creates a proxy for an arbitrary function:

```
fun fib (0 | 1) = 1
  | fib n       = fib (n-1) + fib (n-2)
val fib : int -> int = _fn
val fib' = Remote.proxy fib
val fib' : int -> int = _fn
```

The resulting function has the same type as the function it references. When it is applied, all arguments are forwarded to the original function, and the result is transferred back:

```
fib' 20
val it : int = 10946
```

Pickling a proxy does only pickle the respective reference and not the referenced function. When the proxy is transferred to a different site (e.g. by offering it as part of a package) and then applied at that site, all arguments will be automatically transferred to the site hosting the referenced function, the result will be computed there, and finally transferred back to the client site. That is, applying a proxy is practically a remote procedure call (RPC). Transfer is again defined by pickling semantics.

# Part III

# Description: Distributed Constraint Solver - AJACS/C

Chapter 4

# AJACS/C Description - Distributed Patterns and Implementation

This chapter will describe the AJACS/C Constraint Solver to the reader. First present its scope and model, later on how it may be integrated with the DSM-PM2 library [1] and finally present the implementation details as a more in-depth look of the AJACS/C architecture.

## 4.1 AJACS/C Model Description

Due to its organization AJACS/C (as AJACS [3] is) produces independent states as result of state expansion. Independent in the sense that each store (plus the constraint problem containing the constraints them selfs) carries all the information necessary to be considered a possible solution for a given problem. This state independence will be the basis for a distributed concept to take shape since in theory it shall be possible to parallelize constraint problem solving by spreading each produced state among several processing units without too much foreseen interaction. This way all processing nodes should be able to 'walk' through the problem space with the minimal knowledge or awareness of each other.

The minimal information each processing node requires, for its state iteration and propagation, is to know:

- Where to look at for new states to search;
- Where to store the expanded potential solutions, i.e. the states that resulted from a successful propagation;

- where and how to signal the eventual found solutions to a problem master controller.

It is foreseen to be relatively easy to experiment DCP over DSM-PM2 on a 'native' way, i.e., using a direct approach where the C language is to be used for implementation. In result the AJACS/C inherits the basic AJACS Model and implements it in the C language environment.

### 4.1.1 AJACS/C Static Model

As said before the AJACS/C constraint solver, to be developed fully in the C language, inherits most of its model carachteristics from the AJACS [3] system, that was implemented in Java. Please refer to AJACS [3] for full details on the AJACS model or refer to figure 3.1 (All Classes of AJACS) for a brief picture of the AJACS classes.

The first challenge was then to sucessufuly implement / port the AJACS model to the C environment. To accomplish this most of the AJACS object characteristics were ported directly, when possible to C data structures, namely to *structs.*

See in figure 4.1 a representation of the AJACS/C static structure final look.

All the information regarding a certain Constraint Program, involved constraints, number of variables necessary to map the particular problem as for its initial state, has its heart in the *Problem* structure.

The **Problem** is the static core of a given constraint system and is the essencial part of the constraint problem initialization. It contains information regarding:

- **"sInit"**: The initial state of the problem (the initial store with the first set of values for each variable);
- **"Constraints List"**: The list of all the problem constraints;
- **"List of Constraints per var"**: A special list of constraints that holds "meta-information" regarding which constraints are associated to each variables.

This last item is very useful to know at any time exactly which is the list of constraints associated to each variable of the problem. This is crutial information to know at propagation time where all constraints, associated to some variable which its domain value has been changed, needs to be satisfied.

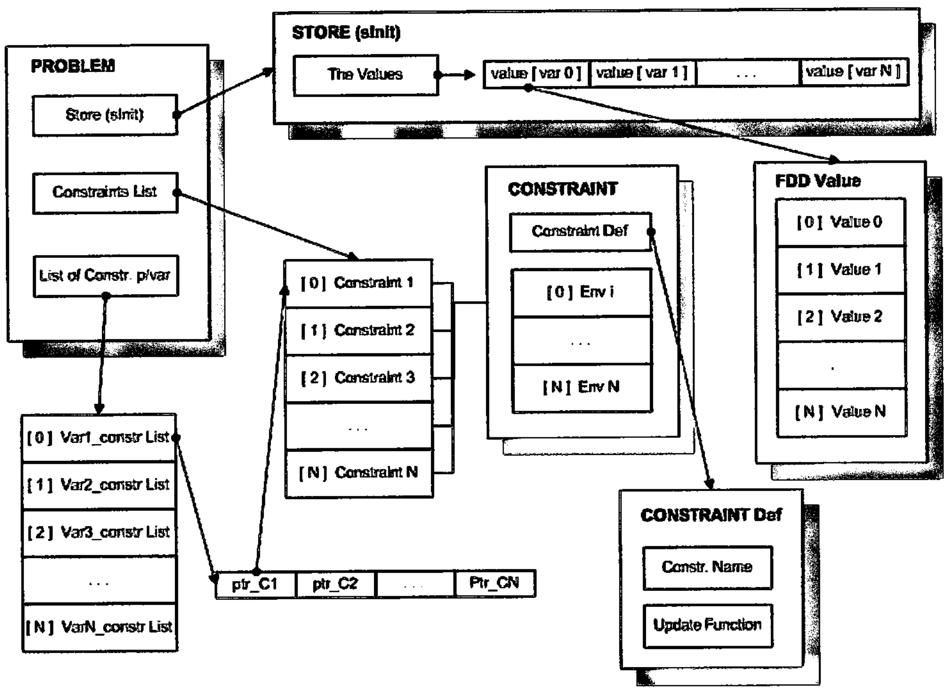


Figure 4.1: AJACS/C Static Model

If the *Problem* structure is the "heart" of the constraint problem the *Constraint* structure is the "brain".

The **Constraint** contains static information regarding all the relations between the variables. The *Constraint Def* holds the definition of the constraint namely its name and the associated update function. The *Update Function* is called during propagation and contains the list of steps necessary to update the constraint's involved variables domains.

Finally the **Store** structure holds information regarding the current state of the variables domains, represented by *FDD Values*. An **FDD Value** represents a finite domain associated to some variable. *sInit* is the specific store that holds the initial state of the problem.

### 4.1.2 AJACS/C Dynamic Model

In the last section the AJACS/C static structure was presented. That represents the information necessary to caracterise a problem, in other words all the building blocks necessary to identify one problem as specific. By knowing the "Problem" one application understands *what* the problem is but still is not sufficient to know *how to* solve it.

For solving a constraint problem the application still needs to know:

1. How to select and expand new states (stores) from the initial state?
2. How to search over the expanded problem space (Depth-First search, Breadth-First search, other, ...) and recognize the problem solution(s)?

For 1) the answer is that the constraint solver application needs to implement some *Strategy* to know how to split some state into its child states (being those potential solutions for the problem). The **Strategy** structure contains information on how to select and split one store in other potential solution stores. The adopted mechanism in AJACS/C is to reduce one variable, on the currently iterated store, turning it into a ground value (singleton). Due to this value domain reduction, the application will then trigger the problem **propagation** so that all the other variables may be updated according to the introduced change.

For 2) the constraint solver application needs to implement some *Search* mechanism to garantee a successful, efficient and full iteration on all the potential problem space. The **Search** procedure will implement some algorithm search and in conjunction with the *Strategy* walk through all the problem space and find all the possible solutions (if any). AJACS/C currently implements the **DFS** search mechanism by using a stack **(LIFO)** for storing all the splitted states. Stores are evaluated one by one from the top of the stack and any splitted states stored also at top of the stack.

Figure 4.2 illustrates the AJACS/C Dynamic Model. Roughly speaking the **Problem** injects all the necessary constraint static information. The **Search** supplies the search algorithm (AJACS/C uses DFS by default but others could be used). Stores **S12** and **S133** (in blue) are identified as problem solutions which means that all the contained variable values are ground - all variables have been instantiated with some value.

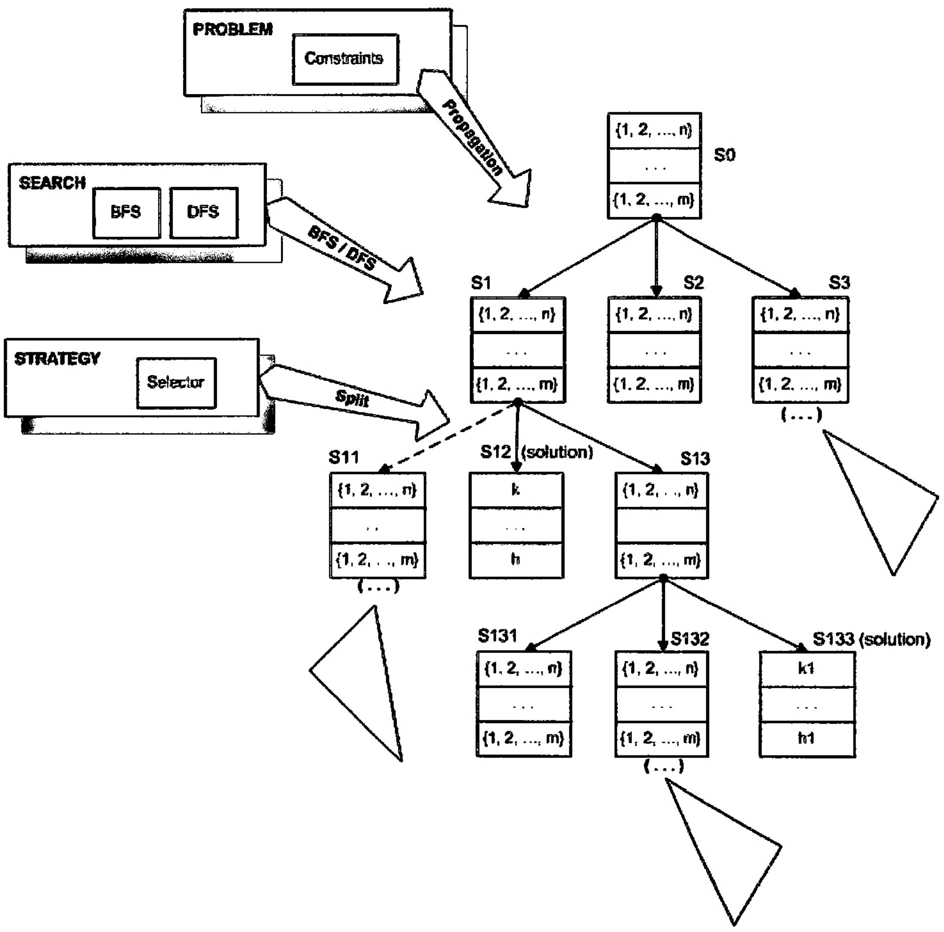


Figure 4.2: AJACS/C Dynamic Model

### 4.1.3 AJACS/C Extensability

With this model AJACS/C is extensible in the way that new constraints can easily be added to the system. To define a new constraint the programmer just needs to define the associated **Update Function** with the intended constraint effect.

To note also that the AJACS/C dynamic properties are also extensible in what the *Search* and *Strategy* is concerned. The programmer is allowed to switch or add alternatives, for instance propose new search methods with specific objectives or propose different store split strategies in order to change how the constraint solver behaviour works.

## 4.2 Distributed Sharing Patterns with AJACS/C and DSM-PM2

AJACS/C implementation allows it for distribution, due to the store independence feature. The next step was the integration with the DSM-PM2 [1] module (see figure 4.3) for a real distributed example. For this the focus now turned to the PM2 and DSM-PM2 offered capabilities. The objective is the implementation of DSM-PM2 application examples that may sucessfully integrate and run AJACS/C problems.

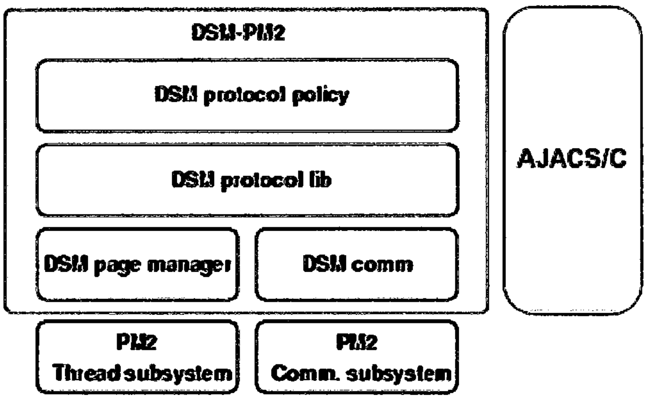


Figure 4.3: DSM-PM2 AJACS/C Architecture

With the AJACS/C DSM-PM2 integration in mind two distribution patterns were designed for experimentation:

1. Centralized Distribution Pattern
2. Local Distribution Pattern

### 4.2.1 Centralized Distribution Pattern

This Pattern designates one cluster node has the **master node** and the remaining as the **worker nodes** (different nodes meaning different machines in the PM2 configured cluster). The master and worker profiles are triggered/enrolled at run-time execution where the first configured cluster node will assume a master profile and the remaining the worker profile (see figure 4.4).

The idea behind the master profile is for it to maintain a central DSM structure (hereby the pattern name) that holds all the current, still to be investigated, problem states. All nodes have write and read access to this central structure. Every worker will be allowed to **get stores** (new jobs) and **put stores** (the resulting state produts of the last iteration and propagation).

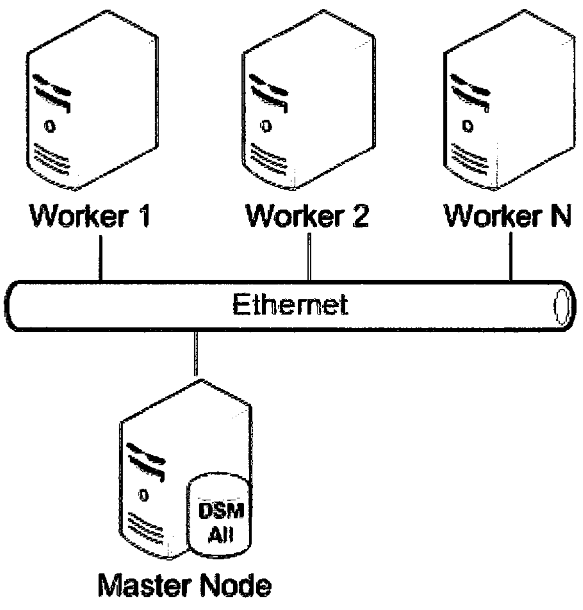


Figure 4.4: Centralized DSM structure Architecture

For a more complete and efficient approach the designed pattern will make sure that after the initialization phase, compreending the central data structure creation and remote execution of all the threads on all nodes, the node that took the master profile will spawn an additional local thread to endorse the worker profile, this way all machines/nodes will behave as workers after the initial initialization step has been performed.

This way the **centralized DSM data-structure**, resident in the Master Node, will be the data communication link between all the nodes, the

DSM-PM2 coherence protocol will abstract the user to the synchronization overhead management of the shared structure and assure the correct access and behaviour from all threads in all nodes in a safe and coherent manner.

See figure 4.5 for the Master centralized pattern pseudocode:

```
init_states = search_initial_states(sInit);
dsm_list.put(init_states);
FOR i = 1 TO n
        Worker[i] = new RemoteWorker(dsm_list);
Worker[0] = new LocalWorker(dsm_list);
WHILE < not_all_finish() > DO
        wait();
printSolutions();
```

Figure 4.5: Centralized Distribution Pattern - Master Algorithm

See figure 4.6 for the Worker centralized pattern pseudocode:

```
WHILE < dsm_list not empty > DO
   j = dsm_list.get();
   L = search_solutions(j);
   FOREACH l in L DO
           IF <l is solution>
           THEN print_solution(l);
           ELSE dsm_list.put(j);
```

Figure 4.6: Centralized Distribution Pattern - Worker Algorithm

This distribution pattern itends to verify what is the application behaviour (DSM-PM2 with AJACS/C) when the problem space is shared in real-time by all working nodes during the search execution. This model is more simple and easier to implement but more communication between the nodes is expected. Results will be evaluated in chapter 5 of this thesis.

### 4.2.2 Local Distributed Pattern

In this distribution model all workers have a **dedicated DSM local data structure** to manage its share of the new states expansion, so every worker **gets** and **puts** jobs directly from/into its local data stucture (see figure 4.7).

On execution start the initial state is spawned and the result child stores are distributed among all the workers (including the one that initilized the search) on a round-robin fashion. After this point all workers start their search independently.

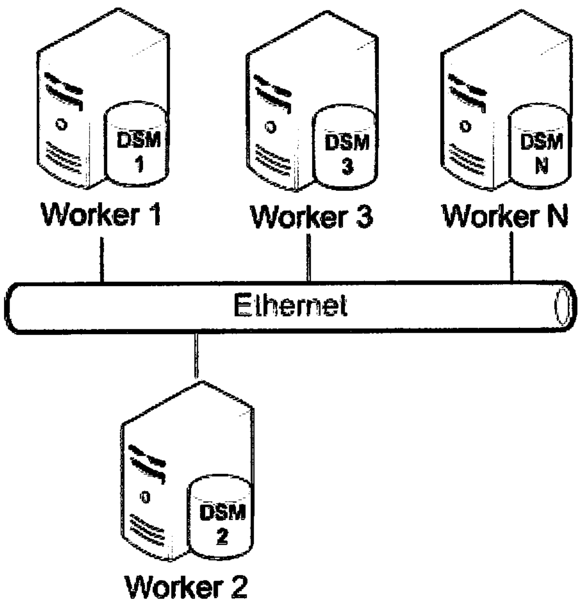


Figure 4.7: Local DSM structure Architecture

See figure 4.8 for the Master node local pattern pseudocode:

```
init_states = search_initial_states(sInit);
LET w = 0;
FOREACH k in initial_states
        Worker[w].local_dsm_list.put();
        w = (w + 1) % number_nodes;
```

Figure 4.8: Local Distribution Pattern - Master Algorithm

When the worker nodes have their share of the problem space inside its DSM local structure, each node will start the search independently from each other. All solutions are signaled by each worker node when found.

See figure 4.9 for the Worker node local pattern pseudocode:

This distribution pattern itends to verify what is the DSM-PM2 AJACS/C application behaviour when the problem space is equaly, or similarly equaly, distributed among the DSM structures of all the nodes before the actual start of the search execution. This model is closer to a **full parallelization mechanism**, less simple to implement because each node must manage its DSM local data structure but in the other hand little communication is

```
WHILE < local_dsm_list is not empty >
     nextS = local_dsm_list.get();
     L = search_solutions();
     FOREACH k in L DO
       IF < k is solution >
       THEN printSolution(k);
       ELSE local_dsm_list.put(k);
END
```

Figure 4.9: Local Distribution Pattern - Worker Algorithm

to be expected between the network nodes. Examples and results will be evaluated in section 5 of this thesis.

## 4.3 AJACS/C Implementation Details - API

This section will present the AJACS/C implementation details. The idea is to present an inner look (and evidence) on the developed constraint solver (AJACS/C) and can be of more interest to the hardcore reader. The general reader may skip this section.

### 4.3.1 bittarr.h

**Functions Overview**

- `ba_init();`
- `ba_new();`
- `ba_copy();`
- `ba_assign();`
- `ba_value();`
- `ba_toggle();`
- `ba_all_assign();`
- `ba_ul2b();`
- `ba_count();`
- `ba_intersection();`
- `ba_union();`

- `ba_diff();`
- `ba_complement();`

### Structs & Variables

```
typedef struct
{
  elem_t size; bit *vector;
} BitVector;
```

### ba_init

Declaration:
```
elem_t ba_init(void);
```

```
/*
   PRE:  Must be called before use of any other ba_ functions.
   Should only be called once.
   POST: Returns the number of values that can be stored in one
   of type 'bit'.  If <limits.h> does not define CHAR_BIT' then
   the module global variable 'BITS_SZ' has been set to the
   appropriate value.
*/
```

### ba_new

Declaration:

```
bit *ba_new(const elem_t nelems);
```

```
/*
   PURPOSE: dynamically allocate space for an array of 'nelems'
   bits and initalize the bits to all be zero.
   PRE:  nelems is the number of Boolean values required in an
   array
   POST: either a pointer to an initialized (all zero) array of
   bit OR space was not available and NULL was returned
   NOTE: calloc() guarantees that the space has been initialized
```

```
    to 0.
    Used by: ba_ul2b(), ba_intersection() and ba_union().
  */
```

**ba_copy**

Declaration:

```
void ba_copy(bit dst[], const bit src[], const elem_t size);

  /*
    PRE:  'dst' has been initialized to hold 'size' elements.
    'src' is the array of bit to be copied to 'dst'.
    POST: 'dst' is identical to the first 'size' bits of 'src'.
    'src' is unchanged.
    Used by: ba_union()
  */

   ---------------------------------------------------------------
                    Assigning and Retrieving Values
   ---------------------------------------------------------------
```

**ba_assign**

Declaration:

```
  void ba_assign(bit arr[], elem_t elem, const bool value);

  /*
    PURPOSE: set or clear the bit in position 'elem' of the
    array 'arr'
    PRE:     arr[elem] is to be set (assigned to 1) if value
    is TRUE, otherwise it is to be cleared (assigned to 0).
    POST:    PRE fulfilled.  All other bits unchanged.
    SEE ALSO: ba_all_assign()
    Used by:  ba_ul2b()
  */
```

**ba_value**

Declaration:

```
bool ba_value(const bit arr[], const elem_t elem);

/*
   PRE:  arr must have at least elem elements
   POST: The value of the 'elem'th element of arr has been
   returned (as though 'arr' was just a 1-dimensional array
   of bit)
   Used by: ba_b2str() and ba_count()
*/
```

**ba_toggle**

Declaration:

```
void ba_toggle(bit arr[], const elem_t elem);

/*
   PRE:  arr must have at least elem elements
   POST: The value of the 'elem'th element of arr has been
   flipped, i.e. if it was 1 it is 0; if it was 0 it is 1.
   SEE ALSO: ba_complement()
*/
```

**ba_all_assign**

Declaration:

```
void ba_all_assign(bit arr[], const elem_t lsize, const
                   bool value);


/*
   PRE:  arr has been initialized to have *exactly* size
   elements.
   POST: All 'size' elements of arr have been set to 'value'.
   The array is in canonical form, i.e. trailing elements
   are all 0.
```

```
    NOTE: The array allocated by ba_new() has all elements 0
    and is therefore in canonical form.
    SEE ALSO: ba_assign()
    Used by: ba_ul2b()
  */
```

**ba_ul2b**

Declaration:

```
bit *ba_ul2b(unsigned long num, bit *arr, elem_t *size);

  /*
    PRE:  Either
            'arr' points to space allocated to hold enough
    'bit's to represent 'num' (namely the ceiling of the base
    2 logarithm of 'num').  'size' points to the number of bit
    to use. OR 'arr' is NULL and the caller is requesting that
    enough space be allocated to hold the representation before
    the translation is made. 'size' points to space allocated
    to hold the count of the number of bit needed for the
    conversion (enough for MAXLONG).
    POST: A pointer to a right-aligned array of bits
    representing the unsigned value num has been returned and
    'size' points to the number of 'bit's needed to hold the
    value. OR the request to allocate space for such an array
    could not be granted

    NOTES: - The first argument is unsigned.
           - It is bad to pass a 'size' that is too small to
    hold the bit array representation of 'num' [K&R II, p.100].
           - Should the 'size' be the maximum size (if size > 0)
    even if more bits are needed?  The user can always use a
    filter composed of all 1s (see ba_all_assign()) intersected
    with result (see ba_intersection()).
  */
```

**ba_b2str**

Declaration:

```
char * ba_b2str(const bit arr[], const elem_t size,
                  char * dest);

 /*
   PRE: 'arr' is a bit array with at least 'size' elements.
   Either 'dest' points to enough allocated space to hold
   'size' + 1 characters or 'dest' is NULL and such space
   is to be dynamically allocated.
   POST: Either 'dest' points to a null-terminated string
   that contains a character representation of the first
   'size' elements of the bit array 'arr'; OR  'dest' is
   NULL and a request to dynamically allocate memory for a
   string to hold a character representation of 'arr' was
   not be granted.
   Used by: ba_print()
 */
```

**ba_print**

Declaration:

```
bool ba_print(const bit arr[], const elem_t size, FILE * dest);

----------------------------------------------------------------
                    Mathematical Applications
----------------------------------------------------------------
```

**ba_count**

Declaration:

```
unsigned long ba_count(const bit arr[], const elem_t size);

 /*
   PRE:  'arr' is an allocated bit array with at least 'size'
   elements
   POST: The number of 1 bits in the first 'size' elements of
   'arr' have been returned.
   NOTE: if arr is not in canonical form, i.e. if some unused
   bits are 1, then an unexpected value may be returned.
 */
```

**ba_intersection**

Declaration:

```
bool ba_intersection(bit first[], bit second[], bit * result[],
                     const elem_t size_first, const elem_t size_second);

/*
   PRE:  'first'  is a bit array of at least 'size_first'  elements.
   'second' is a bit array of at least 'size_second' elements.
   'result' points to enough space to hold the as many elements as the
    smallest of 'size_first' and 'size_second'; OR 'result' points to
   NULL and such space is to be dynamically allocated.
   POST: TRUE has been returned and 'result' points to a bit array
   containing the intersection of the two arrays up to the smallest of
   the two sizes; OR  FALSE has been returned and 'result' pointed to
   NULL (a request was made to allocate  enough memory to store the
   intersection) but the required memory could not be obtained.
   NOTE: This runs faster if the 'first' array is not smaller than
   'second'.
 */
```

**ba_union**

Declaration:

```
bool ba_union(bit first[], bit second[], bit * result[], const elem_t
              size_first, const elem_t size_second);


/*
   PRE:  'first'  is a bit array of at least 'size_first'  elements.
   'second' is a bit array of at least 'size_second' elements. 'result'
   points to enough space to hold the as many elements as the largest
   of 'size_first' and 'size_second'; OR 'result' points to NULL and
   such space is to be dynamically allocated.
   POST: TRUE has been returned and 'result' points to a bit array
   containing the union of the two arrays (up to the size of the largest
   of the two sizes); OR FALSE has been returned and 'result' pointed
   to NULL (a request was made to allocate enough memory to store the
   union) but the required memory could not be obtained.
   NOTE: This runs faster if the 'first' array is not smaller than
   'second'.
 */
```

**ba_diff**

Declaration:

```
bool ba_diff(bit first[], bit second[], bit * result[],
             const elem_t size_first, const elem_t size_second);

/*
   PRE:  'first'  is a bit array of at least 'size_first'  elements.
   'second' is a bit array of at least 'size_second' elements.  'diff'
   points to enough space to hold the as many elements as the largest
   of 'size_first' and 'size_second';  OR 'diff' points to NULL and
   such space is to be dynamically allocated.
   POST: TRUE has been returned and 'diff' points to a bit array
   containing the union of the two arrays (up to the size of the largest
   of the two sizes); OR  FALSE has been returned and 'result' pointed
   to NULL (a request was made to allocate enough memory to store the
   result) but the required memory  could not be obtained.
   NOTE: This runs faster if the 'first' array is not smaller than
   'second'.
 */
```

**ba_complement**

Declaration:

```
void ba_complement(bit arr[], const elem_t lsize);

/*
   PRE:  'arr' is a bit array composed of *exactly* 'size'
   elements.
   POST: All the bits in 'arr' have been flipped and 'arr' is
   in canonical form.
   SEE ALSO: ba_toggle()
*/
```

**ba_dotprod**

Declaration:

```
Declaration:
unsigned long ba_dotprod(const bit first[], const bit second[],
                         const elem_t size_first,
                         const elem_t size_second);

/*
   PRE: 'first' is an array of at least 'size_first' bits.
   'second' is an array of at least 'size_second' bits.
   POST: The scalar product of the two vectors represented by the
   first 'size_first' elements of 'first' and the first 'size_second'
   elements of 'second' have been returned.
*/
```

### 4.3.2 constraints.h

#### Functions Overview

- `eq_update();`
- `le_update();`
- `lt_update();`
- `noattack_update();`
- `alldifferent_update();`
- `create_constraint_X_Y();`

#### Structs & Variables

```
typedef struct {
  ConstraintDef *constr;
  int env[6];
}  Constraint;

typedef struct {
  char* name;
  int (*update)();
  int nargs;
} ConstraintDef;

ConstraintDef *constraintDefs;
```

**eq_update**

Declaration:

```
int eq_update(int env[], int nargs, Store* s, int i);

/*
  X = Y
  Variable (i) changed, update (store) according to constraint (=)

  PRE:
  'env' is the constraint environment;  'nargs' is the size of the
  environment; 'store' is the store that holds the values; 'i' the
  changed variable
  POST: store with updated values.
  NOTE: The bittarr.h intersection() function is used.
  USED BY: updateAll()
*/
```

**le_update**

Declaration:

```
int le_update(int env[], int nargs, Store* s, int i);

/*
  X <= Y
  Variable (i) changed, update (store) according to constraint (<=)

  PRE:
  'env' is the constraint environment;  'nargs' is the size of the
  environment; 'store' is the store that holds the values; 'i' the
  changed variable
  POST: store with updated values.
  USED BY: updateAll()
*/
```

**lt_update**

Declaration:

```
  int lt_update(int env[], int nargs, Store* s, int i);

  /*
    X < Y
    Variable (i) changed, update (store) according to constraint (<)

    PRE:
    'env' is the constraint environment;  'nargs' is the size of the
    environment; 'store' is the store that holds the values; 'i' the
    changed variable
    POST: store with updated values.
    USED BY: updateAll()
  */
```

**noattack_update**

Declaration:

```
int noattack_update(int env[], int nargs, Store* s, int i);

  /*
    X no attack Y (Queens example Constraint)
    Variable (i) changed, update (store) according to constraint

    PRE:
    'env' is the constraint environment;  'nargs' is the size of the
    environment; 'store' is the store that holds the values; 'i' the
    changed variable
    POST: store with updated values.
    USED BY: updateAll()
  */
```

**alldifferent_update**

Declaration:

```
int alldifferent_update(int env[], int nargs, Store* s, int i);

  /*
```

```
    X != Y (n-Fractions example Constraint)
    Variable (i) changed, update (store) according to constraint

    PRE:
    'env' is the constraint environment;  'nargs' is the size of the
    environment; 'store' is the store that holds the values; 'i' the
    changed variable
    POST: store with updated values.
    USED BY: updateAll()
  */
```

**create_constraint_X_Y()**

Declaration:

```
create_constraint_X_Y(int c_idx, char* name,
                      int X_pos, int Y_pos, void* func);

  /*
    Creates a X<->Y Cconstraint (e.g. X = Y, X no attack Y, ...)

    PRE:
    c_idx is the constraint definition identifier
    name is the constraint name
    X_pos defines of X is left or right side of the constraint
    Y_pos same as for X_pos
    func is the update function pointer
    POST: store with updated values.
    USED BY: constraint application
  */
```

### 4.3.3 fdd_value.h

**Functions Overview**

- `eq_update();`
- `le_update();`
- `lt_update();`

- ground();
- empty(v);
- equal();
- nth();
- first();
- last();
- next();
- get();
- cardinality();
- clear_fromto();
- clear();
- set();
- copy_new();
- new_value_single();
- new_value();
- invalid();
- printValue();
- new_IntArray();

## Structs & Variables

```
typedef struct {
  size_y size;
  bit* vector;
} BitVector;

typedef struct {
  size_y size;
  int* arr;
} IntArray;
```

```
typedef struct {
  int min;
  int max;
  BitVector bv;
} fdd_value;

typedef struct {
  int min;
  int max;
  int size;
} pm2_fdd_value;
```

**ground**

Declaration:

```
bool ground(fdd_value* v);

A value is ground if it contains a single element.
Returns true if ground, false otherwise.
```

**empty**

Declaration:

```
bool empty(fdd_value* v);

A value is empty if it contains no elements.
Returns true if empty, false otherwise.
```

**equal**

Declaration:

```
bool equal(fdd_value* v1, fdd_value* v2);

Two values are considered equal if they contain the exact same elements.
Returns true if equal, false otherwise.
```

**nth**

Declaration:

```
int nth(fdd_value*, int x); // nth element

Returns the values nth element.
```

**first**

Declaration:

```
int first(fdd_value* v);    // first element

Returns the values first element
```

**last**

Declaration:

```
int last(fdd_value* v);     // last element

Returns the value's last element
```

**next**

Declaration:

```
int next(fdd_value* v, int e);  // next element

Returns the values element sequentially after e.
Returns -1 if e is the last element.
```

**get**

Declaration:

```
bool get(fdd_value* v, int i);

Checks if the i element position is set.
Returns true if i is set, false otherwise.
```

**cardinality**

Declaration:

```
int cardinality(fdd_value* v);

Returns the cardinality (number of values) of the fdd_value
```

**clear_fromto**

Declaration:

```
void clear_fromto(fdd_value* v, int i, int k);

Clears (unsets) all elements in the value from position i to k
(inclusive).
```

**clear**

Declaration:

```
void clear(fdd_value* v, int i); // clear bitset value

Clears (unsets) the values element position i.
```

**set**

Declaration:

```
void set(fdd_value* v, int e);  // set bitset value

Sets the values element position e.
```

**copy_new**

Declaration:

```
fdd_value* copy_new(fdd_value* v);

Clones a value.
```

### new_value_single

Declaration:

```
fdd_value* new_value_single(int e);
```

```
Creates a singleton value.
```

### new_value

```
fdd_value* new_value(int min, int max);
```

```
Creates a new value.
```

### invalid

```
bool invalid(fdd_value* v);
```

```
A value is considered invalid if its maximum is equal to its minimum
both equal to -1.
```

### printValue

```
void printValue(fdd_value* v);
```

```
Prints  a value on screen.
Uses ba_b2str().
```

### new_IntArray

```
IntArray* new_IntArray(int size);
```

```
Creates a new IntArray structure array.
```

### 4.3.4 problem.h

#### Functions Overview

- `updateAll();`
- `new_problem();`
- `add_constraint();`
- `add_Cv();`

#### Structs & Variables

```
typedef struct {
  Store* sInit;          // initial store
  int nC;                // # constraints
  Constraint*  C;        // list of constraints
  int nCv;               // size of store
  Constraint*** Cv;      // list of constraints per variable
} Problem;
```

#### updateAll

Declaration:

```
int updateAll(Problem* P, Store* s, int i);

Function called to start each propagation run. Updates the stores
values according to every constraint.
```

#### new_problem

Declaration:

```
Problem* new_problem(Store* s, int n_constraints);

Create, allocate memory for, a new Problem. The problem size is
established by the size of its store and by the number of involved
constraints.
Returns a pointer to a Problem.
```

#### add_constraint

Declaration:

```
void add_constraint(Problem* P, int var, Constraint* c,
                    IntArray* cVx);
```

```
Adds a Constraint to the Problem, i.e., includes the constraint
pointer to the constraints list. Fills up the Constraints list with
the constraints environment.
```

#### add_Cv

Declaration:

```
void add_Cv(Problem* P, int var, Constraint* c);
```

```
Adds/Associates the constraint given by "c" to the variable given by "  var"
```

### 4.3.5 search.h

#### Functions Overview

- `backtrack();`
- `new_search();`
- `search_solution();`
- `search_solution_no_backtrack();`

#### Structs & Variables

```
typedef struct {
  int index;     // current value[index]
  int solutions; // number of solutions
} Search;
```

**backtrack**

Declaration:

```
void backtrack(Strategy* St, Search* Sr);

Backtracks for more solutions, i.e., gets back up on the search
tree (to the ancestor node) and tries a different element (index).

Special Note: The backtrack feature was implemented as in the
original AJACS model but can be considered an optional feature
since AJACS/C may work without its implementation.

For not using backtracing mechanism AJACS/C proposes the alternative
function: search_solution_no_backtrack()
```

**new_search**

Declaration:

```
Search* new_search(int index);

Creates and initializes a new Search.
```

**search_solution**

Declaration:

```
void search_solution(Problem* P, Strategy* St, Search* Sr);

Find the Problem solutions. Takes the Problem P, implements the
Strategy St.
```

**search_solution**$_{n}o_{b}$*acktrack*

Declaration:

```
void search_solution(Problem* P, Strategy* St, Search* Sr);

Find the Problem solutions without using backtracking.
Takes the Problem P, implements the Strategy St.

Note: Uses a stack to hold the stores as they are evaluated
```

### 4.3.6 store.h

#### Functions Overview

- `nvars();`
- `getValue();`
- `setValue();`
- `nextStore();`
- `printStore();`
- `printStore2();`
- `new_store();`
- `copyStore();`
- `copyStore2();`
- `store_2_pm2Store();`
- `pm2Store_2_store();`

#### Structs & Variables

```
typedef struct Store Store;

struct Store {
  Store* ancestor;         // ancestor store (state)
  int var;                 // active var
  int nvars;               // number of variables (lines)
  fdd_value *theValues[0]; // the values
};

typedef struct {
  int var;                     // active var
  int nvars;                   // number of variables (lines)
  pm2_fdd_value theValues[12]; // the values
} pm2_Store;
```

**nvars**

Declaration:

```
int nvars(Store* s);
```

Returns the number of vars of Store s.

**getValue**

Declaration:

```
void getValue(Store* s, int i);
```

Gets a value from Store s at position i.

**setValue**

Declaration:

```
void setValue(Store* s, int k, fdd_value* v);
```

Sets a value in Store s at position k.

**nextStore**

Declaration:

```
Store* nextStore(Strategy* St);
```

Creates, and returns, a new Store cloned from the one hold by the Strategy St.

**printStore**

Declaration:

```
void printStore(Store* s);
```

Prints a store from a Store pointer on screen.

**printStore2**

Declaration:

```
void printStore2(Store s);
```

```
Prints a store from a Store variable on screen.
```

**new_store**

Declaration:

```
Store* new_store(int size);
```

```
Creates a new Store pointer.
```

**copyStore**

Declaration:

```
Store* copyStore(Store* old_s);
```

```
Clones and returns a Store pointer from old_s Store pointer.
```

**copyStore2**

Declaration:

```
Store* copyStore2(Store old_s);
```

```
Clones and returns a Store variable from old_s Store variable.
```

**store_2_pm2Store**

Declaration:

```
pm2_Store store_2_pm2Store(Store* s);
```

```
Transforms an AJACS/C Store into a PM2 Store.
Note: A PM2 Store is an fdd_value, i.e., with bitset different from bi  tarr.
```

**pm2Store_2_store**

Declaration:

```
Store* pm2Store_2_store(pm2_Store pm2_s);
```

```
Transforms a PM2 Store into an AJACS/C Store.
Note: A PM2 Store is an fdd_values, i.e., with bitset different from b  itarr.
```

### 4.3.7 strategy.h

**Functions Overview**

- `selectVar();`
- `selectValue();`
- `new_strategy();`

**Structs & Variables**

```
typedef struct {
  Store* store;        // initial store
} Strategy;
```

**selectVar**

Declaration:

```
int selectVar(Store* s);
```

```
Selects the next non-ground variable to initiate the next split and
propagation run.
```

**selectValue**

Declaration:

```
fdd_value* selectValue(int var, Store* s, int index);
```

```
Selects and returns the stores value corresponding to the index nth
position in the form of a singleton value. This value represents the
next basic option for propagation and a possible next solution.
```

**new**$_s$*trategy*

Declaration:

```
Strategy* new_strategy(Store* s);
```

```
Creates and returns a new Strategy. In the beginning the Strategy
holds the Store s.
```

# Part IV

# Examples and Results: Distributed AJACS/C with DSM-PM2 benchmarking

# Chapter 5

# Examples and Interpretation of Results

The PM2 (Parallel Multi-Threaded Machine)[4], introduced on early chapters, is a low level generic runtime system which integrates multithreading management (Marcel) and a high performance multi-cluster communication library Madeleine [6].

PM2 incorporates a DSM module (DSM-PM2) [1] that claims to be ready to provide the developer the ability to build programs that take full advantage of the DSM concept.

The design and final use of the DSM library is highly dependent on the selected consistency model. For this DSM-PM2 offers the possibility to use four different built-in consistency models:

- LI_HUDAK, a sequential consistency protocol;
- Migrate Thread, a peculiar protocol in which threads are moved when they need some data that is outside its node scope;
- ERC, an eager release consistency protocol;
- HBRC [47], an Home Based Release Consistency protocol.

The main objective of this chapter is to integrate and experiment AJACS/C with DSM-PM2 with a set of chosen examples. Through this next chapter we explain how DSM-PM2 is configured, what will our cluster architecture be like and finally a set of designed examples for experimentation are introduced, its sampling figures presented as for interpretation of this results.

## 5.1 PM2 Configuration

The PM2 library [4] is still on its early beta releases. The release used for this thesis study was the: pm2-2005-03-16 [1] - "pm2-2005-03-16.tar.gz".

The PM2 library allows a wide range of configuration aspects to be set up. Among others the user is allowed to:

- Choose and configure its favorite PM2 communication library from the ones available: these are the **Madeleine** libraries (currently between an experimentation range from *mad1, mad2, mad3 and mad4*;
- Configure the thread handling library - this is the marcel library;
- Configure the DSM library;
- Configure the network characteristics (TCP, UDP, VIA [45], others ...);
- Configure the debug information;
- Configure other basic pm2 characteristics.

The PM2 setup used for running the examples was:
(as result of pm2-config-flavor –text — option 4) — flavor pm2)

```
Module pm2    with options: opt build_static
Module dsm    with options: opt build_static
Module marcel with options: opt build_static mono
                            smp_shared_queue marcel_main
Module mad3   with options: opt build_static tcp
Module tbx    with options: opt build_static
Module ntbx   with options: opt build_static
Module init   with options: opt build_static
```

With "opt" meaning "optimized". To note that we configured the **madeleine 3** [6] library, the one that comes by default on this PM2 release. To note also that we configured **TCP** for network protocol transport layer. **UDP**, as an alternative, could not be set up on this PM2 release.

## 5.2 Cluster Configuration

Our cluster configuration (see figure 5.1) is composed of 4 Intel Pentium 4 CPU 2.80GHz nodes with 512Kb cache and 256MB RAM interconnected through Fast-Ethernet.

[1]Download at: http://gforge.inria.fr/projects/pm2/

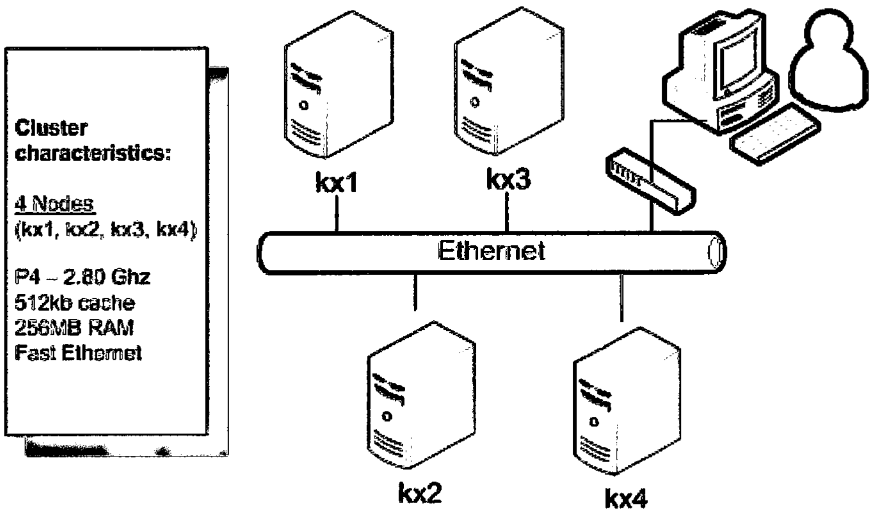


Figure 5.1: Experimenting Cluster Configuration

To configure PM2 with the chosen cluster setup the user just needs to invoke the **pm2conf** command. As an example, running the command: *pm2conf kx1 kx2 kx3 kx4* will setup all the cluster available machines to run the DSM-PM2 example.

## 5.3 How to create a DSM-PM2 AJACS/C Example

When creating an example for this thesis study the programmer is in fact creating a DSM-PM2 application artifact that makes use of the AJACS/C library API (detailed in section 4.3).

For doing this the programmer first creates a PM2 example application, using the PM2 library (including the DSM-PM2 features). Next picks from one of the two distribution models described in section 4.2. Finally he writes the DSM-PM2 AJACS/C application using the AJACS/C API, just making sure to link the AJACS/C library on the application build process.

Figure 5.2 illustrates this process:

To run a DSM-PM2 AJACS/C application the user just needs to run the program as other regular PM2 example, by invoking the **pm2load** command.

The compiler used to build the example applications was the GNU Compiler v3.2.3

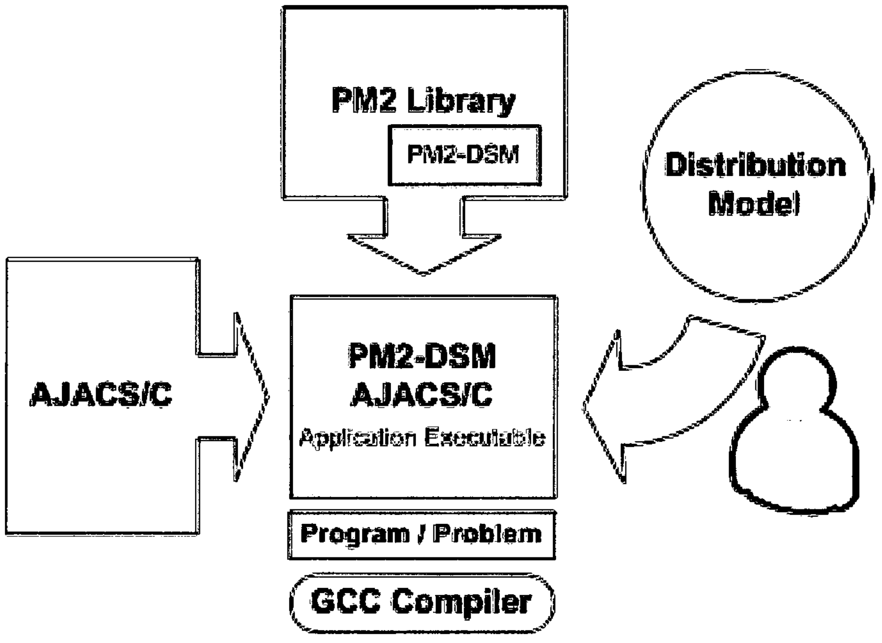


Figure 5.2: Building a DSM-PM2 AJACS/C application

## 5.4 Queens Example

### 5.4.1 Specification

The queens puzzle is the problem of putting 'n' chess queens on an NxN chessboard such that none of them is able to capture any other using the standard chess queen's moves. The colour of the queens is meaningless in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

### 5.4.2 Problem Description

On this problem one "queen" position on the chess board maps directly to one **Variable**, so the **Store** structure will hold as many variables as the number of queens specified.

As for **Constraints**, and as for specification, the relation between the queens (variables) will be such that on a given solution no queen can "attack" another queen. The constraint "noattack" is then designed as relation between two queens. In a given problem there will exist the following number of "noattack" constraints:

```
constr(0) = 0
constr(1) = 1
constr(n) = constr(n-1) + (n-1)
```

For the specific example of four Queens there will exist 6 corresponding constraints (figure 5.3). Note: The *noattack* constraint is bi-directional, e.g. for constraint **1 no attack 2** Queen 1 does not attack 2 and neither Queen 2 attacks 1. What happens is that during propagation (where one variable domain is reduced), all the constraints where that variable was defined are activated that is why does not matter the direction of the constraint and so there is no need to define the inverse constraint, that is **2 noattack 1** is unnecessary.

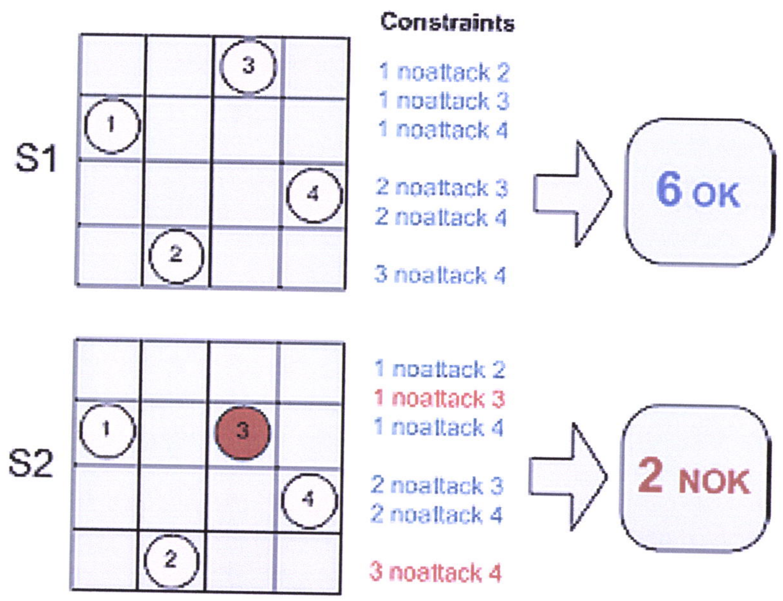


Figure 5.3: Queens example

In the example of figure 5.3, the Store **S1** succeeds all the constraints of the problem which means it may be considered a valid store to split and continue the search. In this trivial specific case **S1** is at the same time a valid propagated store and a solution. **S2** violates two constraints and as so will be discarded from the problem space.

**Search**, in AJACS/C PM2 and for the Queens example, will be the successive handling of available stores (spawing from initial store) using a **DFS** (Depth-first search) / **LIFO** approach. After the split of some store the child stores are included at the top of the stores stack. Note: LIFO was the chosen approach but FIFO / Breadth-First search could also be used. LIFO was prefered due to the fact that we are doing, in any case, full tree traversing, i.e. looking for all possible solutions for a problem. LIFO is better suited as we may be insterested in searching for a first solution quickly.

In each search step the adopted **Strategy** is to look for the first non-ground variable available on current store (using a top-down look-up). That variable is chosen to be the next to be iterated and reduced, that is, all singleton values of that variable are successevily tested, triggering propagation on the rest of the store. If propagation is deamed successful the resulting store will be added to the search list for further search steps.

The search & strategy steps are repeated until there are no more stores available to iterate on.

In the case of the Queens example, the **Problem** solutions will be the collection of stores that contain only ground (singleton) values, so where all variables are instatiated with single values. These remaning stores are the ones that survived all the iterations and propagations and so that satisfy the "noattack" constraints, where in the end no queen can attack another.

### 5.4.3 Interpretation of Results

The elapsed time, of the examples execution run-time, is measured on the problem inner search computation, right after the distributed program starts performing its search function until there is no more problem space to evaluate and all found solutions are shown to the user. All PM2 and Problem application initialization, including PM2 cluster setup overhead and Problem initialization setup, times are excluded from the timming results below.

When running the distributed application time sampling is organized in the following:

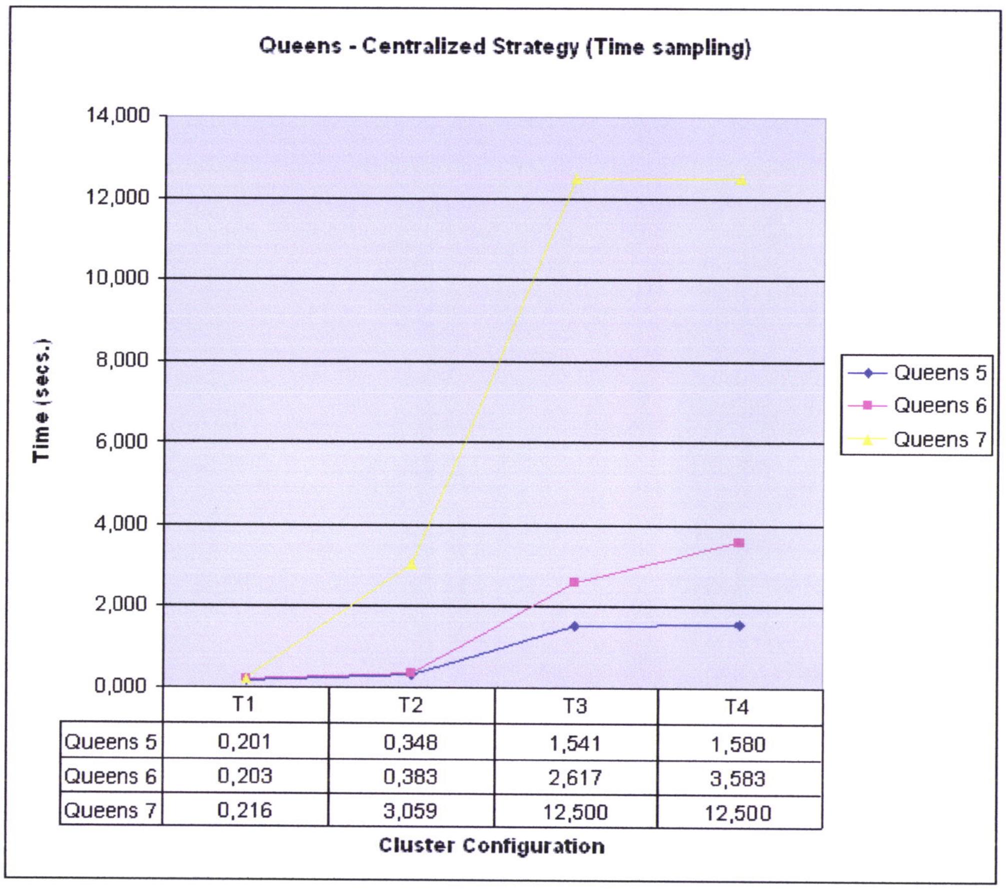
- T1: One-node cluster configuration {kx1} time
- T2: Two-node cluster configuration {kx1, kx2} time
- T3: Three-node cluster configuration {kx1, kx2, kx3} time
- T4: Four-node cluster configuration {kx1, kx2, kx3, kx4} time

Speedup is defined as the execution run-time gain ratio of some cluster configuration in relation to the trivial configuration: single node execution.

Speedup = T1 / Ti;

#### Centralized Distributed Pattern

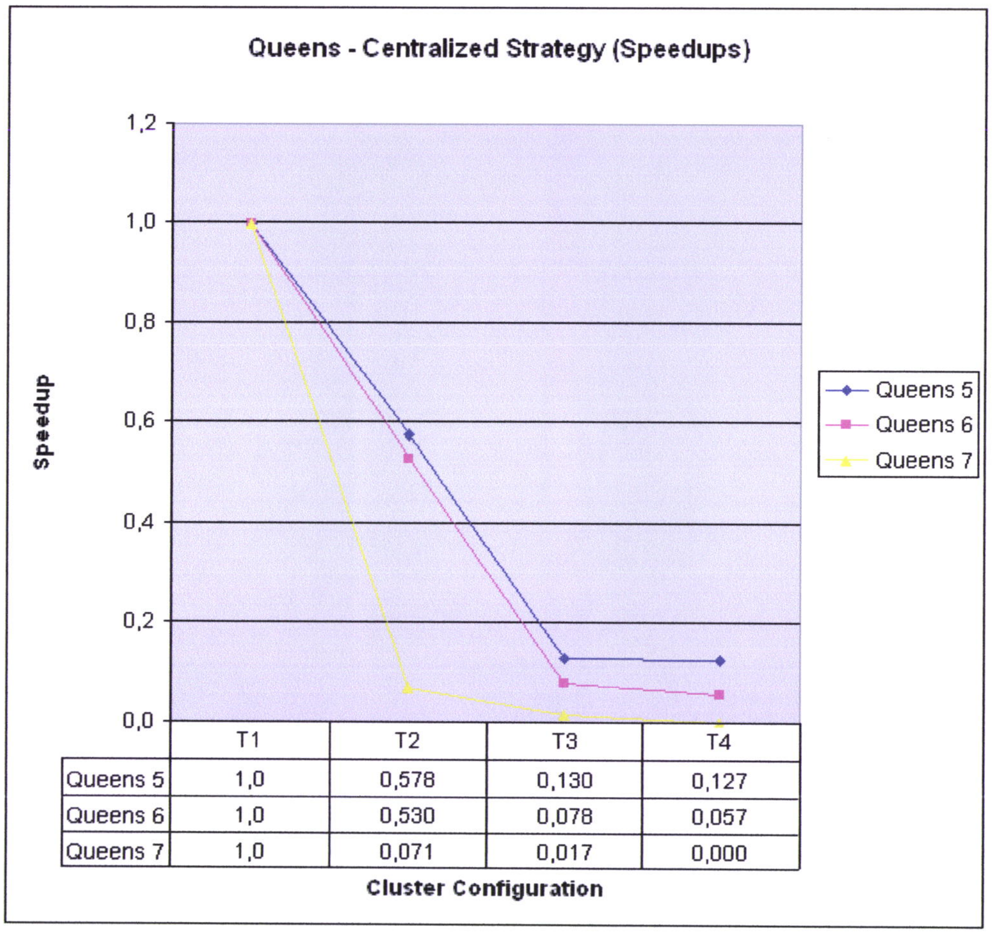
Queens 5, 6 and 7 were tested using the Centralized distribution strategy.



| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| Queens 5 | 0,201 | 0,348 | 1,541 | 1,580 |
| Queens 6 | 0,203 | 0,383 | 2,617 | 3,583 |
| Queens 7 | 0,216 | 3,059 | 12,500 | 12,500 |

Figure 5.4: Queens Centralized strategy time results

As the results show (refer to figure 5.4) the Queens example runs slower, taking more time to find the problem solutions, when more nodes are made available for the distributed computation. Queens 8, 9, 10 and 11 examples are not shown in this graphical example results but they proved to consume exponential execution run-time. If we compare the Queens 7 example from the Centralized and Local patterns the results are conclusively different as with a full configured cluster (all 4 nodes) the centralized pattern took around 12,5 seconds to finish against the 0.485 seconds of the local pattern.



| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| Queens 5 | 1,0 | 0,578 | 0,130 | 0,127 |
| Queens 6 | 1,0 | 0,530 | 0,078 | 0,057 |
| Queens 7 | 1,0 | 0,071 | 0,017 | 0,000 |

Figure 5.5: Queens Centralized strategy speedup results

As expected, from the obtained run-time results, speedups are negative using the centralized distribution strategy for the Queens example (refer to figure 5.5).

A major DSM-PM2 sychronization overhead may be a possible explanation for the poor results of the centralized model as the single distributed DSM data structure, that is shared by all nodes, may be the cause for such unefficient results.

Internal debug testing demonstrated that the control access of the different nodes to the centralized DSM data structure proved to be extremely time consuming which raises the possibility of a DSM synchronization / coherence protocol bottleneck.

As a preliminary conclusion the Centralized Distribution pattern seams **not to be** an efficient approach for AJACS/C Constraint Programming distribution using DSM-PM2.

### Local Distributed Pattern

The Queens 5 to 11 examples were tested using the Local Distribution strategy.

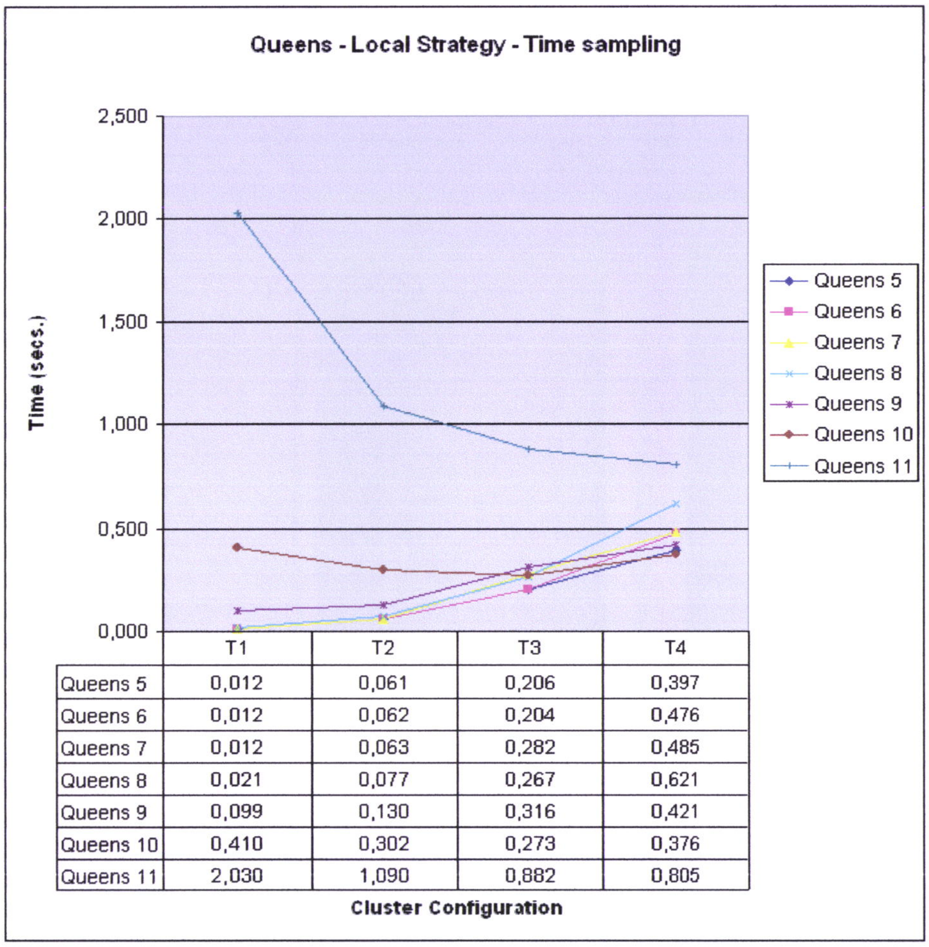
As the results show (see figure 5.6) only examples Q10 and Q11 prove to be efficient when using several nodes for its computation. Examples Q5 to Q9 do not benefit from using more than one node on its computation and even take more time to finish.

Speedups are then only achieved on examples Q10 and Q11 with the later being the only example where linear speedups are achieved (see figure 5.7).

To note on this local distribution pattern that in spite Q5 to Q9 aparent lack of performance when using more nodes for the example computation, the obtained results are not exponentially bad as in the centralized distribution pattern, in fact they are very similar (from T1 to T4).

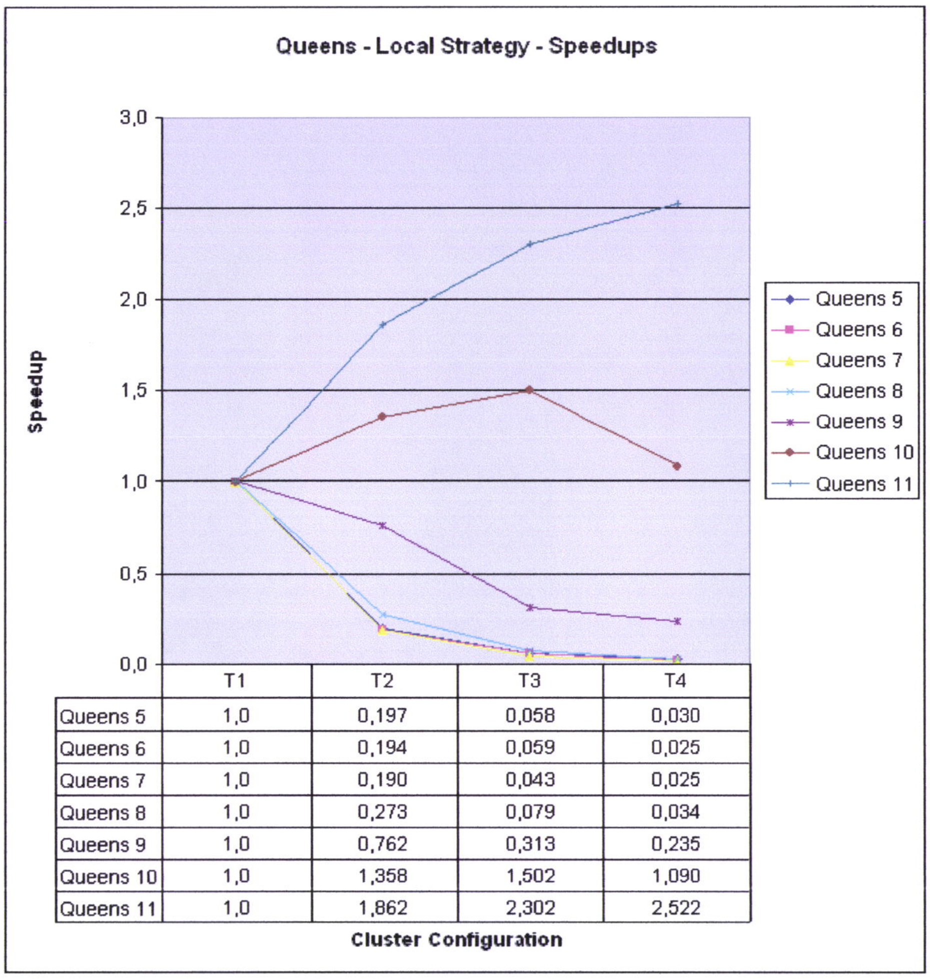
If we add to this observation the fact that Q10 starts to have performance improvement, judging at least for the cluster configuration T2 and T3 speedups, and that Q11 proved to be very efficient when using all 4 nodes we may conclude that the Queens distributed example starts to be efficient upon certain point threshold. These observations suggest that when the computational workload starts to be considerably representative the distributed examples start to perform well and speedups became possible. The apparent lack of performance for cases Q5 to Q9 suggest that these examples were too sparse and with lack of computational representativity to be considered valid examples.

From the observations above, and as preliminary conclusion, the Local Distribution Pattern seems **to be** adequate to the DSM-PM2 AJACS/C implementation.



| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| Queens 5 | 0,012 | 0,061 | 0,206 | 0,397 |
| Queens 6 | 0,012 | 0,062 | 0,204 | 0,476 |
| Queens 7 | 0,012 | 0,063 | 0,282 | 0,485 |
| Queens 8 | 0,021 | 0,077 | 0,267 | 0,621 |
| Queens 9 | 0,099 | 0,130 | 0,316 | 0,421 |
| Queens 10 | 0,410 | 0,302 | 0,273 | 0,376 |
| Queens 11 | 2,030 | 1,090 | 0,882 | 0,805 |

Figure 5.6: Queens Local strategy time results



| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| Queens 5 | 1,0 | 0,197 | 0,058 | 0,030 |
| Queens 6 | 1,0 | 0,194 | 0,059 | 0,025 |
| Queens 7 | 1,0 | 0,190 | 0,043 | 0,025 |
| Queens 8 | 1,0 | 0,273 | 0,079 | 0,034 |
| Queens 9 | 1,0 | 0,762 | 0,313 | 0,235 |
| Queens 10 | 1,0 | 1,358 | 1,502 | 1,090 |
| Queens 11 | 1,0 | 1,862 | 2,302 | 2,522 |

Figure 5.7: Queens Local strategy speedup results

### 5.4.4 Conclusions

The **Queens** example show that speedups are possible using the Local Distribution strategy. The example proved not to be suited for the centralized strategy where no meaningful results were achieved, or better no speedup was found.

One probable reason for this example inadequacy for the centralized strategy is that the central DSM structure synchronization and DSM-PM2 communication overhead masks any potential distribution gain. By constantly interacting with the DSM centralized structure for getting and storing "jobs" the nodes force a huge (constant) synchronization of this DSM structure. Any gain obtained by distributing "work" between nodes is quickly absorbed by the inherent synchronization and communication overhead.

The lack of visibility on how the DSM data structure is partitioned among the different nodes (by DSM-PM2) and which parts of this structure are in fact invalidated when new stores are written to this centralized structure was a major difficulty and incertainty factor on the experimentation.

By the contrary the example is found to be adequate for the Local distribution pattern strategy as the search space computation was splitted across nodes on a somewhat efficient way, assuring that the computational representativity of the example is sufficiently high enough speedups are possible and mask any DSM-PM2 syncronization and communication overhead.

## 5.5 n-Fractions Example

### 5.5.1 Specification

#### Original Specification

The original n-Fractions puzzle [42] is specified as follows. Find 9 distinct non-zero digits that satisfy:

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1$$

where BC is shorthand for 10B+C, EF for 10E+F and HI for 10H+I.

#### n-Fractions Specification

A simple generalization is as follows. Find 3n non-zero digits satisfying:

$$\sum_{i=1...n} \frac{x_i}{y_i z_i} = 1$$

where:

$$y_i z_i$$

is shorthand for:

$$10y_i + z_i$$

and the number of occurrences of each digit in 1..9 is between 1 and *ceil(n/3)*. Since each fraction is at most 1/99, this family of problems has solutions for at most n $\leq$ 99. An interesting problem would be to find the greatest n such that at least one solution exists. A further generalisation might specify that the fractions sum to *ceil(n/3)*.

### 5.5.2 Problem Description

On this problem one letter of the n-fractions correspond directly to one **Variable**, so the **Store** structure will hold as many variables as **n_fract * 3** the number of fractions times 3 (the number of variables in each fraction).

As for **Constraints**, and as for specification, the only given relation between the variables, and candidate to be a constraint, is the fact that the occurrences of each digit in [1..9] is between 1 and ceil(n/3), specifically for n=2 and n=3 no variable is allowed to have repeated values from other variables. The constraint "alldifferent" is then allowed as relation between two variables (see figure 5.8). In a given problem there will exist the following number of "alldifferent" constraints:

```
constr(0) = 0
constr(1) = 1
constr(n) = constr(n-1) + n-1
```

From the example illustrated in 5.8 one can conclude that the fact of the resulting store is ground, which means all its variables are ground (with a single value), does not necessarily mean that the store is a solution of the n-Fractions problem, against what happened for the Queens example, where all the resulting ground stores would necessarily be problem solutions. An extra step, after successful propagation of a potential store will be necessary. The store must suffer a final validation, in this case the n-Fraction specification rule itself (see Problem Specification above 5.5.1).

Search / Strategy will be the same as for the Queens example. In the case of the n-Fractions example, the **Problem** solutions will be, as described

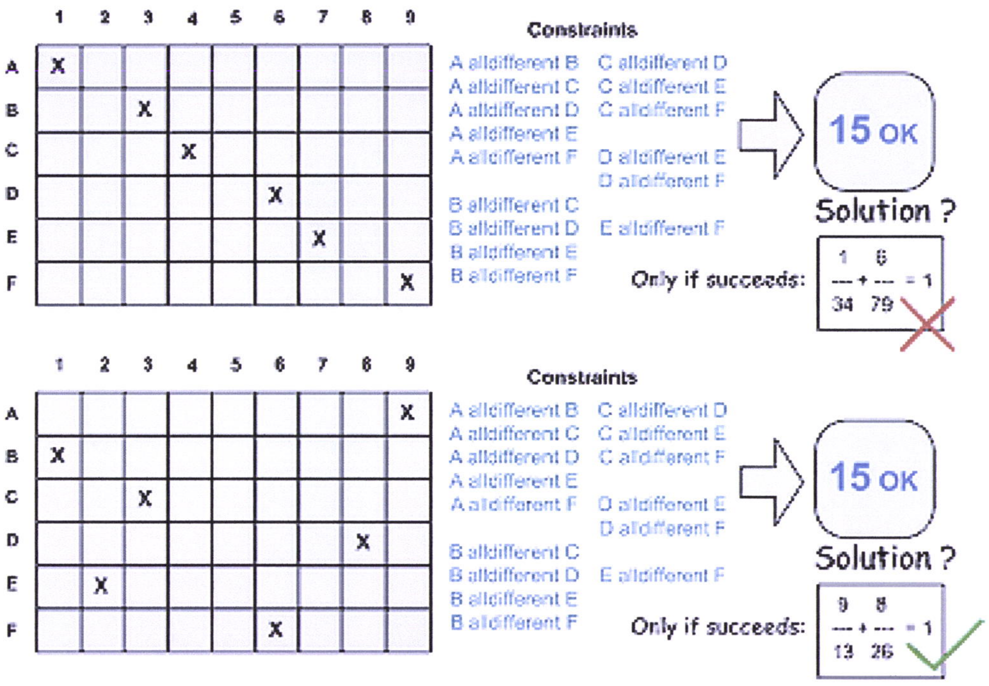


Figure 5.8: n-Fraction (n=2) example

above, the collection of stores that contain only ground (singleton) values and that additionaly satisfy the main problem "contraint", that all fractions sum up exactly 1.

### 5.5.3 Interpretation of Results

#### Centralized Distributed Pattern

Tests were performed using this strategy for the n-Fractions example but no meaningful results were obtained. As for the Queens example all tests took more run-time when using more than one node for the program execution and as so no speedups were achieved for n-Fractions example using the Centralized distribution strategy.

The same interpretation of results and preliminary conclusions on the centralized distribution model obtained from the Queens example should apply also to the n-Fraction example.

#### Local Distributed Pattern

The n-Fractions (n=2) example was tested using the Local distribution pattern strategy.

The results show (figure 5.9) that the application proved to be more run-time efficient when using more nodes on its execution.

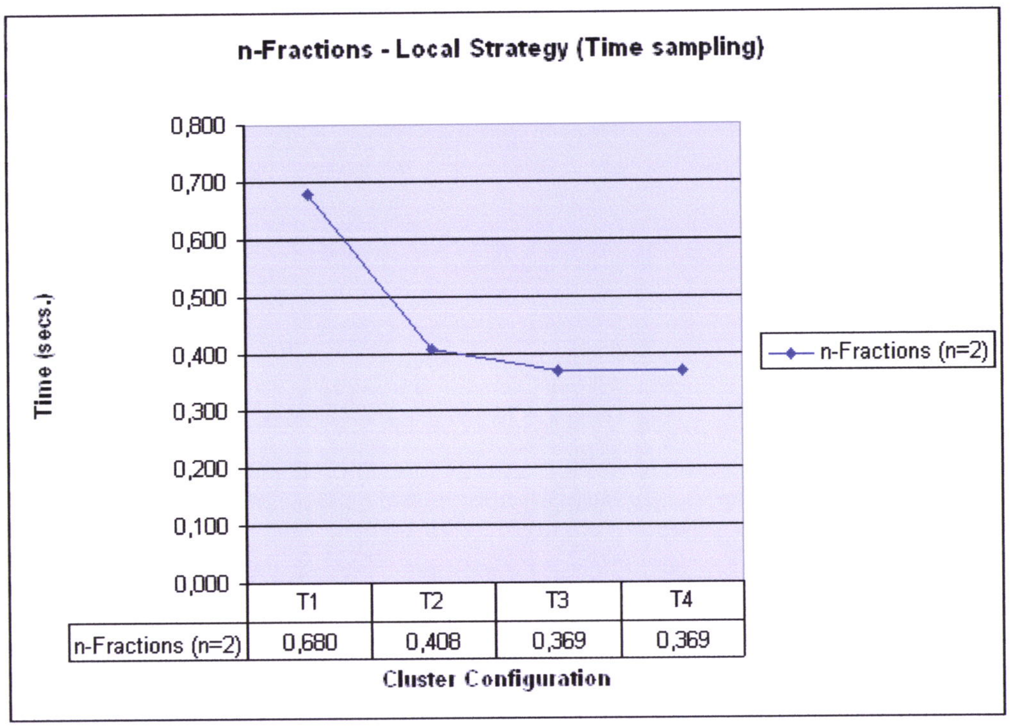
Significant speedups were achieved using this strategy (see to figure 5.10).

### 5.5.4 Conclusions

The n-Fractions example show, as for the queens example, that speedups are possible using the Local Distribution strategy.

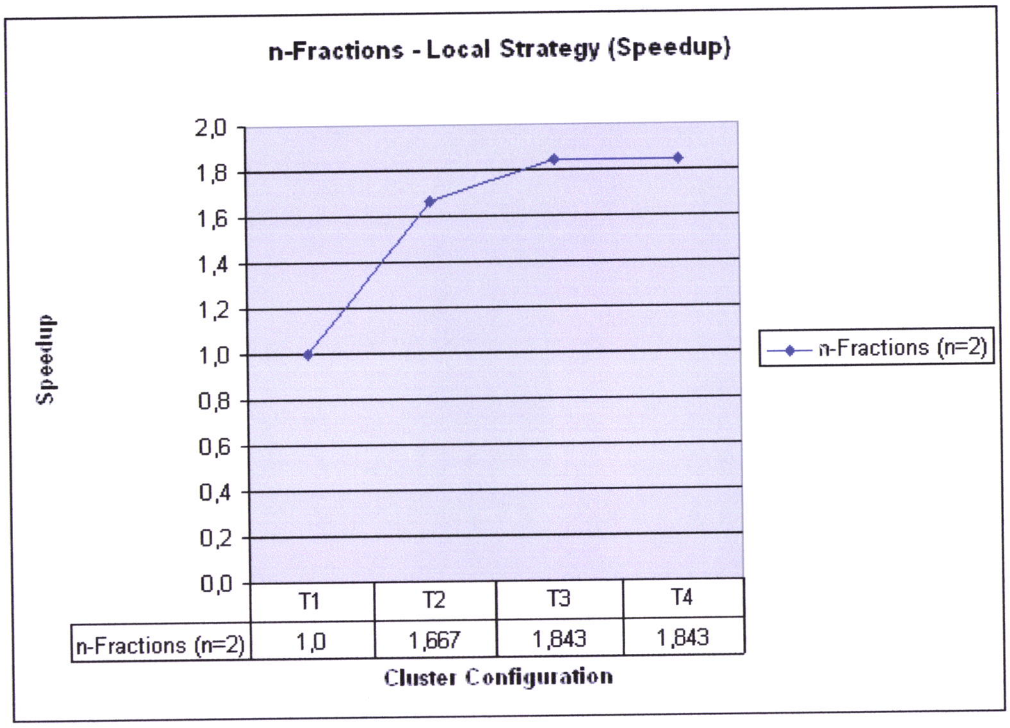
The example proved **not to be suited at all** for the centralized strategy where no meaningful results were achieved, or better no speedup was found possible. One probable reason for this example inadequacy for the centralized strategy is the fact that the constraint being applied does not reduce the problem space very efficiently or fast enough and as so the problem tends to generate and iterate on a huge number of candidate state solutions till very late on the problem. This huge number of candidate **stores** creates an heavy bottleneck on the DSM structure synchronization and destroys any hope of obtaining speedups.

By the contrary the example is found to be **well suited** for the Local distribution strategy as this heavy computation (due to the huge number of search space) can be distributed across nodes on a very efficient way, efficient enough to obtain speedups over DSM-PM2 syncronization and communication overhead.



| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| n-Fractions (n=2) | 0,680 | 0,408 | 0,369 | 0,369 |

Figure 5.9: n-Fractions (n=2) Local strategy time results



|  | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| n-Fractions (n=2) | 1,0 | 1,667 | 1,843 | 1,843 |

Figure 5.10: n-Fractions (n=2) Local strategy speedup results

# Part V

# Conclusion and Future: AJACS/C with DSM-PM2

# Chapter 6

# Conclusions and Future Work

This chapter will present this thesis study conclusions to the reader as for certainties and possibilities for future work.

## 6.1 Work performed on this thesis

One of the accomplished objectives of this thesis was the development of a Constraint Solver in the C language and its experimentation on a distributed environment. This Constraint Solver is **AJACS/C**. The distribution environment consisted of a total of 4 workstation nodes connected through Ethernet. The Memory architecture follows a CC-NUMA approach, described in section 2.1 of this thesis.

### 6.1.1 AJACS/C Development

AJACS/C takes its base from the AJACS system [3] and the main motivation was to port this system to the C language for a smooth integration with the elected Parallelization library for experimentation: **PM2** [4](Parallel Multi-Threaded Machine). This integration allowed the development of distributed PM2 applications using the AJACS/C engine. It is now possible to design and experiment distributed constraint problems using PM2 and AJACS/C.

### 6.1.2 DSM-PM2 AJACS/C Integration

Parallelization is achieved by using PM2 but PM2 offers even more. PM2 contains a special module called DSM-PM2 that offers DSM abstraction capabilities. With DSM-PM2 is now possible to develop distributed constraint problems using AJACS/C with real distributed shared memory abstraction to the programmer.

### 6.1.3 Distribution Models Design

For distributing a certain constraint problem, developed by AJACS/C, the programmer needs not only to design a DSM-PM2 program.

To allow a controlled distribution two distributed models were designed:

- The Centralized Distribution Model;
- The Local Distribution Model.

The **Centralized Distribution Model** implements a pure DSM system where all the information is stored on a single structure that is shared (made visible) to all the cluster nodes. Each node job will be to negociate with this centralied data structure for retrieving and adding jobs (problem stores). This model is simple to implement where only a single data structure is created (using the DSM-PM2 primitives). All nodes share the same parallel code and data structures and all will work until there are no more jobs to process.

The **Local Distribution Model** is less bold but more pragmatic than the Centralized model. In this model each node possesses one DSM data structure that only each can interact with. This gives complete independence to the nodes as they do not need to "communicate" with the other nodes during the parallel computation. The initial state is spawned and its childs distributed across the different nodes. From that point forward each node will live alone with its piece of workload.

### 6.1.4 AJACS/C with DSM-PM2 Experimentation

To experiment AJACS/C and DSM-PM2 with real representative cases two examples were implemented:

- The classic Queens example;
- The n-Fractions example.

The Queens example gives alot of choices to the experimentation campaign and it maps perfectly to the AJACS/C. Every variable is a Queen and the constraints space is reduced to only a single constraint: "noattack" where no queen is allowed to attack another. This problem is also very flexible as we are allowed to experiment Queens for N variables. In the experimentation campaign we used from Queens 5 to 11.

The Queens example was tested using both the Centralized and Local Distribution models. The experimentation conclusions can be found on chapter 5 and later on this chapter.

For the sake of experiment representativity another example was implemented: the n-Fraction example. This example implements the "alldifferent" constraint and was also tested using both distribution models. Conclusions for the n-Fraction example can also be found on chapter 5 and later on this chapter.

## 6.2 Conclusions

**AJACS/C** is now a newly available Constraint Solver. Fully developed in the C language it inherits the AJACS [3] main characteristics. AJACS/C main feature is the ability to perform constraint solving without the need to use the backtracking mechanism. Stores are spawned and stored, after a successful propagation, in some data structure (queue or stack). The programmer is allowed to search over the problem space using its favorite or costumized search technique which makes AJACS/C highly extensible and modifiable.

**AJACS/C was successfuly integrated with the PM2[4] library** (developed in C), specially with the DSM-PM2 module [1], that offers the distributed shared memory feature to PM2 and AJACS/C. With this integration made possible the programmer is now allowed to design AJACS/C Distributed Constraint Solving programs using the DSM-PM2 mechanism.

The AJACS/C DSM-PM2 integration was evaluated by experimenting two examples with two distinct distribution models. Judging from both examples, Queens and n-Fraction, experimentation campaigns, it is concluded that **DSM-PM2 is adequate for parallelizing AJACS/C programs using the DSM paradigm.** However the experimentation results indicate that the extent of this success, that can be measured by effective speedups, is **highly dependent on the distributed model choice.**

Distribution models that rely on centralized DSM structures tend to generate huge run-time overhead which seams to indicate that the DSM-PM2 does not handle well when is subject to high synchronization and node intercommunication need. The way how DSM-PM2 performs the DSM structure partitioning split through all the cluster nodes is not obvious nor trivial which made hard to know exactly which parts of the structure were associated to some node. This does not necessarily mean that DSM-PM2 is not correctly implemented nor that it may be still imature. The version used for experimentation was "pm2-2005-01-16". More recent versions were made available since then. Experimenting new versions of DSM-PM2 may be the subject of further work out of scope of this thesis.

On ther other hand the **DSM-PM2 behaves flawlessly, not surprisingly though, when a des-centralized model is used**. In this sort of models there is full parallelization of the problem space and little or no inter-node communication is expected so it is natural that they behave better.

This indicates that better results are to be expected, from AJACS/C DSM-PM2 integration, if the programmer decides to do full parallelization of the Constraint Problem, that is to spread the problem workload, through the different nodes beforehand.

AJACS/C DSM-PM2 integration is a reality with effective results, as **real speedups were obtained**, but the DSM-PM2 module did not deliver flawless results, at least with the distributed models that were implemented.

Further work is possible in order to explore and further assess this, or other DSM library adequacy to the Constraint Programming paradigm. The next chapter will present future work possibilities.

## 6.3 Future Work

- Evolve AJACS/C for more CSP examples: AJACS/C has still only two examples (if we exclude some more trivial cases): - Queens and n-Fraction. More examples will turn the AJACS/C Constraint Solver richer in both examples and features (as new types of problems and constraints would necessarily be developed).
- Continue benchmarking new releases of DSM-PM2 with AJACS/C: This thesis evaluated the "pm2-2005-03-16" release. New releases are available that may prove more efficient.
- Design and Experiment new distribution models: Two distribution models were developed: Centralized and Local Distribution Models. New models with additional features like heuristics and load-balancing techniques may improve even further the AJACS/C DSM-PM2 results. An "Hybrid" model that could take benefict from both models would be possibly the ideal scenario. As the centralized model originates huge overhead we could use a Local model instead but to avoid doing wrong distribution of work through the nodes, on tipical Local distribution, some feedback loop mechanism could be implemented to avoid the situation where some node(s) may be stopped and others still running with still alot of work to be perfomed. A possible title for this new mode could be: "Hybrid Model: Local Distribution Strategy with Load Balancing Feedback"
- A line of work is under way to allow mixing the single-Cell solver with other instances thereof is another line which are being followed:

there are dual-Cell blade systems which provide shared memory (albeit NUMA). These provide one first level of distribution outside a single Cell processor and represent a shared memory layer similar to the original AJACS organization: stores (or problems, as per AJACS terminology) may be shared among different processors. A further distribution layer can be obtained when we consider a network of such blades, falling back onto the AJACS/C model. In short, the port of AJACS to C, based on PM2-DSM, be it the one based on the Cell processor or otherwise, is undergoing active development and more signficant experimental results are expected soon. For details refer to paper titled: "Design for a Parallel and Distributed Hybrid Constraint Programming Library" [48].

# Bibliography

[1] Gabriel Antoniu and Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, volume 2026 of *Lect. Notes in Comp. Science*, pages 55–70, San Francisco, April 2001. Held in conjunction with IPDPS 2001. IEEE TCPP, Springer-Verlag.

[2] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27:1279–1297, October 2001.

[3] Lígia Ferreira and Salvador Abreu. Design for AJACS, yet another Java Constraint Programming framework. *Elsevier Electronic Notes in Theoretical Computer Science*, 48, 2001.

[4] Raymond Namyst and Jean-Franois Mhaut. PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.

[5] Peter Van Roy and Seif Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, November 1999. Part of International Conference on Logic Programming (ICLP 99).

[6] Olivier Aumage. Heterogeneous multi-cluster networking with the Madeleine III communication library. In *Proc. 16th Intl. Parallel and Distributed Processing Symposium, 11th Heterogeneous Computing Workshop (HCW 2002)*, Fort Lauderdale, April 2002. Held in conjunction with IPDPS 2002. 12 pages. Extended proceedings in electronic form only.

[7] K. Li. Ivy A shared virtual memory system for parallel computing. In *Proc. of the 1988 Intl Conf. on Parallel Processing (ICPP88), volume*

*II*, Fort Lauderdale, April 2002. Held in conjunction with IPDPS 2002. volume II, pages 94101, August 1988.

[8] S. Zhou, M. Stumm, K. Li, and D.Wortman. Heterogeneous distributed shared memory. In *IEEE Trans. on Parallel and Distributed Systems, 3(5):540554, September 1992.*

[9] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type–specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).* pp. 168–176, Mar. 1990.

[10] B. N. Bershad, M. J. Zekauskas and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conference.* p.528537, Feb 1993.

[11] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference.* pages 115131, January 1994.

[12] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. of the 21th Annual Intl Symp. on Computer Architecture (ISCA94).* pages 325337, April 1994.

[13] B. Fleisch and G. Popek Mirage: A Coherent Distributed Shared Memory Design

[14] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J.M. BernabeuAuban, P.W. Hutto, M.Y.A. Khalidi, and C.J.Wileknloh. The design and implementation of the Clouds distributed operating system. In *Computing Systems Journal, 3, Winter 1990.*

[15] G. Schoinas Issues on the Implementation of PrOgramming SYstem for DistriButed AppLications: A Free Linda Implementation for Unix Networks ftp://nic.funet.fi/pub/unix/parallel/POSYBL.TAR.Z

[16] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. In *IEEE Transactions on Software Engineering.* pages 190–205, March 1992.

[17] Delp, G. S. and Farber MemNet: An Experiment on High-Speed Memory Mapped Network Interface. In *Technical Report 85-11-IR, Dept. of Electrical Engineering, University of Delaware, 1986..*

[18] Lenoski, D. E. et al. Design of the Stanford DASH multiprocessor. In *Technical Report CSL-TR-89-403, Computer Systems Laboratory, Stanford University, December 1989.*

[19] H. Hellwagner and A. Reinefeld, editors. SCI: Scalable Coherent Interface. In *Architecture and Software for HighPerformance Compute Clusters.* volume 1734 of LNCS StateoftheArt Survey. Springer Verlag, Oct. 1999. ISBN 3540666966.

[20] Burkhardt, H. et al. Overview of the KSR1 Computer System. In *Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.*

[21] HAGERSTEN, E. et. al. DDM A Cache-Only Memory Architecture. In *Computer, Vol. 25, N 9, Septermber 1992.* pp. 44-54

[22] MAPLES, C. & WITTIE, L. Merlin: A Superglue for Multicomputer Systems. In *Compcon 90. IEEE Computer Society Press, Los Alamitos, 1990.* pp. 73-81

[23] LUCCI, S. et. al. Reflective-Memory Multiprocessors. In *Proc. 28th IEEE/ACM Hawaii Intl Conf. Systems Siences. IEEE Computer Society Press, Los Alamitos, 1995.* pp. 85-94

[24] Bisiani R., Ravishankar, M. PLUS: A Distributed Shared-Memory System In *Proceeding of the 17the Annual International Symposium on Computer Architecture.* Vol. 18, No. 2, May 1990, pp. 115-124.

[25] A.W. Wilson Jr., R.P. LaRowe Jr., J. Teller Hardware Assist for DSM In *Proceedings of the 13th International Conference on Distributed Computing Systems, May 1993.* pp. 314-324.

[26] D. Chaiken, J. Kubiatowicz, A. Agarwal Software-Extended Coherent Shared Memory: Performance and Cost In *Proceedings of the 21th Annual International Symposium on Computer Architecture, April 1994.* pp. 314-312.

[27] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, J. Hennessy The Stanford FLASH Multiprocessor In *Proceedings of the 21th Annual International Symposium on Computer Architecture, April 1994.* pp. 302-313

[28] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proc. of the 23rd Annual International Symposium on Computer Architecture.* pages 34-43, May 1996

[29] Wolfgang Karl and Martin Schulz Hybrid-DSM: An efficient alternative to pure software DSM systems on NUMA architectures. In *In Proceedings of the 2nd International Workshop on Software DSM.* (held together with ICS 2000), May 2000.

[30] M. Blumrich, R. Alpert, Y. Chen, D. Clark, S. Damianakis, C. Dubnicki, E. Felten, L. Iftode, K. Li, M. Martonosi, and R. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In *In Proceedings of the 25th International Symposium on Computer Architecture ISCA-25, Barcelona, Spain, May 1998.*

[31] A. Colmerauer Prolog II Reference Manual and Theoretical Model In *Internal Report, GroupeIA, U Aix-Marseille (Oct 1982).*

[32] A. Colmerauer Prolog, with unification replaced by constraint resolution. An Introduction to Prolog III. BYTE 12(9):177-182 (Aug 1987) Aix-Marseille, ca 1984.

[33] N.Heintze et al. The CLP(R) programmer's manual IBM T. J. Watson Research Center, 1992.

[34] Wolfgang Karl and Martin Schulz Constraint Satisfaction in Logic Programming. In *MIT Press, 1989.*

[35] Daniel Diaz The GNU Prolog web site, http://gnu-prolog.inria.fr/

[36] Ilog(c) The ILOG web site, http://www.ilog.com

[37] MPI: A Message-Passing Interface standard. In *The International Journal of Supercomputer Applications and High Performance Computing, 8, 1994.* In *Technical report, 1995. http://www.mpi-forum.org.*

[38] Youssefh. Disolver: The Distributed Constraint Solver. website: http://research.microsoft.com/%7Eyoussefh/DisolverWeb/Disolver.html

[39] Salvador Abreu. GC: A Constraint Solver in Java. In *Proceedings of the ESSLLI'96 Workshop on Programming Language Implemenatation, Prague, Czech Republic, 1996.*

[40] R. Zivan and A. Meisels. Message delay and discsp search algorithms. In *Journal paper in Ann. of Math & AI.* vol. 46, pp, 415–439, April 2006

[41] Christian Mller Run-Time Byte Code Compilation, Optimization, and Interpretation for Alice Diplomarbeit, 2006 Leif Kornstaedt Alice in the Land of Oz. In *Proceedings of the First Workshop on Multi-language Infrastructure and Interoperability (BABEL), 2001.*

[42] Brahim Hnich, Ian Miguel, Ian P. Gent, Toby Walsh website: http://www.csplib.org/

[43] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local-Area Network. In *IEEE Micro, 15(1):29-36, February 1995.*

[44] M. Eberl, H. Hellwagner, M. Schulz and B. Herland. SISCI: Implementing a Standard Software Infrastructure on an SCI Cluster. In *Proceedings of the First German Workshop on Cluster Computing, Chemnitz, Germany, November, 1997.* website: http://citeseer.ist.psu.edu/eberl97sisci.html

[45] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. In *IEEE Micro, pages 66–75, MarApr 1998.*

[46] L. Prylli and B. Tourancheau BIP: A New Protocol Designed for High Performance Networking on Myrinet. In *Rolim [25], pp. 472485.*

[47] Gabriel Antoniu and Luc Boug Implementing multithreaded protocols forrelease consistency on top of the generic DSM-PM2 platform. In *In Proceedings of the International Workshop on Cluster Computing (IWCC01), volume 2326 ofLNCS, pages 182191, Mangalia, Romania, August 2001..*

[48] Luís Almas, Rui Machado and Salvador Abreu. Design for a Parallel and Distributed Hybrid Constraint Programming Library. In *Salvador Abreu and Vitor Santos Costa, proceedings of the 7th International Colloquium on Implementation of Constraint and Logic Programming Systems. Universidade do Porto, Portugal, 2007.*