Departamento de Informática

Constraint Programming on a Heterogeneous Multicore Architecture

Rui Mário da Silva Machado
<ruimario@gmail.com>

Supervisor: Salvador Abreu (Universidade de Évora, DI)

Évora
2008

Departamento de Informática

Constraint Programming on a Heterogeneous Multicore Architecture

Rui Mário da Silva Machado
<ruimario@gmail.com>

Supervisor: Salvador Abreu (Universidade de Évora, DI)

Évora
2008

Abstract

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. Some approaches to constraint programming favour completeness, such as propagation-based systems. Others are more interested in getting to a good solution fast, regardless of whether all solutions may be found; this approach is used in local search systems. Designing hybrid approaches (propagation + local search) seems promising since the advantages may be combined into a single approach.

Parallel architectures are becoming more commonplace, partly due to the large-scale availability of individual systems but also because of the trend towards generalizing the use of multicore microprocessors.

In this thesis an architecture for mixed constraint solvers is proposed, relying both on propagation and local search, which is designed to function effectively in a heterogeneous multicore architecture.

Keywords: Constraint Programming, Cell

Resumo

Programação com restrições numa arquitectura multiprocessador heterogénea

As bibliotecas para programação com restrições são úteis ao desenvolverem-se aplicações em linguagens de programacão normalmente mais utilizadas pois não necessitam que os programadores aprendam uma nova linguagem, fornecendo ferramentas de programação declarativa para utilizacão com os sistemas convencionais. Algumas soluções para programação com restrições favorecem completude, tais como sistemas baseados em propagação. Outras estão mais interessadas em obter uma boa solucão rapidamente, rejeitando a necessidade de encontrar todas as soluções; esta sendo a alternativa utilizada nos sistemas de pesquisa local. Conceber soluções híbridas (propagação + pesquisa local) parece prometedor pois as vantagens de ambas alternativas podem ser combinadas numa única solução.

As arquitecturas paralelas são cada vez mais comuns, em parte devido à disponibilidade em grande escala de sistemas individuais mas também devido à tendência em generalizar o uso de processadores *multicore* ou seja, processadores com várias unidades de processamento.

Nesta tese é proposta uma arquitectura para resolvedores de restrições mistos, dependendo de métodos de propagação e pesquisa local, a qual foi concebida para funcionar eficazmente numa arquitectura heterogénea multiprocessador.

Acknowledgments

This important phase of my life could not have been so successful as it was without the support of people I know and met and to whom I am profoundly thankful.

First of all, I would like to thank my loving parents and my brother just for being present.

A loving thanks to my girlfriend Swani for being herself, for the patience, advice and her love. I would like to dedicate her this work.

A very special thanks to my tutor Dr. Werner Kriechbaum and my advisor Dr. Salvador Abreu for their time, availability, support, leadership and friendship at every moment.

Thanks to all of the Linux on Cell Test and Development Teams as well my whole department with whom was a pleasure to work with.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.* - Moore, Gordon E. (1965)

The microprocessor industry has been shifting its focus towards multiprocessor chips. Ever since it was stated, Moore's law has adequately described the progress of newly developed processors.

Before the turn of the millennium, the performance improvements of a processor were mostly driven by frequency scaling of an uniprocessor. Still, multiprocessor chips were already seen as a valid approach to increase performance. The focus on multiprocessor designs became clear with the diminishing returns of frequency scaling and sometimes physical limitations emerging.

The emergence of chip multiprocessors is a consequence of a number of limitations in today's microprocessor design: deep pipelining performance is exhausted, reduced

benefits of technology scaling for higher frequency operation, power dissipation approaching limit and memory latency.

To address these limitations and continue to increase the performance of processors, many chip manufactures are researching and developing multi-kernel processors.

An innovative and interesting example of such an architecture is the Cell Broadband Engine.

IBM, Sony and Toshiba Corporation have jointly developed an advanced microprocessor, for next-generation computing applications and digital consumer electronics. The Cell Broadband Engine is optimized for compute-intensive workloads and rich media applications, including computer entertainment, movies and other forms of digital content.

One important feature and a major difference from other new architectures is the fact that the Cell Broadband Engine is a heterogeneous multi-kernel architecture. Essentially, this means the architecture is composed of several processor cores and these cores have different instruction sets. One visible and direct consequence is that the simple re-compilation of any software program is not enough to take advantage of the processor's capabilities.

At a first and quick glance, one might tend to believe that the Cell Broadband Engine is just a normal Power PC with several co-processors, but Cell Broadband Engine is much more than that. These "co-processors" are powerful and independent processors each requiring a separate compiler, and have very specific features like DMA transfers and interprocessor messaging and control.

Such new CPU architectures offer a significant performance potential but pose the challenge that new programming paradigms and tools have to be developed and evaluated to unlock the power of such an architecture. Today software architectures for the exploitation of heterogeneous multi-core architectures are still a field of intensive research and experimentation.

The key term is parallelization. Software must be able to take advantage of dozens

or even hundreds of hardware threads. But parallelizing programs is not an easy task. Issues such as race conditions, data dependencies, communication and interaction between threads, with poor debugging support are extremely error-prone. Programmers tend to think sequentially and not in parallel and often this way of thinking is reinforced by major programming languages and their paradigms.

More declarative languages like functional and logic programming languages have fewer and more transparent dependencies and aliasing. Therefore such languages are much easier to extract parallelism from. The main problem with such languages is their lack of generalized adoption. They are mostly used in academia or very specialized groups.

Constraint Programming is a useful declarative methodology which has been applied in several ways:

1. As an extension to existing programming languages, such as Prolog, taking advantage of the complementarity provided by the two approaches (backtracking vs. propagation). This is the case for most Constraint Logic Programming (CLP) implementations.

2. As a library in which constraints become data structures of the host language, which are operated on by the library procedures. This is the case, for instance, for ILOG Solver [32] and GECODE [7].

3. As a special-purpose language, appropriate for solving problems formulated as constraints over variables. This is the case with, among others, Oz [34], OPL [31] or Comet [25].

The declarative nature of constraint satisfaction problems (CSP) strongly suggests that one tries to parallelize the computational methods used to perform the tasks related to solving CSPs, namely propagation. Indeed, this has been explicitly incorporated into most languages mentioned in point 3, which provide mechanisms to promote distributed execution of various aspects of the process.

In this thesis, we chose to follow approach number 2: to provide a library for constraint programming for an existing language. CASPER is inspired by the scheme used in AJACS [12] and extends it to include both propagation and local search techniques. CASPER relies on a purely functional approach to representing search-space state stores, and is designed to ensure that parallelization is viable by avoiding sharing as much as possible, to achieve the highest degree of independence between search-space state stores.

Outline

This thesis is organized as follows: after this Introduction chapter, Chapter 2 provides a background on the various topics and notations used in the subsequent chapters: different Parallel Computers architectures in general are presented as well as models and architectures of parallel programming. Finally, Constraint Programming is introduced and all associated notations which concern to this thesis. In chapter 3, the AJACS model is presented along with its main concepts and parallel architecture. The following chapter, chapter 4, presents the Cell/B.E. processor in general and details its internal architecture. In Chapter 5, we present and describe in detail the framework which we developed in the course of the present work and in chapter 6, some tests and their results are presented and discussed. Finally, in chapter 7 some conclusions are drawn and future directions for the work are suggested.

Chapter 2

Background

Parallel computing is a mainstay of modern computation and information analysis and management, ranging from scientific computing to information and data services. The inevitable and rapidly growing adoption of multi-core parallel architectures within a processor chip by all of the computer industry pushes explicit parallelism to the forefront of computing for all applications and scales, and makes the challenge of parallel programming and system understanding all the more crucial. The challenge of programming parallel systems has been highlighted as one of the greatest challenges for the computer industry by leaders of even the largest desktop companies.

Heterogeneous chip multiprocessors present unique opportunities for improving system throughput, reducing processor power. On-chip heterogeneity allows the processor to better match execution resources to each application's needs and to address a much wider spectrum of system loads - from low to high thread parallelism - with high efficiency.

2.1 Architectures

Parallel Computer Memory Architectures

One way to classify multiprocessor computers is based on their memory architectures or how processor(s) interact(s) with memory.

Shared Memory

In shared memory computers (figure 2.1.1), the same memory is accessible to multiple processors in a global address space.

All processors can work independently but since memory is shared its access must be synchronized by the programmer. When one task accesses one memory resource, this resource cannot be changed by some other task. Every change in a memory location is therefore visible to all other processors.

Shared memory machines can be divided into two main classes based upon memory access times: Uniform Memory Access(UMA) and Non-Uniform Memory Access (NUMA). In NUMA the memory access times depends on the memory location relative to a processor where in UMA the access times are equal.

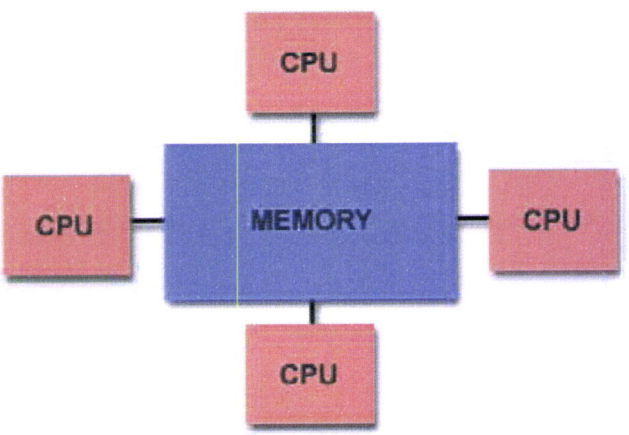

Figure 2.1.1: Shared memory architecture

Distributed Memory

Distributed memory systems (figure 2.1.2) require a communication network to connect the processors and their memory.

Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors. Because each processor has its own local memory, it operates independently in the sense that changes made to its local memory have no effect on the memory of other processors. When a processor needs to access data in another processor's memory, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is again the programmer's responsibility.

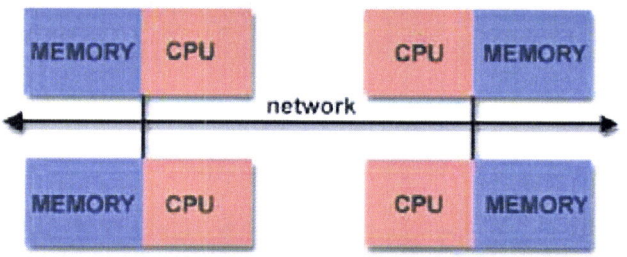

Figure 2.1.2: Distributed memory architecture

Hybrid Distributed-Shared Memory

It is possible to combine the previous architectures in a hybrid model, as figure 2.1.3 illustrates. In fact, that's what modern large computers do.

The processors in a Symmetric MultiProcessing (SMP) can access the system's memory globally like in shared memory computers and use networking to move data across the several distributed SMP machines as with the distributed memory systems.

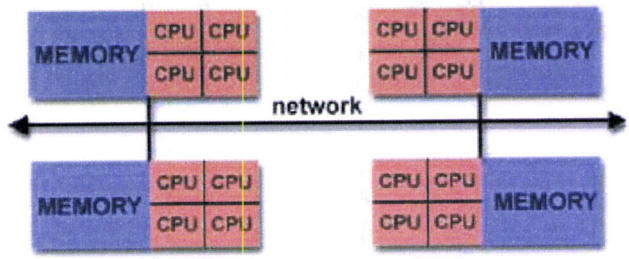

Figure 2.1.3: Hybrid memory architecture

2.2 Parallel Programming

Traditional Von Neumann computing platforms contain a single processor, which computes a single thread of control at each instant. High-performance computing platforms contain many processors, with potentially many threads of control. Parallel programming has become the default in many fields where immense amounts of data needs to be processed as quickly as possible: oil exploration, automobile manufacturing, pharmaceutical development and in animation and special effects studios. Such different tasks and the algorithms associated with them present different styles of parallel programming. Some tasks are data-centric and algorithms for working on them fit into the SIMD (Single Instruction, Multiple Data) model. Others consist of distinct chunks of distributed programming, and these algorithms rely on good communication models among subtasks.

Challenges

The key to parallel programming is to locate exploitable concurrency in a task. The basic steps for parallelizing any program are:

- Locate concurrency.
- Structure the algorithm(s) to exploit concurrency.
- Tune for performance.

The major challenges are:

- Data dependencies.
- Overhead in synchronizing concurrent memory accesses or transferring data between different processor elements and memory might exceed any performance improvement.
- Partitioning work is often not obvious and can result in unequal units of work.
- What works in one environment might not work in another, due to differences in bandwidth, topology, hardware synchronization primitives and so on.

Parallel Programming Models

A parallel programming model is an abstraction to express parallel algorithms in parallel architectures. One goal of a programming model is to improve the productivity of the programmer. It includes areas of applications, programming languages, compilers, libraries, communications systems, and parallel I/O. It is up to the developers to choose the model which best suits their needs.

Parallel models are implemented in several ways: as libraries invoked from traditional sequential languages, as language extensions, or completely new execution models.

Shared Memory Model

In the shared-memory programming model tasks share a common address space which they read and write asynchronously. An advantage of this model from the programmer's point of view is that the notion of data co-ownership is lacking, thus there is no need to specify explicitly the communication of data between tasks.

Still, this model needs mechanisms to synchronize access to the shared memory (e.g. locks, semaphores).

Advantages

- Conceptually easy to understand and hence design programs for
- Easy to identify opportunities for parallelism

Disadvantages

- Lack of portability as this model is often implemented in an architecture specific programming language
- May not be suitable for loosely coupled distributed processors due to the high communication cost

One implementation of this model is Distributed Shared Memory (DSM). In DSM, the common address space can point to memory of other machine. DSM can be implemented in hardware or in software. In software, a DSM system can be implemented in the operating system or as a programming library.

Although DSM gives users a view such that all processors are sharing a unique piece of memory, in reality each processor can only access the memory it owns. Therefore the DSM must be able to bring in the contents of the memory from other processors when required. This gives rise to multiple copies of the same shared memory in different physical memories. The DSM has to maintain the consistency of these different copies, so that any processor accessing the shared memory should return the correct result. A memory consistency model is responsible for the job.

Intuitively, the read of a shared variable by any processor should return the most recent write, no matter if this write is performed by any processor. The simplest solution is to propagate the update of the shared variable to all the other processors as soon as the update is made. This is known as sequential consistency (SC). However, this can generate an excessive amount of network traffic since the content of the update may not be needed by every other processor. Therefore certain relaxed

memory consistency models were developed. Most of them provide synchronization facilities such as locks and barriers, so that the shared memory access can be guarded to eliminate race conditions. The most popular memory consistency models are: sequential consistency (SC), eager release (ERC), lazy release (LRC), entry (EC) and scope (ScC).

Threads Model

The threads model spins around the concept of a thread. A program can be split up in several threads that run simultaneously. In a uniprocessor system this "simultaneously" means "almost simultaneously" because the processor can switch between threads so fast that it gives the illusion of executing more than one thing at the same time. In modern multiprocessor and multicore architectures, threads are really executed at the same time, in different units.

The threads model is usually associated to a shared memory architecture and could be included in the shared memory programming model. But since the concept of thread is so widely and independently used that it deserves a place of its own.

Advantages

- The overhead associated with creating a thread is much less than creating an entire process
- Switching between threads requires much less work by the operating system
- Many programmers are familiar with writing multi-threaded programs because threads are a basic construct in many modern programming languages like Java

Disadvantages

- Writing a multi-threaded program can be much thougher than for other programming models
- Synchronization mechanisms are required to control access to shared variables

Two implementations of this model are POSIX threads [5] and openMP([2] [10]).

Message Passing Model

Message passing is probably the most widely used parallel programming model today. Message-passing programs create multiple tasks, with each task encapsulating local data. Each task is identified by a unique name, and tasks interact by sending and receiving messages to and from named tasks.

The message-passing model does not preclude the dynamic creation of tasks, the execution of multiple tasks per processor, or the execution of different programs by different tasks. However, in practice most message-passing systems create a fixed number of identical tasks at program startup and do not allow tasks to be created or destroyed during program execution. These systems are said to implement a Single Program Multiple Data (SPMD) programming model because each task executes the same program but operates on different data.

Advantages

- This model is applicable to both tightly coupled computers and geographically distributed systems
- Message passing libraries provide a set of functionality and level of control that is not found in any of the other models
- All other parallel programming models can be implemented by the message passing model

Disadvantages

- All of the responsibility for an effective parallelism scheme is placed on the programmer. The programmer must explicitly implement a data distribution scheme and all interprocess communication and synchronization, while avoiding deadlock and race conditions
- Some parallel programmers prefer to have this level of control however it can be difficult for novice programmers to implement effective parallel programs

Two widely used implementations are MPI [16] and PVM [9].

MPI is a message passing library specification. MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries. The goal of MPI is to provide a widely used standard for writing message passing programs. In MPI, all parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs. The number of tasks dedicated to run a parallel program is static. New tasks can not be dynamically spawned during run time although the MPI-2 specification addresses this issue.

Data Parallel Model

Another commonly used parallel programming model, the data parallel model, calls for exploitation of the concurrency that derives from the application of the same operation to multiple elements of a data structure, for example, "add 2 to all elements of this array, or increase the salary of all employees with 5 years service. A data-parallel program consists of a sequence of such operations. As each operation on each data element can be thought of as an independent task, the natural granularity of a data-parallel computation is small, and the concept of locality does not arise naturally. Hence, data-parallel compilers often require the programmer

to provide information about how data are to be distributed over processors, in other words, how data are to be partitioned into tasks. The compiler can then translate the data-parallel program into an SPMD formulation, thereby generating communication code automatically.

Advantages

- Gives the user the ability to process large volumes of data very fast
- Only one piece of code needs to be produced to implement all of the parallelism

Disadvantages

- Due to the large volumes of data involved in a typical data parallel computation, this model may not be suitable for geographically distributed processors
- Requires high bandwidth communications to transfer and share data

Hybrid Model

In this model, any two or more parallel programming models are combined. Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP).

Another common example of a hybrid model is combining data parallel with message passing. Data parallel implementations on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer.

The advantages and disadvantages come from the combination of the models being used.

2.3 Designing Parallel Programs

One possible goal for parallel programming is performance improvement. In this perspective, the design of parallel programs demands an extra effort from the programmer. Several factors must be considered in order to decrease the execution wall clock time.

It is necessary to understand the problem that one wants to parallelize. One should identify the program's hotspots, bottlenecks and inhibitors. Hotspots are where most work is done and can be identified by using profiling tools. Bottlenecks are areas which slow down the program's execution as for example I/O such as disk access. They should be minimized by restructuring the code or even using another algorithm. Inhibitors are portions of code that restrain parallelism since they are not independent. A good example of an inhibitor is a data dependency.

Partitioning is one of the first steps to be made. It represents the way how we divide the work being done that can be distributed to multiple tasks. There are 2 ways to do this. Functional partitioning focuses on the computation (or control of the program) and each parallel task performs a part of the overall task. Domain partitioning focuses on the data associated to the problem and the parallel tasks works on a different portion of the data.

As there are several tasks running in parallel, they might need to communicate with each other by sharing or exchanging data. This need for communication depends on the task being performed since some tasks run very independently and some don't. Very independent tasks don't need to share data - embarrasingly parallel - while dependent tasks are not that simple and therefore communication factors must be included. When designing a parallel application, one must pay attention to factors such as cost of communication, bandwidth/latency in synchronous/asynchronous communication and the scope of the communication.

Another factor is synchronization. Synchronization is needed when two or more tasks need access to a shared resource and this is very important because it influences the

correctness of the computed result. There are different ways to synchronize accesses such as using locks or by communication between tasks.

Dependencies exist between program statements when the order of statement execution affects the results of the program. A data dependency results from multiple use of the same location(s) in storage. Data dependencies are one of the primary inhibitors to parallelism.

To obtain a maximum performance, all parallel tasks should be busy all the time thus a correct distribution of the work is required. A scheduler is a possibility if one desires a dynamic assignment of jobs.

Granularity is, in parallel computing, the ratio of computation over communications. With fine-grain parallelism, the amount of computational work done between communications is relatively small. In coarse-grain parallelism, the ratio is high, with large amounts of computational work between communications/synchronization events. Both types of granularity have their advantages and drawbacks. The choice for the level of granularity depends on the algorithm, the data set and the hardware environment.

A proper evaluation of limits and costs should be performed. Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

$$speedup = \frac{1}{1 - P} \tag{2.3.1}$$

When introducing the number of processors (N):

$$speedup = \frac{1}{\frac{P}{N} + S} \tag{2.3.2}$$

where P = parallel fraction, N = number of processors and S = serial fraction.

This law gives an idea of how much gain one gets with the parallelization although other issues should be taken into account to evaluate the costs and complexity of parallelization. This includes resource requirements, portability and scalability.

2.4 Parallelism and Programming Languages

There are 2 two major approaches to parallel programming: implicit parallelism and explicit parallelism. In implicit parallelism, the compiler or some other program is responsible for the parallelization of the computational task. In explicit parallelism, the programmer is responsible for the task partitioning through language constructs or extensions (we already referred MPI as an explicit parallelism specification).

It has long been recognized that declarative programming languages, including logic and functional programming languages, are potentially better suited to parallel programming. The key factor is a clear separation of "what" the program computes from the details of "how" the computation should take place.

Imperative programming languages rely on state and time. The state changes over time as variables are assigned values and time must be considered when looking at an imperative program. On the other hand, declarative languages are independent of time. While imperative programs consist of a set of statements executed in order, declarative programs don't care about the order or even how many times an expression is evaluated.

The most common way of achieving a speedup in parallel hardware is to write programs that use explicit threads of control in imperative programming languages in spite of the fact that writing and reasoning about threaded programs is notoriously difficult. A lot of work has been put in imperative languages (like Java, C++ and C#) to take advantage of concurrency.

It is worthwhile to note about *concurrent* and *parallel* programming that, although both want to exploit concurrency that is, execution of computations that overlap over time, it can safely be argued that they are not synonymous. In either case, both tend to use the same models (see above) and imply communication between tasks.

Concurrency is a language concept that expresses logically independent

> computations. Parallelism is an implementation concept that expresses activities that happen simultaneously. In a computer, parallelism is used only to increase performance. [35]

Concurrency and parallelism are orthogonal concepts.

There are three main models of concurrency in programming languages (from [35]): declarative concurrency, message passing concurrency and shared state concurrency.

Declarative concurrency is the easiest paradigm of concurrent programming. It keeps programs simple and without race conditions or deadlocks.

It relies on declarative operations. A declarative operation is independent (does not depend on any execution state outside of itself), stateless (has no internal execution state that is remembered between calls), and deterministic (always gives the same results when given the same arguments). ' The drawback of this paradigm is that it doesn't allow programs with nondeterminism. Only more academic languages like Oz [33] and Alice [24]use this paradigm.

Message passing is a model in which threads share no state and communicate with each other via asynchronous messaging. It extends declarative concurrency introducing communication channels to remove the limitation with non-determinism. This is the model employed by the Erlang language.

The **Shared state** consists of a set of threads accessing a set of shared passive objects. The threads coordinate among each other when accessing the shared objects. They do this by means of coarse-grained atomic actions, e.g., locks, monitors, or transactions.

The concurrency is more expressive and gives more control to the programmer but reasoning with this model is more complex. Shared state concurrency is the model employed by the Java language.

2.5 Constraint Programming

Introduction

> *"Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."* - Eugene C. Freuder, Constraints Journal, 1997

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

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

Historical Remarks

Constraint programming has been used in Artificial Intelligence Research since the 1960's. The first system known to use constraints was Sketchpad, a program written by Ivan Sutherland, the "father" of computer graphics, that allowed the user to draw and manipulate constrained geometric figures on the computer's display.

In the 70s, Logic Programming was born. Most of the development and achievements in the field of constraint programming was done by Logic Programming researchers because constraints have a very natural relationship with logical reasoning and one

of the reasons why the extension of logic languages such as Prolog to include constraints has been so formally clean, convenient and natural. In fact, the main step towards modern constraint programming was achieved when it was noted that logic programming (with unification over terms) was just a particular kind of constraint programming. It has led to the definition of a general framework called constraint logic programming (CLP [21]).

It is easy to get confused and see Constraint Programming as something strictly related to Logic Programming when in fact constraint theory is completely orthogonal to the programming paradigm.

In the late 80's, one powerful observation has been that constraints can also be used to model communication and synchronization among concurrent agents, in such a way that these tasks are now described in a more general and clean way, and can be achieved with greater efficiency and flexibility. Such observation is the basis of the concurrent constraint programming framework (CC [1]).

Already since the beginning of the 90's, constraint-based programming has been commercially successful. One lesson learned from applications is that constraints are often heterogeneous and application specific. In the beginning of constraint programming, constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language. To allow more flexibility and customization of constraint solvers, Constraint Handling Rules (CHR) was proposed. CHR [14] is essentially a concurrent committed-choice language consisting of multi-headed rules that transform constraints into simpler ones until they are solved.

The architecture of constraint programming is also suited for embedding constraints in more conventional languages. This characterizes *constraint imperative programming.*

In constraint imperative programming, the user can use constraints which relate program's variables and objects. Besides using the language's conventional features and define constraints which are implemented in the integrated constraint solver,

the user can define his own constraints, augmenting the solver's capabilities.

Imperative languages can also be extended with language elements from logic programming, such as non-deterministic computations with logical variables and backtracking.

Constraints systems

A constraint system is a formal specification of the syntax and semantics of the constraints. It defines the constraints symbols and which formulae are used for reasoning in the context of programming languages.

Finite Domain constraints have a finite set as its domain. This domain can be integers but also enumerable types like colors or resources which should be planned for a process.

Many real-life combinatorial problems can be modeled with this constraint system. Finite domain constraints are very well suited to puzzles, scheduling and planning. For example: an university's timetabling system.

Boolean constraints are a special case of finite domain constraints where the domain contains only two values, true and false. One area of application of Boolean constraints is modeling digital circuits. They can be applied to the generation, specialization, simulation and analysis (verification and testing) of the circuits.

There are often cases when, instead of imposing a constraint C, we want to speak (and possibly constrain) its truth value. For example, logical connectives such as disjunction, implication, and negation constrain the validity of other constraints. A *reified* constraint is a constraint C which reflects its validity in a boolean variable B:

$$C \iff B = 1 \wedge B \in \{0, 1\}$$

A **Rational Tree** is a tree which has a finite set of subtrees. Tree constraints can be used for modeling data structures, such as lists, records and trees, and for expressing

algorithms on these data structures. One of the applications for this constraint system is program analysis where one represents and reasons about properties of a program.

Constraints can also have **Linear Polynomial Equations** domains. Here, Linear arithmetic constraints are linear equations and inequalities. This type of constraints is important for graphical applications like computer aided design (CAD) systems or graphical user interfaces, but also for optimization problems as in linear programming.

The **Non-linear Equations** constraint system is an extension of the one for linear polynomials. Problems for solving this type of constraints arise with the inclusion of, for example, multiplication or trigonometric functions. Despite the fact of not having a trivial solver, non-linear constraints appear in the modeling, simulation and analysis of chemical, physical processes and systems. An application area is financial analysis or robot motion planning.

Constraint Solving Algorithms

An important component is the **constraint solver**. A **constraint solver** implements an algorithm for solving allowed constraints in accordance with the constraint theory. The solver collects the constraints that arrive incrementally from one or more running programs. It puts them into the constraint store, a data structure for representing constraints and variables. It tests their satisfiability, simplifies and if possible solves them. The final constraint that results from a computation is called the answer or solution.

There are two main approaches for constraint solving algorithms: variable elimination and local consistency (local propagation) techniques. Variable elimination achieves satisfiability completeness while local-consistency techniques have to be interleaved with search to achieve completeness.

Variable-elimination algorithms work by eliminating multiple occurrences of vari-

ables. Typically the allowed constraints are equations which are computed to obtain a normal form (or solution). An example of a variable elimination algorithm is the Gaussian method for solving linear polynomial equations.

Local consistency (local propagation) basically adds new constraints in order to cause simplification: new sub-problems of the initial problem are simplified and new implied constraints are computed (propagated).

Local consistency problems must be combined with search to achieve completeness. Usually, search is interleaved with constraint solving: a search step is made, adds a new constraint that is simplified together with the existing constraints. This process is repeated until a solution is found.

Search can be done by trying possible values for a variable X. These search procedures are called *labeling*. Often, a labeling procedure will use heuristics to choose the next variable and value for labeling. The chosen sequence of variables is called variable ordering.

Constraint Satisfaction Problem (CSP)

Constraint Satisfaction arose from research in the Artificial Intelligence (AI) field. A considerable amount of work has been focused on this paradigm contributing to significant results in constraint-based reasoning.

A Constraint Satisfaction Problem (CSP) consists of:

- a set of variables $X = \{x_1, ..., x_n\}$,
- for each variable x_i, a set Di of possible values (its domain),
- and a set of constraints restricting the values that the variables can simultaneously take.

A solution to a CSP is an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied. This includes finding:

- just one solution, with no preference as to which one.
- all solutions.
- an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables.

A CSP is a combinatorial problem which can be solved by search. This differs from Constraint Solving. Constraint solving uses variables with infinite domains and relies on algebraic and numeric methods.

Search Algorithms

A CSP can be solved by trying each possible value assignment and see if it satisfies all the constraints. Then there's backtracking, a more efficient approach. Backtracking incrementally attempts to extend a partial solution toward a complete solution, by repeatedly choosing a value for another variable and keeping the previous state of variables so that it can be restored, should failure occur.

The problem with such techniques is the late detection of inconsistency. Hence various **consistency techniques** were introduced to prune the search space, by trying to detect inconsistency as soon as possible. Consistency techniques range from simple node-consistency and the very popular arc-consistency to full, but expensive path consistency [30].

One can combine systematic search algorithms with consistency techniques. The result are more efficient constraint satisfaction algorithms. Backtracking can be improved by looking at two phases of the algorithm: moving forward (forward checking and look-ahead schemes) and backtracking (look-back schemes) [3].

Also very important is the order in which variables are considered. The efficiency of search algorithms like backtracking that attempts to extend a partial solution depends on this order. Likewise, the order of the values chosen for a variable affects

the algorithm performance. Various heuristics for ordering of values and variables exist.

Another approach to guide search is using heuristics and stochastic algorithms also known as Local Search.

The term heuristic is used for algorithms which find solutions among all possible ones ,but they do not guarantee that the best will be found,therefore they may be considered as approximately and not accurate algorithms.These algorithms,usually find a solution close to the best one and they do so fast and easily. Sometimes these algorithms can be accurate, that is they actually find the best solution.

To avoid getting stuck at "local maxima/minima" they are equipped with various heuristics for randomizing the search. Their stochastic nature cannot guarantee completeness like the systematic search methods.

Some examples of this kind of algorithms are the classics Hill-Climbing and Greedy algorithms [36] as well as Tabu-Search [17], Min-Conflict [37] or GSAT.

Constraints and Programming Languages

Constraint Logic Programming

Constraint Logic Programming (CLP [22]) is a combination of logic programming and constraint programming. The addition of constraints makes programs more declarative, flexible and in general, more efficient.

In the end of the 70's, efforts have been made to make logic programming more declarative, faster and more general. It was at this point, that it was recognized that constraints could be used in logic programming to accomplish the objectives of declarativeness, speed and generality. By embedding a constraint solver to handle constraints new possibilities open up. For example, constraints can be generated (and checked) incrementally, thus catching inconsistency early in the solving process or in other words, making the program execution faster.

In CLP, a store of constraints is maintained and kept consistents at every computation step. Each clause of the CLP program matching one of the goals in the store gets its constraints and goals accumulated in the store. However, the new constraints can only be added if they are compatible with those already present in the store. This means that the satisfiability of the whole new set of constraints has to be maintained.

CLP languages combine the advantages of LP languages (declarative, for arbitrary predicates, non-deterministic) with those of constraint solvers (declarative, efficient for special predicates, deterministic). Specially useful is the combination of search with solving constraints which can be used to tackle combinatorial problems (usually with exponential complexity).

Concurrent Constraint Programming

Concurrent Constraint Programming arises from the observation that constraints can be used to model concurrency and communication between concurrent processes (agents). It is a generalization of CLP with added concurrency [11].

This new paradigm leads to many consequences. One of the most important is that the entailment operation is now present (wasn't in CLP) so any constraint can be checked for satisfiability or for entailment. A constraint is entailed when its information is already present in the constraint, entailing the former.

The computation state is a collection of constraints, and each of the concurrent agents may either add a new constraint to the state (like in CLP) or check whether a constraint is entailed by the current state. Such a test may succeed or fail, but the new thing is that it can also suspend, and this happens when the new constraint is not entailed by the store but is consistent with it. If after another agent adds enough information to the store to make it entail or be inconsistent with the considered constraint, then the suspended action will be resumed and either succeed or fail. The framework is therefore monotonic, that is, constraints can never be deleted.

This add/check/suspend is based on a ask-and-tell mechanism. Tell means imposing a constraint as it happened in CLP or in other words, adding a constraint to the store. Ask means "asking" if a constraint already holds (this is done by an entailment test). One important difference between CLP and CC is don't-care non-determinism. Don't-care non-determinism (also referred as committed choice) means that if there are different clauses to choose from, just one arbitrary clause will be taken and the alternatives will be discarded. This means search is being eliminated leading to a gain in efficiency but like always, a loss in expressiveness and completeness.

Constraint Handling Rules

Constraint Handling Rules (CHR) is one of the many proposals made to allow more flexibility and customization of constraint solvers. Instead of a built-in constraint solver which is hard to modify, CHR defines simplification and propagation over user-defined constraints [6].

The CHR language has become a major specification and implementation language for constraint-based algorithms and applications. Algorithms are often specified using inference rules, rewrite rules, sequents, proof rules, or logical axioms that can be directly written in CHR. Based on first order predicate logic, the clean semantics of CHR facilitates non-trivial program analysis and transformation. About a dozen implementations of CHR exist in Prolog, Haskell, and Java.

CHR are essentially a committed-choice language consisting of guarded rules with multiple head atoms. CHR define simplification of, and propagation over, multi-sets of relations interpreted as conjunctions of constraint atoms. Simplification rewrites constraints to simpler constraints while preserving logical equivalence (e.g. $X > Y, Y > X \Rightarrow \mathit{false}$). Propagation adds new constraints which are logically redundant but may cause further simplification (e.g. $X > Y, Y > Z \Rightarrow X > Z$). Repeatedly applying the rules incrementally solves constraints (e.g. $A > B, B > C, C > A$ leads to false). With multiple heads and propagation rules, CHR provide

two features which are essential for non-trivial constraint handling.

Imperative Constraint Programming

The Constraint Imperative Programming (CIP) family of languages integrates constraints and imperative, object-oriented programming. In addition to combining the useful features of both paradigms, the ability to define constraints over user-defined domains is also possible.

Embedding constraints in conventional programming languages is usually done by extending a language's syntax, through a library or by creating new languages. Some languages have been developed to provide constraints reasoning. For example, **Oz** [34] is a high-order concurrent constraint programming system. It combines ideas from logic, functional and concurrent programming. From logic programming it inherits logic variables and logic data structures to try to provide problem solving capabilities of logic programming. Oz comes with constraints for variables over finite sets (finite domain variables) .

ILOG CP [32] is a C++ library that embodies Constraint Logic Programming (CLP) concepts such as logical variables, incremental constraint satisfaction and backtracking. It combines Object Oriented Programming (OOP) with CLP. The motivation for using OOP is that the definition of new classes is a powerful mean for extending software. Modularity is something that has been recognized as a limitation in Prolog.

Application Areas

Some example application areas of constraint programming [4], [40]:

- Computer graphics (to express geometric coherence in the case of scene analysis, computer-aided design,...)

- Natural language processing (construction of efficient parsers,speech recognition with semantics,...)
- Database systems (to ensure and/or restore consistency of the data)
- Operations research problems (like optimization problems: scheduling, sequencing, resource allocation, timetabling, job-shop, traveling salesman,...)
- Molecular biology (search for patterns, DNA sequencing)
- Business applications (option trading)
- Electrical engineering (to compute layouts, to locate faults, verification of circuit design...)
- Internet(constrained web queries)
- Numerical computation (computation with guaranteed precision for chemistry, engineering, design,...)

Real applications developed:

- Lufthansa : Short-term staff planning
- Hong-Kong container Harbor : Resource Planning
- Renault: Short-term production planning
- Nokia : Software configuration for mobile phones
- Airbus : Cabin layout
- Siemens : Circuit verification
- Caisse d'Epargne : Portfolio management

2.6 Summary

In this chapter we presented the background related to this thesis' work. Parallel computing, as a mainstay in today's computing environments, creates a great challenge to the computing industry. The adoption of multi-core parallel architectures is one of the driving forces and bringing parallel programming to applications at every scale.

One way to classify multiprocessor computers is based on their memory architectures or how processor(s) interact(s) with memory: shared-memory, distributed memory and hybrid memory.

The variety of architectures require different abstractions to extract parallelism. For that exist several programming models suitable for expressing programs in different parallel architectures, increasing the programmer's productivity like for example the message passing model or the threads model.

Together with a good programming model, developing parallel programs includes considering several factors for getting performance improvements. The list is rather extensive:

- identify hot spots, bottlenecks and inhibitors
- partitioning
- cost of communication, latency/bandwidth in synchronous/asynchronous communication
- synchronization
- dependencies
- work distribution
- granularity

Obviously parallelism is also reflected in programming languages. Besides parallel hardware and good abstractions and design, the implementation must be done. Usually this happens by implicit or explicit parallelism.

The most common way is to write programs with explicit threads in languages like Java and C++. Still, declarative languages are recognized as better suited to parallel programming.

There are three main models of concurrency - a language concept - in programming languages (from [35]): declarative concurrency (Oz), message passing concurrency (Erlang) and shared state concurrency (Java).

One declarative approach to programming is Constraint Programming, one of the main topics of this thesis. The idea of constraint-based programming is to solve problems by stating constraints which must be satisfied by the solution(s) of the problem. Constraint programming has been used since the 60's and gone through several improvements (CLP, CC, CHR).

Constraints can be used in different programming paradigms like logic programming or imperative programming. Also, constraints are very flexible allowing different domains for reasoning:

- finite domain
- boolean
- rational trees
- linear polynomial equation
- non-linear equations

In constraint programming one wants to solve constraint problems with the help of the solver. The solver implements an algorithm to solve constraints through variable elimination or propagation. One important component from a solver (when using

propagation) is search. And this is a whole new category with a extensive variety of techniques and algorithms, from backtracking to heuristic methods.

Constraint programming was combined with programming languages in different forms (section 2.5): constraint logic programming (CLP), concurrent constraint programming (CC), constraint handling rules (CHR) and imperative constraint programming.

And although not being a well-known paradigm, constraint programming has applications in several areas (computer graphics, operations research or molecular biology) and is already used by several companies like Lufthansa, Renault and more.

Chapter 3

AJACS

AJACS is a toolkit developed for Concurrent Constraint programming implemented in Java. AJACS relies on a distributed shared memory (DSM) system, operating under a special JVM implementation, Hyperion, which compiles to C. The target code then uses the PM2 multi-threading library, over which a DSM implementation has been constructed and is used to share memory ranges (in the form of Java objects), under an appropriate consistency model.

AJACS' architecture is centered around a few key concepts and a parallel execution model.

This chapter presents AJACS giving relevance to what is important for this thesis' work: its model and parallel architecture for obtaining faster resolution of a CSP by exploiting the search space in parallel.

3.1 Concepts

A brief enumeration of the main concepts of AJACS follows:

Value

A *Value* represents a subset of a variable's domain. A value is said to be ground if it contains exactly one *singular value* that is, exactly one element from a variable's domain.

Variable

Variables are, abstractly, thought of as the set of *Values* with the same index to the *Store* in a set of *Stores*.

Store

A *Store* is an indexed collection of *Values*. Each index of the *Store* represents a variable of the problem we are trying to solve. At each propagation step, a new *Store* is created from the current one. The most recent *Store* differs from its ancestors by one *Variable* with a restricted domain. This difference must be saved in the Store in order to obtain the successor stores.

Constraint

A *Constraint* is a relation between variables of a problem. In the AJACS model, *Constraints* are responsible for propagation, after changing one variable's value.

A *Constraint* affects a given number of variables. This is called the constraint's *environment*.

After propagating the changes done to a variable, the *Constraint* holds a boolean value which conforms to the *Store*'s consistency: true when the store is consistent, false otherwise.

Problem

As already referred, a CSP is defined by a set of variables and associated domain (i.e. a *Store*) together with a set of constraints over those variables. A *Problem* models exactly this definition.

Search

The concept of *Search* embodies the procedure which finds solutions for a given *Problem*. A *Search* is a series of search steps which finish when a solution is found or the search space is exhausted.

Strategy

A search step of the *Search* is the concrete action required by the *Strategy*. A *Strategy* is applied to a *Store* - a state of computation - in order to retrieve its successor. The retrieval of a successor entails:

- which non-ground variable is the next to be selected
- for the select variable, how to reduce its domain that is, which singular value it will take.

3.2 An Example

This example is taken from [13]

Consider the generic problem of assigning a starting time for some activities. The activities have a well known duration, measured in hours. Suppose also that some of the activities, affect resources that cannot be shared, i.e., they are unary and exclusive resources. Consider, for instance, that we want to time tabling teacher's

classes. The teacher has 3 classes, two of them taking 2 hours, and the other taking three hours. Suppose also, that these classes can only take place at Mondays, whose hours we represent by the values from 1 to 9 (1 for Monday's 9:00AM, 2 for Monday's 10:00AM, etc, 9 for Monday's 5:00PM). Assume also that hour 5 (1:00PM) is the lunch break.

The definition of a problem holds a store, and a set of constraints. The initial store, *s_init* for this problem is specified by the three values that represent all the possible initial starting times of the classes. The problem is defined by p = new Problem(s_init). Consider *NoOverlap(i,j,di,dj)*, the constraint that assures that the activities (corresponding to the values, i and j, with durations respectively di and dj) do not overlap in time. The specific constraints for the problem are added to the problem, doing:

```
int[] d={2,2,3}
for (i=0; i<2; ++i)
   for (j=i+1; j<3; ++j)
        p.add (new NoOverlap (i,j,d[i],d[j}))
```

Now that the variables are defined and the constraints are set, it is possible to locate a solution over the search space defined for the variables. Figure 3.2.1 shows the sequence of stores generated in this process. There is an arc from $s \rightarrow s'$ if the ancestor store of s' is s ($s'/s_p \equiv s$). The stores are generated in the sequence defined by the rounded arc.

3.3 Parallel execution architecture

A search tree of a problem is constructed by taking a store and applying a search strategy to it. The resulting stores remain in the tree if they are consistent. Applying the same procedure to each store generated results in the complete search tree of a

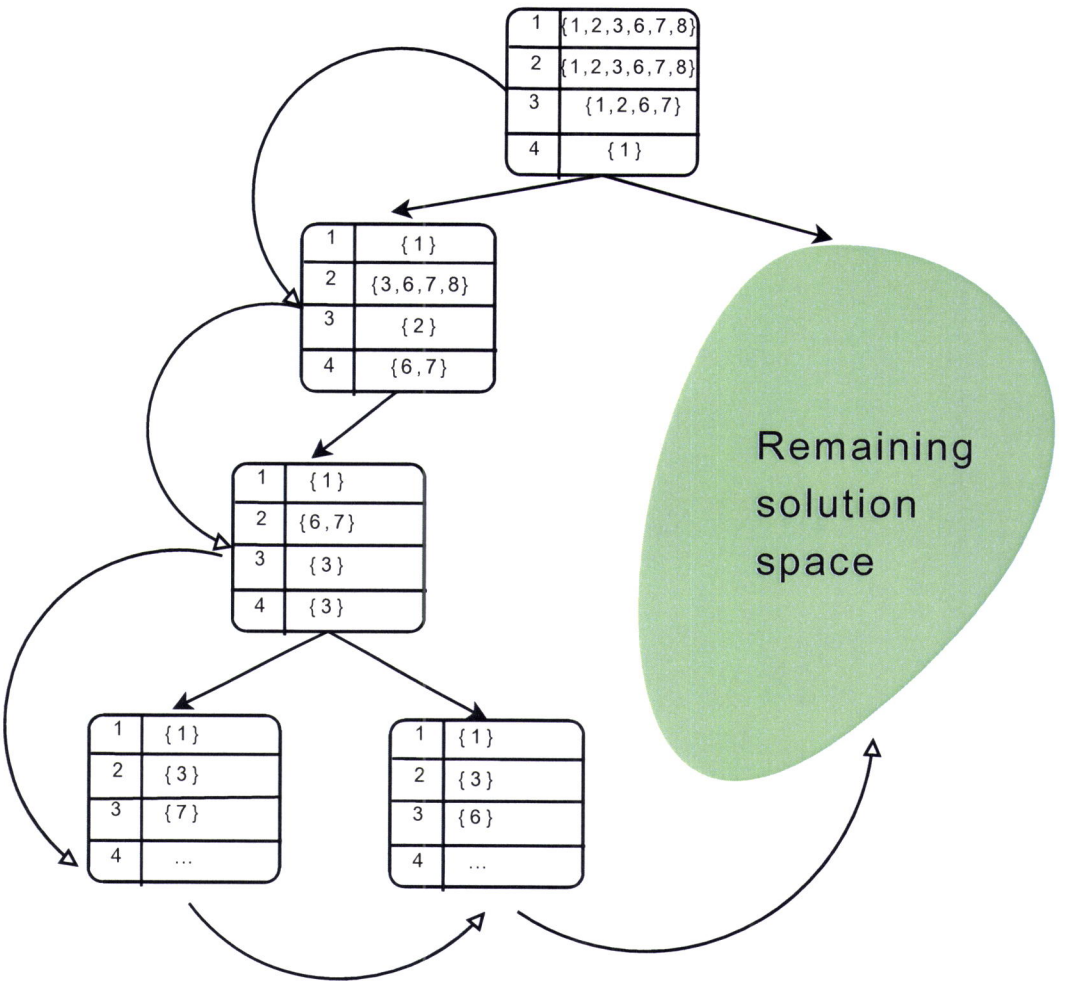

Figure 3.2.1: Time tabling example

problem. This design (partially shown in figure 3.3.1) is suitable for parallelization. It is possible to take a subtree and work on it, separately. This could be done by different agents in parallel (in the figure, represented by the big arrows).

Controllers and Workers

The agents responsible for solving a problem in a parallel environment can be of two different types: Controller and Worker. A problem will have one Controller and several Workers. A Controller is responsible for the management of workers and a

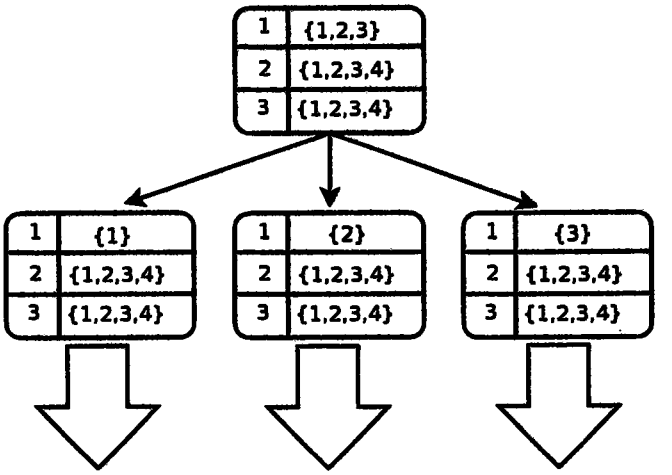

Figure 3.3.1: Ajacs parallel architecture

Worker is responsible for traversing the search tree. A traversal is accomplished by repeating a traversal step:

expand a store and verify if any of these resulting stores is a solution (ground). If it is a solution, then notify the controller that a solution was found. The controller will act according to the problem: if only one solution was needed, then the controller stops all workers and the problem is solved; else expand another store for solving.

The traversals can be executed in two ways which depend on the store which will be subject of a traversal step. The chosen store can be a child of the store expanded in the previous traversal step or it can be any store of the tree that hasn't been expanded yet.

It is very important to retain the fact that each store is self-contained and can be worked on totally independently. Each Store is a different branch of the search tree and each Worker can work on the branch in parallel with other Workers working on other tree branches.

As only one store is processed in each step (by a worker), the child stores must be stored somewhere. Again, there are two solutions for this: a local (in the worker)

or global (in the controller) data structure.

Global and Local management

Global management (of remaining stores) is implemented with a data structure(e.g. as a list) in the controller. The process:

- The controller launches a number of workers that will be responsible for the traversal of the search tree.
- Starting from the initial store (in the controller's list), the workers compete for the work.
- One of the workers gets the store (the others wait until something is in the work list)
- The worker that "gained the store" executes a traversal step. The resulting stores will be in the list except the one that will the done by "the first worker". The others workers can start to work on the stores in the controller's list.
- The whole process ends when: one of the workers finds a solution (only one solution wanted) or there are no more stores in the list to compute (exhausted search space and all computation is done).

With private management the process is slightly different. There is no global list of stores. Each worker maintains its own list of stores and processes all the stores in this list unless there's an idle worker. When a workers list is empty, it gets a store to process from the busiest worker and proceeds the traversal from there.

A Worker has 3 states: Working, Waiting, Finishing. The transition between states depends on the management policy. Figure 3.3.2 demonstrates this as well as Table 3.3.

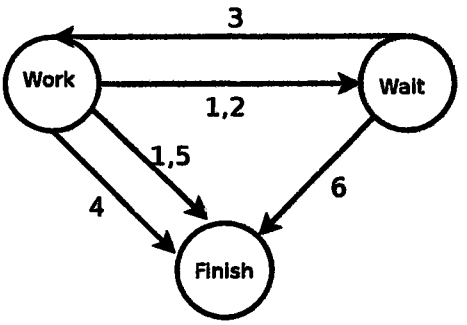

Figure 3.3.2: Worker's state transition diagram

Table 3.3.1: W-worker; C-controller; X-some worker; U-list of workers

Global Management	Private Management
1 - Controller's list is empty	1 - W's list is empty
2 - $\exists X(\neq W) \in U : work(X)$	2- $\exists X(\neq W) \in U : work(X)$
3 - W gets a store from C	3 - W gets a store from some worker
4 - W found a solution	4 - W found a solution
5 - $\forall X(\neq W) \in U, wait(X)$	5 - $\forall X(\neq W) \in U, wait(X)$
6 - C commanded to finish	C commanded to finish

3.4 Summary

AJACS provides an interesting starting point for this thesis' work. AJACS' implementation relied on a distributed shared memory system and special a JVM implementation and was developed in Java. But the implementation details are to be neglected since all is to be developed from scratch. What is important here is AJACS' concepts and its parallel architecture and how they can be adopted to match a multicore architecture like the Cell/B.E.

Clearly relevant is the independence of a Store from the rest the Stores in the search tree which allows an independent and out-of-order treatment by one process. The Store is a piece of work, a block of data to be processed and might match the Synergistic Processor Element (SPE) and its aptitude for data processing. Together with the concept of Global management in which every Worker accesses a global structure to gather work but then works on its own space seems to be a good fit for

Cell's processor heterogeneity as discussed in the following chapter.

Chapter 4

Cell Broadband Engine

This chapter presents the Cell Broadband Engine. Both processor types, PPE and SPE are overviewed as well as related concepts and programming for this architecture.

4.1 Overview

The Cell Broadband Engine (CBE) is the first implementation of the Cell Broadband Engine Architecture. This implementation is a single-chip multiprocessor [18] with nine cores operating on a shared, coherent memory. The nine processors are distinguished in two types: Power Processor Element (PPE) and Synergistic Processor Element (SPE). There is one PPE and 8 SPEs.

The Power Processor Element (PPE) is a 64-bit Power PC Architecture processor. It complies with 64-bit Power PC Architecture and runs 32-bit and 64-bit OS and applications.

The Synergistic Processor Element (SPE [28]) is tailored to run compute-intensive SIMD applications. They are totally independent elements, each able to run their own application program or thread. The access to memory is coherent, including memory-mapped I/O space.

The SPEs provide the application, the performance speedup while the PPE runs the operating systems and usually the main thread of control. Both PPE and SPEs support a rich instruction set that includes SIMD functionality. In the SPEs using SIMD brings great performance advantage or, the other way around, scalar code looses a few cycles since the SPEs always loads and stores a quadword at a time.

The most significant difference between the SPE and PPE lies on how they access the memory. The PPE accesses main storage with load and store instructions that move data between main storage and its registers, the contents of which may be cached. The SPEs, in contrast, access main storage with Direct Memory Access (DMA) commands that move data and instructions between main storage and a private local memory, called Local Store (LS). A SPE fetches its instructions from its LS and has no cache. This 3-tiered organization allows asynchronous DMA transfers from main memory, parallelizing computation and fetching of data (see figure 4.1.1).

Furthermore, the memory latency problem is directly tackled. The few cycles needed to set up a DMA transfer are a much better trade-off compared to the hundreds of cycles of a delayed sequential program with a load instruction on a cache miss. In addition, a SPE can have up to 16 simultaneous DMA transfers, clearly outperforming traditional processors in memory access.

The Element Interconnect Bus (EIB) is the communication path for commands and data between all processor elements on the CBE processor and the on-chip controllers for memory and I/O. The EIB supports full memory- coherent and symmetric multiprocessor (SMP) operations. Thus, a CBE processor is designed to be ganged coherently with other CBE processors to produce a cluster.

The EIB consists of four 16-byte-wide data rings which transfer 128 bytes (one PPE cache line) at a time. Processor elements can drive and receive data simultaneously. The connection order is important to programmers seeking to minimize the latency of transfers on the EIB: latency is a function of the number of connection hops, such

that transfers between adjacent elements have the shortest latencies and transfers between elements separated by six hops have the longest latencies. The EIB's internal maximum bandwidth is 96 bytes per processor-clock cycle. Multiple transfers can be in-process concurrently on each ring, including more than 100 outstanding DMA memory requests between main storage and the SPEs.

Figure 4.1.1 provides a view of the complete processor.

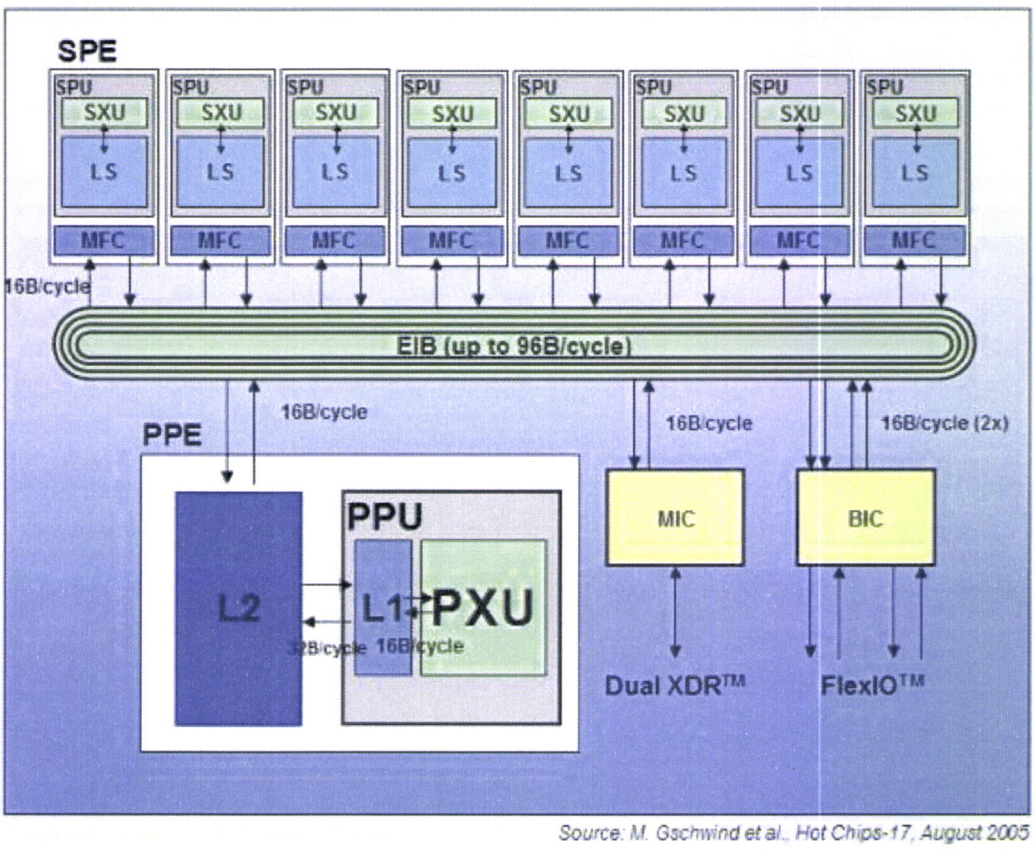

Figure 4.1.1: Cell Broadband Engine

4.2 Power Processor Element - PPE

The PPE consists of a 64-bit, multi-threaded Power Architecture processor with two concurrent hardware threads. The PPE supports the Power Architecture vec-

tor multimedia extensions (Altivec) using SIMD execution units. The processor has a memory subsystem with separate first-level 32-Kilobytes instruction and data caches, and a 512-Kilobytes unified second-level cache.

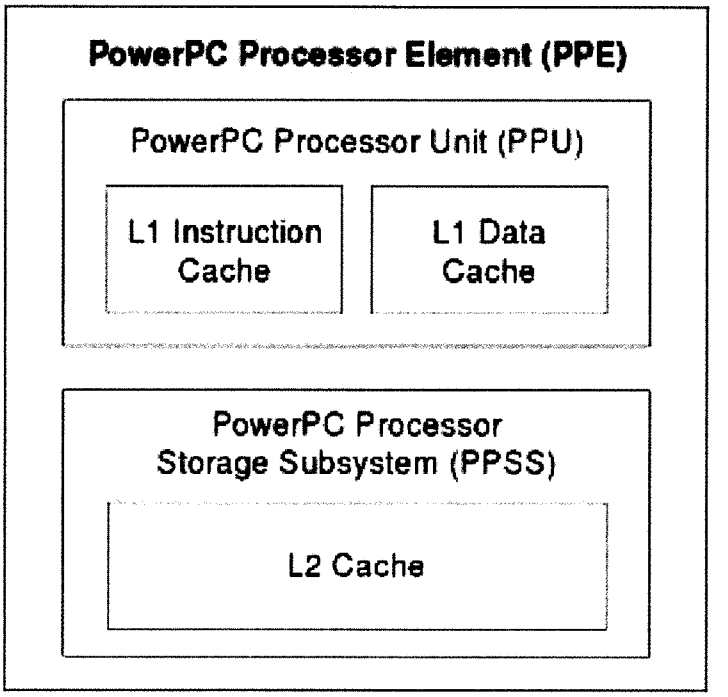

Figure 4.2.1: Power Processor Element

4.3 Synergistic Processor Element - SPE

The eight SPEs provide the computation workhorse in a CBE system. A SPE is a new processor designed to accelerate a wide range of workload by providing an efficient data-parallel architecture and the synergistic Memory Flow Controller (MFC), guaranteeing coherent data transfers from and to main memory. The SPU cannot access main memory directly; it obtains instructions and data from its 256-Kilobyte Local Store (LS) and it must issue DMA commands to the MFC to bring data into the LS or write results back to main memory. In parallel to MFC data transfers, the SPU processes data stored in its private local store.

The local store architecture has simple logic, as cache-hit and coherence logic do not affect the critical memory access operations during load and store operations, allowing faster and more compact implementations. All data accesses with load and store operations refer directly to physical locations within an SPE's local store without further translation.

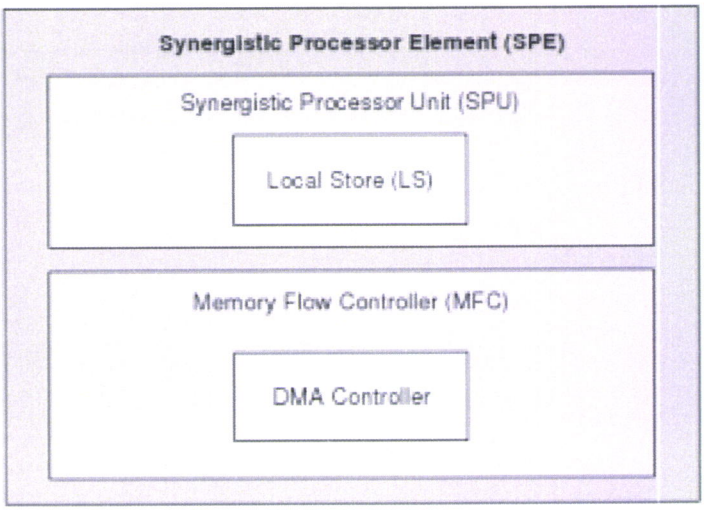

Figure 4.3.1: Synergistic Processor Element

Memory Flow Controller

Each SPE includes a Memory Flow Controller (MFC), which performs data transfers between SPU-local storage and main memory. The access to system memory is supported by a high-performance direct memory access (DMA) for data transfers that can range from a single byte to 16-byte blocks.

A MFC transfer request specifies the local store location as the physical address in the local store and it specifies the system memory address as a Power Architecture virtual address, which the MFC's memory management logic translates to a physical address based on system-wide page tables.

Using the same virtual addresses to specify system memory locations independent

of processor element enables data sharing between threads executing on the PPE and SPE. For example, a PPE-generated pointer can be passed to the SPE which in turn can use it to specify a source or target of a DMA transfer.

4.4 Programming the Cell/B.E.

Structure of a Cell/B.E. application

A CBE application executes in a heterogeneous architecture consisting of PPE and SPE cores. In order to match such heterogeneity, a CBE application consists of two classes of instructions corresponding to each of the architectures [38].

Currently, one CBE application correspondes to a process that can have associated PPE and SPE threads that are dispatched to the correct processor. An application starts with a single PPE thread and control is entirely on the PPE. After the start, this PPE main thread is able to create more threads to execute both on PPE and SPEs, supported by a management library.

The SPE runtime management library (libspe2) is the standardized low-level application programming interface that enables applications access to the SPEs.

Applications do not have control over the physical SPEs. All what applications do is to manage software constructs called **SPE contexts**. These SPE contexts are logical representation of an SPE. The library libspe2 includes additional functions such as transfering application data to and from the SPE's Local Store and initiating the execution of a recently transfered executable.

To be able to use multiple SPEs simultaneously, an application must create at least as many threads as concurrent SPE contexts with support from something like POSIX threads (pthreads). Once an application has initiated the SPE threads, execution can proceed independently and in parallel on PPE and SPE cores.

The Cell Broadband Engine Architecture (CBEA) allows a variety of program-

ming models such as an accelerator model based on remote procedure call, function pipelines and autonomous SPE execution. The simplest is the accelerator model where compute-intensive functions are offloaded to the SPEs. Developers can also compose function pipelines where each SPE executes a set of functions on a data stream and then copies its output to the next pipeline stage implemented on another SPE. Autonomous SPE execution consists of an SPE thread which uses its MFC to independently transfer its input data to the local store and copy result data to the main memory.

Data multi-buffering

To hide memory latency to external memory, data transfers are best performed by each SPE using data multi-buffering like double buffering. With double buffering the SPU operates on one data set in one buffer while the MFC transfers the next data set into the second buffer. This way compute-transfer parallelism is exploited that is, independent SPU execution and MFC data transfer. This is one of the parallelism [19] forms supported by the Cell/B.E.

Application loading and the CESOF format

When starting an application, the OS loads the object file and the execution of the main PPE thread begins. The application then goes by initiating the SPE threads. To accomplish this, the PPE must first transfer the SPE image to an SPE's Local Store. The PPE initiates a transfer of the SPE image by requesting to the SPE's MFC, a transfer from main memory. After the transfer, the PPE issues a request to start the SPU.

To accommodate PPE and SPE programs in one single source file and allow sharing of common variables, the CESOF file format was created.

With CESOF, programmers can achieve some of the effects of linking PPE and

SPE executables. The PPE linker can create a single PPE-ELF executable file that contains code and data for both PPE and SPE processor elements. An OS can load PPE and SPE programs that run concurrently and work cooperatively from an integrated PPE executable image.

Surely all the details related to the structure of the CESOF format are out of this thesis' scope. Nevertheless an understanding of how such a file is created is rather important for understanding some of the frameworks' design considerations.

Figure 4.4.1 illustrates the process.

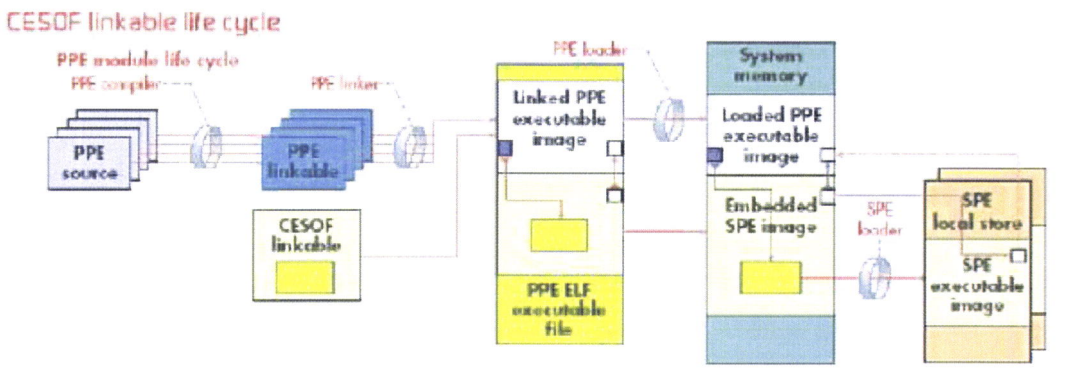

Figure 4.4.1: Cesof file creation

A tool called *ppu-embedspu* wraps an SPE executable file into a CESOF linkable file. The CESOF file contains the image of the original SPE executable plus additional PPE symbol information.

The CESOF linkable, which is itself a PPE linkable, can now be linked with other PPE linkables to form a PPE executable. The PPE executable image contains not only the PPE code modules but also the embedded SPE executable image - a CBE executable. The PPE loader can load the CBE executable including the embedded SPE executable image(s) just like any other PPE executable into the effective address space. From there, an SPE loader can load the SPE executable image into the target local store.

4.5 Summary

The Cell Broadband Engine is an interesting architecture with interesting features, quite different from conventional processors.

The Cell/B.E. is composed of 1 PPE and 8 SPEs connected by an internal high-bandwidth bus (EIB). The PPE runs the operating system and has a controlling role. The SPEs are independent processors, tailored at running compute-intensive tasks.

The heterogeneous nature of the architecture promises orders of performance increase but requires an extra effort from the programmer who has to, for example, coordinate the memory accesses of the SPEs via DMA commands.

Programming for the Cell/B.E. needs to consider some aspects usually not present in a normal programming environment. Specifically and in what concerns directly to this thesis, one should consider:

- how to take advantage of the multiple SPEs. This includes dividing the program across all cores in an effective manner.
- DMA transfers should be SPE-initiated and be overlapped with computation (when possible) to avoid stalls.
- SPE's Local Store has only 256 KB for data and code.
- how to use vector code (SIMD) and large register file whenever possible.
- reduce branching since the SPU assumes sequential instruction flow.
- different instructions sets (PPE and SPE) which implies different sources
- the different address spaces. This is particularly important with references or pointers: a pointer passed from PPE to the SPE can't just be de-referenced but has to involve a DMA transfer.

- memory alignment. LS and main memory addresses must be aligned for DMA transfers (this is particularly cumbersome).
- data sharing and dependencies have to be carefully designed since all processors share the same main memory.

Together with these aspects, programmers must work with a double-toolchain for both PPE and SPE (two instruction sets) as well as the new CESOF file format (described in section 4.4).

The Cell/B.E. architecture definitely pushes some complexity to the hands of the programmer but precisely because of its innovative nature, this architecture presents several concepts that will be seen in future microprocessors. Hence there is an opportunity for exploring and experimenting new tools, programming models and frameworks.

Chapter 5

Design of the framework

The framework developed is presented in this chapter. The framework's architecture is divided in 3 levels: AJACS Level, Cell Level and Application Level. Each level of the system's architecture is explained, focusing on the most important aspects and the decisions made. We then proceed by introducing the hybrid model where the AJACS model was extended to work with local search. The local search method implemented is Adaptive search and this method is also introduced as well as its integration in the system.

5.1 Overview

The work described herein tries to match the current architectural tendency to make parallelism explicitly available to the characteristics of AJACS in order to get a declarative approach to software development in a parallel environment while extracting good performance from such architectures in constraint problem solving.

CASPER *(Cell Adaptive Search and Propagation Engine Resolver)* presents an adaptation of the AJACS model to the C programming language. It is an adequacy study of constraint solving in a heterogeneous multicore architecture as the Cell/B.E. .

The general goals underlying the development of the CASPER system are:

- develop a Constraint Solving System in the C language targeted at the Cell/B.E.
- experiment with this Constraint Solving System
- provide a more declarative programming experience and hide the hardware's complexity and details
- take advantage of the Cell/B.E. particular processing power to solve complex problems

In its organization CASPER aims at producing independent states as result of one ancestor state expansion. The states are independent in the sense that each store (plus the *Problem* containing the constraints themselves) carries all the information necessary to be considered a possible solution for a given problem. With this independence, even the connection to its parent state can be removed.

The state independence is the basis for a parallel execution since in theory, it should be possible to parallelize constraint problem solving by distributing the yet unevaluated states among several processing units without too much foreseen interaction. This way, all processing nodes should be able to 'walk' through the problem space with 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 new stores, i.e. the states that resulted from a successful propagation;
- where and how to signal any solutions that may be found to the problem master controller.

5.2 CASPER

The similarity between the Cell/B.E. architecture and the AJACS model has some striking aspects to it. The same terms are used to name the different entities: controller and worker. In the Cell/B.E., the PPE can be seen as the controller processor while the SPEs are the workers. In AJACS, there is also a controller agent for the problem and several workers who try to find a solution. Therefore it is a natural choice to make the PPE responsible for the master role and the provide the SPEs with the worker role in the AJACS model.

5.3 System Architecture

The developed prototype can be partitioned and understood as a 3-layer architecture. These three layers or levels are a form to comprehend the complete framework.

At the bottom level, there's the **AJACS Level**. The **AJACS Level** implements the AJACS model and its associated concepts. Thus, the *Store*, *Constraint* and the rest of the structures are included in this level.

The middle level is named **Cell Level**. The **Cell Level** hides the Cell's programming complexity and interacts with the AJACS Level to solve a problem. It implements the concept of controller and worker from AJACS' parallel architecture (see chapter 3) to run on the Cell/B.E.. All the architecture's details and mechanisms should be considered part of this level.

Moreover, the **Cell Level** provides an interface to the upper level to allow a parallel execution of the problem solving.

The last and upper level, the **Application Level**, is the prototype's "user level". It represents how the user application needs to be designed in order to interact with the layers below.

Very roughly, this level is the source files which state the problem to be solved and

how they should accomplish this.

Figure 5.3.1 illustrates the architecture:

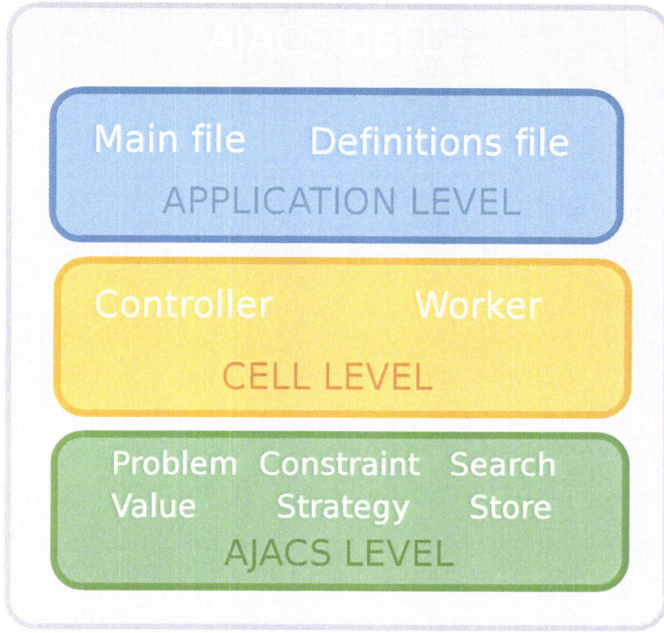

Figure 5.3.1: System architecture

The figure illustrates the overall organization of the prototype. Each level of the architecture is better described in the following sections.

5.4 AJACS Level

The **AJACS Level** implements the concepts present on the AJACS model (see chapter 3 that is, *Problem, Store, Search, Strategy, Value and Constraints.* Implementing all these concepts includes the dependencies between them and the procedures associated to each one like propagation, addition of new values to a Store or defining a new Problem.

The Search Procedure

The Search can be considered the most important aspect of the AJACS Level. The Search is where the real works happens in order to solve the problem and it is this work that's going to keep the SPEs busy. The concepts/entities described before are essentially supported by data structures with associated procedures to operate upon them. Basically, the Search will take all the data stored by AJACS' data structures and find the solution(s) by modifying and replacing this stored data.

In order to implement the Search process, a couple of requirements had to be met:

- First, the need to enforce a small memory footprint in the SPE's Local Store and
- Second, to keep processes as mutually independent as possible.

To meet the design goals a rather simple but effective idea was devised. The idea is to take a store and from this store come up with two complementary sibling stores. From these two sibling stores, the search continues on only one saving its complementary to work on later. This "store mitosis" continues until the search gets to a store which is a solution - it has all variables ground. Figure 5.4.1 provides an example of such "store mitosis".

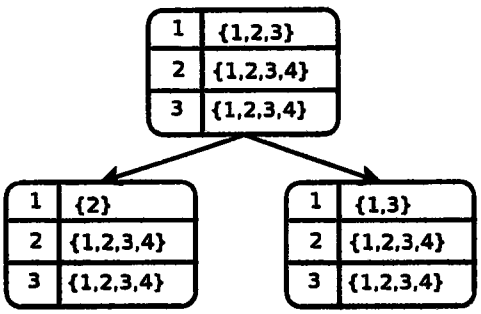

Figure 5.4.1: Stores split

This process guarantees completeness since it will use all possible values for each variable. It also addresses the two aims mentioned above. It keeps a small LS foot-

print by working with only one Store at a time, having a maximum of two Stores in memory by the time of division (where one is transfered to main memory) and then continuing with only one. Finally, the stores are independent from each other: in contrast to a backtracking approach, there are no references to ancestor store. In fact, there are no references to any other Store although there's the implicit relationship with the complementary store but this is just abstract. The complementary Store will be treated as unique and possibly handled by some other worker.

Delving deeper into the search's procedure implementation, one sees that it consists of three steps:

1. **check if the Store is a solution store** that is, if all variables are ground and in affirmative case just terminate and save the solution

2. **produce complementary sibling stores.** By taking the variable being worked on from the current Store, a new value from the variable's domain is selected. With this value, two stores are created: one with the value associated to the variable (the variable is now ground) and another store where the variable has a domain which is complementary to the previous single-element domain. For example, we start with *store1* and the variable X. The variable X as a domain {1,2,3,...,9}. The value 1 is selected from this domain and two stores will be created: *store2* and *store_complementary*. In the *store2*, the variable X has as domain the set 1 - it is ground. On the other hand, *store_complementary* has the variable X with a domain {2,3,...,9} - the complementary one. Note that store1 is no longer useful after this step and its space may be reused for either of the two newly created stores.

3. **do propagation.** To make sure the store we want to keep working on is consistent, we have to perform propagation. Of course the store referred to - the working store - is the one that has now one (more) ground variable. After executing the propagation in the working store three things can occur.

(a) the propagation has failed (at least one variable has an empty domain).

(b) the store turned out to be a solution.

(c) the propagation succeeded but it's no solution.

For both case one and two, the store doesn't allow more progress with it therefore we will proceed the search with the complementary one. Of course, in case we have a solution, it must saved. For the third case, where the propagation succeeded, we continue the search working on the same store and just save the complementary one to be worked on later.

5.5 Cell Level

The **Cell Level** represents the implementation of the *Controller* and *Worker* roles. The *Controller* will run on the PPE and the *Worker* will run on the SPE. Hence, there are one *Controller* and eight *Workers* per Cell/B.E. processor.

The *Controller* sets up the environment and invokes the *Workers* which are responsible for finding solutions to the given *Problem.* Finding solutions includes interacting with the **AJACS Level** by invoking the search procedure (described in section 5.4).

The architecture dependent details like DMA transfers or creation of threads or SPE contexts are implemented at this level in order to hide them from the developer, who should concentrate on the problem to solve and not on architecture or parallelization details.

In the next sections, both the Controller and Worker processes are detailed.

Controller

As already referred, the approach is to have the PPE assume the role of *Controller* role. The *Controller* role can be summarized by the following items:

1. Do an initial expansion of the search tree.
2. Create the SPE contexts.
3. Setup the information to be passed to the SPEs.
4. Create pthreads that manage the contexts.
5. Wait for all to finish.

1. **Do an initial expansion of the search tree.**

   The first step done by the *Controller* is to expand the search tree. By taking the initial *Store*, several sibling stores are created. These sibling stores will be taken by the workers in order to reach a solution.

   The *Controller* is responsible for creating two important data structures. One is the **work list** and the other the **solutions list**. Both lists hold *Stores*.

   The **work list** holds *Stores* which need to be worked on. This is the place where workers will look for and place new work to be done. The expansion of the initial *Store* will place the sibling stores in this list.

   The **solutions list** holds the solution stores found by the workers (if any).

   The first expansion or split of the initial *Store* follows a very simple approach. The idea is to do something similar to what happens with backtracking. By taking the Store's first variable, each of its domain values indicate one sibling Store. Certainly, these new Stores must be consistent. Consistency is verified via propagation.

```
1  For each value in the first variable's domain
2     if value is valid then
3        create a new Store with the first variable ground
4        check for consistency by propagation
5        if store is consistent
6           put it in work list
7        else
8          continue
```

There will be as many sibling Stores as there are correct possible values for the first variable. For example, the initial Store has X as its first variable: $X \in \{1, 2, 3\}$. Following the algorithm the value 1 is taken. It is a valid value therefore a new sibling Store is created and checked for consistency. The Store is consistent with the Problem's constraints so we add it to the work list to be further worked. Now, value 2 is taken and this new sibling Store is also consistent. The work list has now two Stores. Returning to the algorithm's beginning, the value 3 is selected. This time, the sibling Store turns out to be inconsistent (for example, there's a constraint which states that the variable $X < 3$. Thus, the Store is discarded. The expansion is now finished. The workers will have two Stores to start from.

There is a departure from the AJACS model in what concerns the first expansion. In the AJACS model the workers "compete" to get the initial Store and one of the workers does this first expansion whereas in the present case the workers get their configurations prepared by the controller.

2. **Create the SPE contexts**

   Since libspe2 is being used, creating SPE contexts is a required step and definitely the typical scenario when writing Cell/B.E. applications (see section 4.4).

   The number of contexts created depends on a parameter. Essentially, this parameter defines how many workers are required to work to get to a solution. The number of contexts is only limited by the machine's available memory but in our case we want this parameter to be the number of SPEs available (16 in the case of dual Cell/B.E.) so that a worker takes a processor only for itself until the problem is solved.

   Creating the SPE contexts counts as setup overhead since this is a step needed to put the workers working on the problem.

3. **Setup the information to be passed to the SPEs**

The PPE (controller) and the SPEs (workers) must share some data. This data is stored in the main memory to be easily and quickly accessed by all processors. When setting up all the environment, the PPE must provide the location of the common data to the SPEs for these to be able to fetch it via a DMA transfer. The supply is done through a control block holding all the information.

The following description presents the control block:

```
1  struct  block
2  {
3    //problem location
4    unsigned long long problem;
5
6    //solutions number location
7    unsigned long long nsols;
8
9    //list with solutions
10   unsigned long long solution_stores;
11
12   //list with stores to be traversed
13   unsigned long long work_queue;
14
15   //number of stores in work list
16   unsigned long long numb_work;
17
18   //padding for alignment
19   unsigned char pad;
20 };
```

The control block consists of memory addresses from all the data structures set up by the PPE. This is all the information an SPE needs to work on a solution.

4. **Create pthreads that manage the contexts**

As already referred in section 4.4, in order to have concurrent SPE contexts, POSIX threads (pthreads) are used. Basically, each pthread runs a single SPE context or in other words, a Worker. Thus there are as many pthreads as workers and contexts.

Running a context requires performing a synchronous call to the operating system. The calling application (in this case the pthread) blocks until the SPE stops executing and the operating system returns from the system call that invoked the SPE execution. If one wants to use multiple SPEs concurrently then several threads must be created to run several contexts.

As with section 2, the creation of pthreads counts as setup overhead.

5. **Wait for all to finish**

   In the AJACS model, the *Controller* takes an action when a solution is found.

   For now, the Controller waits for all workers to finish, does some cleanup and exits. To wait simply means to wait for the created pthreads to terminate. There is no acknowledgement that a found solution has been found or any form of request for more work on the part of the workers. The workers work as independently as possible and the Controller interacts with them as little as possible.

Worker

The Worker role from the AJACS model is assumed by the SPEs. Each SPE will execute the same functions although working on different data (different Stores).

The Worker is responsible for working on Stores in order to find solutions. Despite the important task, the work done by a Worker can be summarized in three easy steps. Following the same structure used for describing the Controller in section 5.5, here are the steps performed by each worker:

1. Get the control block.
2. Get the problem.
3. Look for solutions.

Get the control block

The worker needs the data locations in order to carry out its process. The control block was already presented in section 3 and all the information it contains. But before accessing the control blocks data, it has to get the control block itself. Hence, the very first step done by the worker needs to be getting the control block with all the information, the one which was setup by the controller/PPE.

Whenever a context is to be run by the PPE, the libspe2's API allows the PPE code to pass an argument to the main function executed by the SPE. This is an easy way to pass initialization arguments to the SPE and a typical method to pass an address of some data to be fetched via DMA from main memory by the SPE.

When the Worker starts executing its code, the location of the control block is already available. The worker just needs to request a new DMA transfer to its MFC, so that the control block with all the information about data locations is made available at the SPEs Local Store.

The function call:

```
1  mfc_get((void *)&ls_b, argp, sizeof(struct block), 31, 0, 0);
```

does exactly this. It means: execute a *get* command to location *ls_b* in the LS from main memory's location *argp* with a size of *sizeof(struct block)*.

Get the Problem

Once the Worker knows the location of the data in main memory, the first thing it fetches is the *Problem* data structure.

The Problem is present in the Local Store throughout the entire lifetime of the worker and, as it holds the constraints, is responsible for propagation, ensuring Store consistency.

Look for solutions

So far the Worker only gathered data from main memory as described in the previous steps. After gathering all the data it needs, the Worker can start performing the search, looking for solutions.

The Worker looks for solutions by utilizing the Search procedure from the AJACS Level. The Search procedure, described in section 5.4, is the entry point from the Cell Level to the AJACS Level.

Now that all data is available, the worker performs more computation by entering a loop:

```
1          while (there's work to be done)
2            dma transfer another store to work on;
3            invoke the search on the transfered store;
4            decrement the amount of work to be done;
```

in which it performs the actual steps in solving the constraint problem.

Each line of the loop is now described:

1 - while (there's work to be done)

One of the informations in the control block is the number of stores to be worked on. As long as the number of stores in the work queue is greater than zero, the worker loops. Now, this value is shared by every worker so the access must be synchronized. The current implementation uses the Cell/B.E. architecture's atomic operations to accomplish this.

The framework implements atomic operations by using the MFC's get- and-reserve-lock-line DMA commands. This special command can be used to implement atomic update primitives on a shared location in system memory. This is a simple method to implement access in shared locations and allows a dynamic number of participants which fits the idea of having a dynamic number of workers working on a problem.

Moreover, the Workers (SPEs) are expected to spend most of the time searching for solutions and hence the number of collisions when trying to access the shared counter is expected to be low.

2 - DMA transfer Store to work on

Before starting, the Worker needs to get a Store to search in. This a normal call to the MFC's get command which requests a transfer from main memory to the SPE's Local Store.

The address from the Stores list is also contained in the control block that was fetched by the Worker. To get the right store from the list only some simple arithmetic is done: $work\_list + (how\_many\_stores\_to\_do - 1)$. This as two direct consequences :

- the workers access this list concurrently without clashing because they all have different values in the how_many_store_to_do ;
- the list of stores to be done starts being processed from the end.

Although the list is shared by all, accessing it can be done without synchronization. The synchronization is only needed to get the list's index from where to fetch the Store in the work list.

Figure 5.5.1 illustrates this. In this example, the SPE gets the number of stores to be done - 2 - from the shared location (the red color means that the access is synchronized) and the Store itself from the work list (the green color means that there's no synchronization needed).

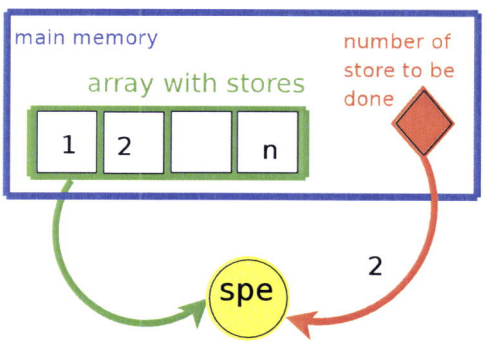

Figure 5.5.1: List index synchronization

Hence, several threads can be accessing the work list and fetching Stores to work on concurrently.

3 - invoke the search from the AJACS level on the transfered store

After transferring the Store, the Worker can start working on it. The Worker calls the search procedure, passing it the recently transfered Store.

In this step, besides the search itself, the transfer of solutions is done. The location where to store solutions was supplied through the control block and the access to main memory follows the same scheme as with the DMA transfer of the store to work on. The location of the variable holding the number of solutions is shared by all the workers and therefore its access must also be synchronized. The number of solutions serves as index to the position in the array of solutions where the worker should save the solution store.

Again figure 5.5.2 illustrates the idea. The similarity between this figure and the previous one intents to demonstrate how similar both actions are. The difference is only on naming and flow direction (note the names and arrows directions).

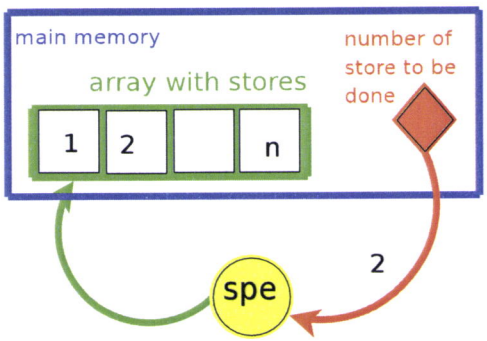

Figure 5.5.2: Solutions index synchronization

4 - decrement the amount of work to be done

Again atomically, access the main memory and decrement the value of the variable which holds the amount of work to be done which controls the Worker's main work loop.

5.6 Application Level

The **Application Level** is the top layer of the architecture. This level does not implement parts of the AJACS model or Cell dependent functions. It simply represents the user program which defines the CSP to be solved.

The two bottom levels (AJACS and Cell) implement the library itself and provide the interface needed for a user to state her problem. Normally, one would not need to mention the user-level when describing a library but there are a couple of peculiarities associated with the CASPER implementation which are definitely worth of examining and exemplifying:

- the eccentric Constraint Programming
- the unconventional Cell architecture

Constraint Programming was already extensively presented in chapter 4. Still, its eccentric, different nature is worth a reminder. When describing a CSP, the programmer's mind set must be undeniably different from the mind set used to program algorithms with current and more mainstream languages like C or Java.

Although the C language is being used, the typical program using the CASPER library is distinct from the usual C program with loops, conditional statements or variables initialization and consists mainly of calls to the library.

One way of illustrating the application-level layer is with an example program. We proceed with such an example.

The best way to describe all this is by showing an example. The following source code is an example of a program. The code is clearly declarative and the source code comments are also included for a complete understanding.

```
int main ()
{
        int N_C=3;
        int c;
        int deb=0;

        Store* sInit;
        Problem* P;           // the Problem
        fdd_value *v1, *v2, *v3;

        Constraint *c1, *c2, *c3;

        Strategy* St;  // the strategy to implement
        Search* Sr;  // the search method
        IntArray* cVx;

        ba_init();
```

This first block of code is the variables' declarations. One might notice the types like Constraint, Search which corresponds to AJACS entities.

```

/*    1) Create values  */

        v1 = new_value(2,8);
        v2 = new_value(3,7);
        v3 = new_value(4,6);
```

Now we created the values v1, v2 and v3. Each one is created with a different domain: for example v1 has a domain of {2,...,8}.

```
/*    1) Create Store and add values  */
        sInit = new_store(3);

        sInit->theValues[0] = *(v1);
        sInit->theValues[1] = *(v2);
        sInit->theValues[2] = *(v3);
```

Having the values we need to create a store and add our values to it.

```
        c = posix_memalign(&c1,16,sizeof(Constraint));
        c = posix_memalign(&c2,16,sizeof(Constraint));
        c = posix_memalign(&c3,16,sizeof(Constraint));
```

```
4
5   // X = Y
6           constraintDefs[0].name = ''X=Y'';
7           constraintDefs[0].nargs = 2;
8           constraintDefs[0].update = &eq_update;
9           c1->constr = constraintDefs[0];
10          c1->env[0] = 0;
11          c1->env[1] = 1;
12
13  //  X <= Y
14          constraintDefs[1].name = ''X<=Y'';
15          constraintDefs[1].nargs = 2;
16          constraintDefs[1].update = &le_update;
17          c2->constr = constraintDefs[1];
18          c2->env[0] = 0;
19          c2->env[1] = 1;
20
21  //  X <= Z
22          constraintDefs[2].name = ''X<=Z'';
23          constraintDefs[2].nargs = 2;
24          constraintDefs[2].update = &le_update;
25          c3->constr = constraintDefs[2];
26          c3->env[0] = 0;
27          c3->env[1] = 2;
```

In the part we defined our constraints. First we allocated aligned space for them using posix_memalign (this is needed because of the DMA transfers between PPE and SPEs). Then we set up all the details of a constraint like its number of arguments or its name.

```
1   /*
2    *    3) Problem
3    *    Add all constraints to the problem
4    */
5
6           P = new_problem(*sInit, N_C);
7
8   // Add constraint 1 to problem
9           cVx = (IntArray*)new_IntArray(3);
10          cVx->arr[0] = (int)c1; // Constraint 1
11          cVx->arr[1] = (int)c2; // Constraint 2
12          cVx->arr[2] = (int)c3; // Constraint 3
13          add_constraint(P,0,c1,cVx);
```

```
14
15  // Add constraint 2 to problem
16          cVx->size=2;
17          cVx->arr[0] = (int)c1;
18          cVx->arr[1] = (int)c2;
19          add_constraint(P,1,c2,cVx);
20
21  // Add constraint 3 to problem
22          cVx->size=1;
23          cVx->arr[0] = (int)c3;
24          add_constraint(P,2,c3,cVx);
```

All the information created so far is needed to create a Problem. We created a Problem, providind it an initial Store (sInit) and the number of constraints (N_C). Then we added each constraint to the Problem.

```
1   /*
2       4) Strategy to implement
3   */
4           St = new_strategy(sInit);
5
6   /*
7       5) Search / Find the solution(s)
8   */
9           Sr = new_search(1); // index to start=1
10          search(P,St,Sr);
11
12
13          return 0;
14  }
```

Finally, we created the default Strategy and a Search. With all the data needed (Problem, Strategy and Search) we invoked the search procedure.

The second pecularity mentioned at the beginning of this section is the unconventional Cell/B.E. architecture. The architecture too was already described in detail in chapter 4. What is more significant here is the fact that the Cell is heterogeneous and as refered before, currently a double-toolchain is used for compiling PPE and SPE programs as it was in fact for the development of this thesis' work.

For some code to run on each processor, it must be compiled with the correct

compiler. The ultimate consequence is that CASPER is composed of SPE and PPE modules since it makes use of both processors.

The **Application Level** must interact with both components, PPE and SPE modules because the user programs consists of code to be run in both processors. For example, the Constraints are functions which must be present to both processors thus compiled two times.

Ideally, the user's source code would be single source, hiding totally the complexity and parallel issues. The current implementation does not allow single source. The user program is always composed of two source files, the main file and the definitions file.

The **main file** states the Problem as seen in the example above and of course, calls the solution finder. Since this file includes the main function (hence the name) it is compiled for the PPE and linked with the PPE modules of the library.

The **definitions file** contains some information that must be present in the SPE (worker) in order for it to work properly. Basically, this information includes the constraints of the problem and some functions related to Adaptive search (this will be clearer in the next section - Extending the search). This **definitions file** contains one *init* function which is called by the Worker. The init function sets up the information needed by the Worker.

Surely, one might argue this is not much different from the existing approach to Cell programming, with two source files, one for the PPE thread and one to run on the SPEs. This is true but all the concerns with parallel design are inexistent. No need to partition data or to synchronize threads. And most of the content present in the definitions file is doubled from the main file and can be copied and easily modified.

With both files (main and definitions) together with the library modules, a binary file is created in the CESOF file format (see figure 4.4.1).

5.7 Extending the search

As referred in 5.4, the search step is the most important. The current *Strategy* takes one *Store* to process and partitions it in two complementary ones. After choosing a variable, the propagation is executed and, recapitulating, three things can happen: a solution, the propagation has succeeded or the propagation has failed. In case a solution was found or with failed propagation, the Search continues with the complementary Store; if the propagation succeeded, then it continues to work on the same Store and puts the complementary one in the work list.

This "*always forward and down in the tree*" approach saves a lot of space (a scarce resource in the SPE's Local Store) since we don't keep any history of performed work or connections to ancestor or any other Store(s) and work only with the current state.

So far, the search is exhaustive which guarantees completeness. But sometimes this is not so important and local search methods provide a very fast way to get a solution.

There are at least two classes of general methods for resolution of constraint problems: complete methods explore the whole search space in order to find all solutions or detect inconsistency and incomplete methods use heuristics to find not all but some solutions. Unfortunately, these methods don't detect inconsistency.

Designing hybrid approaches seems promising since the advantages may be combined into a single approach.

Systematic algorithms for solving CSP typically explore a search tree which is based on the possible values for each of the variables of the solved problem.

The biggest problem of such backtracking-based search algorithms is that they are frequently hindered by innappropriate early choices in the search.

Local search algorithms perform an incomplete exploration of the search space by repairing infeasible complete assignments.

Three categories of hybrid approaches can be found in the literature [23] :

1. performing a local search before or after a systematic search
2. performing a systematic search improved with a local search at some point of the search;
3. performing an overall local search and using systematic search either to select a candidate neighbour or to prune the search space

Adaptive search is an heuristic method in which the key idea of the approach is to take into account the structure of the problem given by the description, and to use in particular variable-based information to design general meta-heuristics.

In our proposal, the propagation-based search is extended with a local search component. At a certain state in the complete search it switches to adaptive search and its heuristic method, by taking the so far grounded variables as constants and a random value from the domain of the non-ground variables as starting points for the search procedure.

5.8 Adaptive Search

Adaptive search [8] is a heuristic (non-complete) method for solving Constraint Satisfaction Problems (CSP). The key idea of the approach is to take into account the structure of the problem given by the CSP description, and to use in particular variable-based information to design general meta-heuristics.

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

The method is not limited to any specific type of constraint but it needs an error function that indicates how much a constraint is violated. Example: an arithmetic constraint $X - Y < C$ will have as error function $max(0, |X - Y| - C)$. The basic idea of this method can be described by 3 steps:

1. compute the error function for each constraint
2. combine for each variable the errors of all constraints in which it appears
3. the variable with the maximum error will be chosen and thus its value will be modified. In this step it uses the well-known Min-Conflict [37] heuristics and select the value in the variable domain that has the best tempting value, that is, the value for which the total error in the next configuration is minimal.

The method also uses an adaptive memory (as in Tabu search [17]) where each variable leading to a local minimum is marked and cannot be chosen for a few iterations.

The Adaptive search method is a generic framework parameterized by 3 components:

- A family of error functions for constraints (one for each type of constraint)
- An operation to combine, for each variable, the errors of all constraints in which it appears
- A cost function for evaluating configurations

Algorithm

this description is closely related to the original paper [8]

Input

Problem given in CSP form

- a set of variables $V = V1, V2, ..., V_n$with associated domains of values
- a set of constraints $C = C1, C2, ..., C_k$with associated error functions
- a combination function to project constraint errors on variables
- a cost function to minimize

Output a sequence of moves (modification of the value of one of the variables) that will lead to a solution of the CSP (configuration where all constraints are satisfied).

Algorithm

```
1  Start from a random assignment of variables in V
2
3  Repeat
4
5        Compute errors of all constraints in C and combine errors on
6        each variable by considering for a given variable only the
7        constraints on which it appears.
8
9        select the variable X (not marked as Tabu) with highest error
10       and evaluate costs of possible moves from X
11
12       if no better move then
13          mark X tabu for a given number of iterations
14
15       else
16          select the best move (min-conflict) and
17          change the value of X accordingly
18
19 Until a solution is found or a maximal number of iterations is reached
```

Some parameters can be introduced in order to control the search, namely for handling restarts. It is possible to parameterize the number of iterations during which a variable should not be modified once it is marked. More, in order to avoid being trapped in local minima, a random reset of a certain number of marked variables is done. Also, like most local search methods, the algorithm has a maximal number of iterations. This means that the main algorithm loops will be executed n times - where n is the maximal number - before it stops.

Integrating the Adaptive Search

In the whole architecture

The Adaptive Search was implemented as an extra module. Recurring to the described System Architecture, the Adaptive Search module is situated at the same level as the AJACS Level. It is used by the Cell Level, concretely by the Worker, to perform the search.

The Adaptive Search module itself is also a library that can be linked with when creating the binary file. Generally, this module implements the algorithm described above in section 5.8 but it is tied to the Store object and considers the work done previously by the propagation-based search.

The need for the connection with the Store object is to take into consideration the work performed before. The Store is the state of how things are, which variables are ground and which domains do they have. These two aspects, ground variables and variables' domains are what the "slightly modified Adaptive Search" needs from the Store.

To take advantage of the work done before, our implementation of Adaptive Search starts by marking the already ground variables (made ground from the complete search) as untouchable variables. Only the non-ground variables will have their values being worked on according to Adaptive Search's algorithm. This is the first place where the Store is needed.

The second place where the Store is needed is when selecting the min-conflict value for a variable. Only values which are in the variable's domain are checked.

The Adaptive search algorithm depends on user-provided functions which model the problem. The `cost_on_variable` functions determines the error for one variable and associated constraints. The `cost_of_solution` function combines all variables errors and should be equal to zero when a solution is found.

Since these functions are provided by the user they can be said to belong to the Application Level. The following reasoning can be made: the functions help to do search which in turn is done by the Worker, the entity which runs on the SPE. The SPE code is defined by the definitions file therefore these functions must be included in it. The need for these functions still fits the two-files model.

In the Worker

The search process is now hybrid. At a certain point, the search switches from the complete propagation-based search method to Adaptive search. The decision of when to switch methods is managed by the Worker, running in an SPE.

So far the Worker's work loop was only finding solutions via the propagation-based exhaustive method. It would get a Store and perform the Search as described before. Now the exhaustive method is only carried out until a certain condition is met, this can be for example "do complete search until n variables are ground" or "do complete search for n iterations".

When the condition is met, the Worker calls the search from the Adaptive Search module, passing it the current Store as the state of current solution finding.

Naturally, if the heuristic search returns a solution then a DMA transfer of the solution is done, saving the solution in main memory as was done in the complete search method.

5.9 Comparison with other work

Constraint programming has been around for quite some time so it is natural that a lot of work has been put into this area. We already refered to some existing work. Here we focus on recent work for doing a comparison: Gecode, IlogCP, Minion, Comet and Choco.

Gecode is a library for developing constraint-based systems and applications. It is implemented in C++ but has interfaces for several other programming languages like Java (GecodeJ) and Ruby (GecodeR). Gecode has a very small kernel (in terms of lines of code) and is reported to be very fast. It allows some modelling (although not being a primary target) and has plans for parallel execution but with no work on this so far. The search on Gecode is based on recomputation and is has different standard search engines (e.g. depth-first search, limited discrepancy search, etc.).

ILOG CP is a C++ library that embodies Constraint Logic Programming (CLP) concepts such as logical variables, incremental constraint satisfaction and backtracking. It is a commercial product with extensive documentation and debugging tools, implementing much more different constraints and search methods. Is offers professional supports but it is closed source and with few techniques published.

Choco is an open-source Java library for constraint programming and explanation-based constraint solving (e-CP). It is built on a event-based propagation mechanism with backtrackable structures. The implementation is opensource.

Minion [20] is a general-purpose constraint solver, with an input language based on the common constraint modelling device of matrix models. It aims at being a black-box providing few options to the user, arguing that the increasing complexity of today's toolkits for constraints has heavy costs in terms of performance and usability. This constraint solver is also implemented in C++ and focuses on a highly-optimised implementation, exploiting the properties of modern processors.

Comet [29] is an object-oriented language supporting a constraint-based architecture for neighborhood search. *The main message is that, although they support fundamentally different types of algorithms, constraint programming and Comet share a common architecture which promotes modularity, compositionality, reuse, and separation of concerns.*

CASPER is a work in progress and will certainly be subject of several modifications and improvements. Still several characteristics allows some comparison with existing

systems.

An evident characteristic of CASPER is its implementation targetting the Cell/B.E.. This is rather new since there are no implementations (at least known of) doing the same. Minion aims at exploiting some properties of modern processors (e.g. cache) but no architecture in particular and this in fact proves to be of value as Minion is reported to perform better than Ilog CP and Gecode for some problems.

The implementation of CASPER is done in the C language, a language not typically used by current implementations like Ilog, Minion and Gecode. The C language is the best supported language in Cell/B.E. and therefore this was a natural choice.

Another evident characteristic from CASPER is its parallel architecture inherited from AJACS. Gecode has plans for parallel search but none of the other toolkits have parallelization has a main target like CASPER.

The search is very customizable is most toolkits (Gecode, Ilog CP and Choco) as well as in Comet, allowing the definition of new search procedures, retrieving of one, some or all solutions or limiting the search space. Unfortunately, CASPER doesn't allow this level of customization and is very static mainly due to its prototypal nature. All the toolkits work with systematic search and doesn't seem to be any references to local search or even hybrid schemes. The exception is Comet (one of the reasons why it is in this list of related work) which has abstractions for the specificities of hybridizations between systematic and hybrid search.

One characteristic present in some of the presented frameworks such as Gecode or Minion is the support for modelling. Gecode, for example, supports regular expressions for extensional constraints and expressing linear and Boolean constraints in the standard way as expressions build from numbers and operators.

5.10 Summary

CASPER is an adaptation of the AJACS model to the C programming language and developed to run on and take advantage of the Cell/B.E. while hidding the complexity of this architecture.

There are some similarities between the Cell/B.E. and the AJACS model. Both have a controller, responsible for management of work and workers that are the real "workforce".

The architecture of CASPER is composed of 3 levels (figure 5.3.1): the AJACS, the Cell and the Application levels.

The bottom level is the **AJACS Level**. It implements the AJACS model and its associated concepts such as *Store* or *Constraint*. Particularly important in this level is the *Search* procedure, which works on independent states to achieve parallel execution and by keeping a minimum of Stores in memory to guarantee a small memory footprint to avoid overflow of the SPE's small Local Store.

The middle level is the **Cell Level**. This level hides the Cell/B.E. programming complexity and interacts with the AJACS Level - calling the search function - to solve a Problem. To this level belong the Controller and Worker concepts implemented for the PPE and SPE, respectively.

The third and upper level, named **Application Level**, represents the user application stating solely the problem to be solved and calling the procedure responsible for the whole problem solving.

Sometimes one does not require completeness or only needs to find quickly one acceptable solution, provided by a local search method. CASPER extends the search by implementing an hybrid approach, combining propagation and local search. The local search algorithm choosen was Adaptive search, an heuristic method in which the key idea of the approach is to take into account the structure of the problem and use problem-oriented and variable-based information to design general meta-

heuristics. The choice has fallen on this algorithm due to its simplicity and good performance.

The integration of the Adaptive search method consists of creating a "jump point" where the switch from propagation-based search to local search is done. When it starts, the adaptive search algorithm considers the work done previously by the complete search. For example, if one variable is already ground, its value won't be changed since it is already consistent.

A quick look and comparison to some existing work (Gecode, Ilog CP, Choco, Minion and Comet) provides a good manner to position and assess CASPER's features. CASPER possesses some unique characteristics when compared with other systems such as the implementation language, the parallel architecture and the hybrid search. Also, it shows that some work can be done in the design in order to allow more customization, dynamism and modelling support.

Chapter 6

Experimental evaluation

In order to get some feedback on the behaviour of our framework, we conducted an initial performance assessment.

The evaluation centers in classical problems used as tests to assess CSP solving. The used test programs are toy-problems but should be enough to give a preliminary idea of how does the framework behaves. These test programs are detailed in the next sections.

Performance measurement here means the wall-clock time needed to run a test program. The wall-clock time is obtained with the Unix utility *time*.

6.1 Hardware and Software environment

The hardware and software environment used is summarized in the two tables

Table 6.1.1: Hardware environment

CPU	Dual Cell system (QS20)
PPE	64-bit dual-threaded
PPE Caches	L1 - 32KB, L2 - 512KB
SPE LS	256 KB
SPE Cache	No cache
Filesystem type	ext3
Memory (RAM)	1024 (512 MB for each Cell/B.E.)

Table 6.1.2: Software environment

Operating System	Linux 2.6.20
Distribuition	Fedora Core 6
PPE compiler	ppu-gcc
Version	4.1.1
SPE compiler	spu-gcc
Version	4.1.1
PPE compiler flags	-O2 -ftree-vectorize
SPE compiler flags	-O2 -ftree-vectorize

6.2 Test programs

N-Queens

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

The problem consists of placing N queens on a chessboard so that it's not possible for a queen to attack one other one on the board. This means no pair of queens can't share a row, a column nor a diagonal and that these are our constraints.

Modelling the problem is then:

- N variables , implicitly queen i is on line i
- each variable with a domain {1,2,...,N}

- the constraints: all variables with a different value (not in same column) and no two queens in the same diagonal.

SEND+MORE=MONEY

The *SEND+MORE=MONEY* is another classical example, usually used to demonstrate and test CSP solvers. This problem consists on assigning a distinct digit to each letter {S,E,N,D,M,O,R,Y} a value so that the equation holds. Also, the letters S and M must be different from 0 (no leading zeros).

The model for this toy problem is therefore straightforward:

- 8 variables (one for each letter)
- each variable with a domain {0,1,2,...,9}
- the constraints: the equation must hold and S and M must have a value different from 0.

Golomb Rulers

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

The model used for this problem is:

- one variable for each mark used (7)
- each variable with a domain {0,1,2,...,25} which is the optimal ruler size for the number of marks

- constraints used: the first variable must be zero, the distances between any two marks are distinct and the variables value is incremental that is, $X_1 < X_2 < ... < X_m$

6.3 Tests results

In this section, we present the experimental results. For each test program, several performance measurements are shown and, wherever necessary, commented on. General conclusions will be drawn in section 6.4.

For all tests the following scenarios used were: With adaptive search (With adaptive) and Without adaptive search (Without adaptive). Both situations were considered using compiler optimization (With Optimization) and no compiler optimization at all (No Optimization). All combinations were done for 1, 2, 4, 8 and 16 Workers.

Queens

The overhead measured for all three Queens tests are shown in table 6.3.1. The results are stable (increased overhead when numbers of workers increases) and very low.

Table 6.3.1: Queens Overhead

Number of workers	1	2	4	8	16
Overhead Queens 4	0.00154	0.002595	0.00483	0.008143	0.022917
Overhead Queens 6	0.001734	0.002819	0.004998	0.011433	0.022045
Overhead Queens 8	0.00167	0.00273	0.00493	0.11437	0.02231

The table 6.3.2 shows the results obtained for all three Queens tests.

In all Queens' tests, the results from Queens4 are the most different. Graphic 6.3.1 illustrates the results for Queens 4. The performance decreases as the number of

Table 6.3.2: Queens results

Number of workers			1	2	4	8	16
Queens 4	No Optimization	With adaptive	0.008	0.09	0.012	0.018	0.030
		Without adaptive	0.009	0.010	0.013	0.017	0.030
	With Optimization	With adaptive	0.017	0.015	0.015	0.018	0.029
		Without adaptive	0.008	0.009	0.012	0.017	0.028
Queens 6	No Optimization	With adaptive	0.255	0.130	0.081	0.051	0.0440
		Without adaptive	0.059	0.035	0.025	0.023	0.031
	With Optimization	With adaptive	0.090	0.049	0.034	0.027	0.029
		Without adaptive	0.031	0.022	0.017	0.019	0.030
Queens 8	No Optimization	With adaptive	1.149	0.610	0.380	0.220	0.130
		Without adaptive	1.575	0.797	0.404	0.232	0.137
	With Optimization	With adaptive	0.369	0.196	0.114	0.072	0.053
		Without adaptive	0.653	0.336	0.173	0.102	0.067

Workers is augmented except for the scenario with adaptive search and no optimization where the performance is better increasing the number of Workers up to 4. Surprisingly, the scenarios with adaptive search were slower than with complete search.

With Queens 6 everything starts looking more interesting. Looking at 6.3.2, in all scenarios the performance is better as more workers are added. Still, as with Queens4, the adaptive search behaves poorer than the complete search. Also the difference between optimized and non-optimized code is particularly noticeable when using adaptive search.

The Queens 8 test (figure 6.3.3) also behaves better as the numbers of Workers is increased. This is particularly clearer up to four Workers. The difference between optimized and non-optimized code is the greatest so far.

In contrast with the two previous Queens, Queens8 is faster using adaptive again specially with 1, 2 and 4 Workers and then takes almost the same time with and without adaptive search.

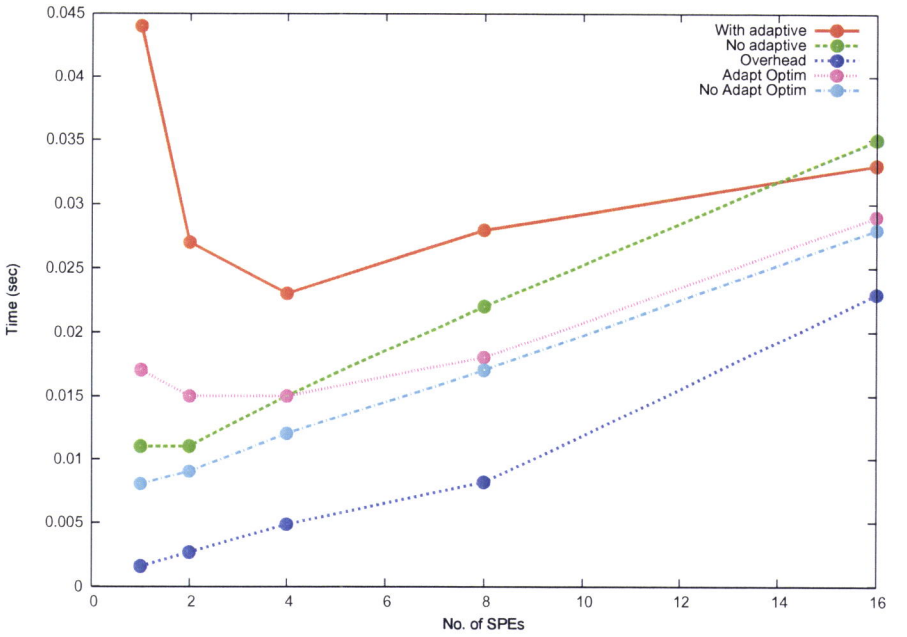

Figure 6.3.1: Queens4 plot

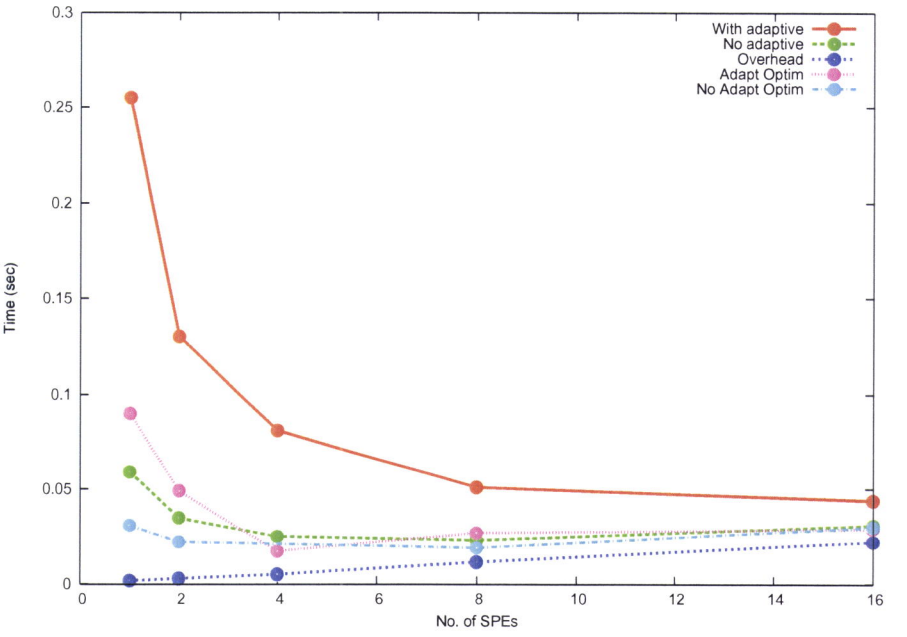

Figure 6.3.2: Queens6 plot

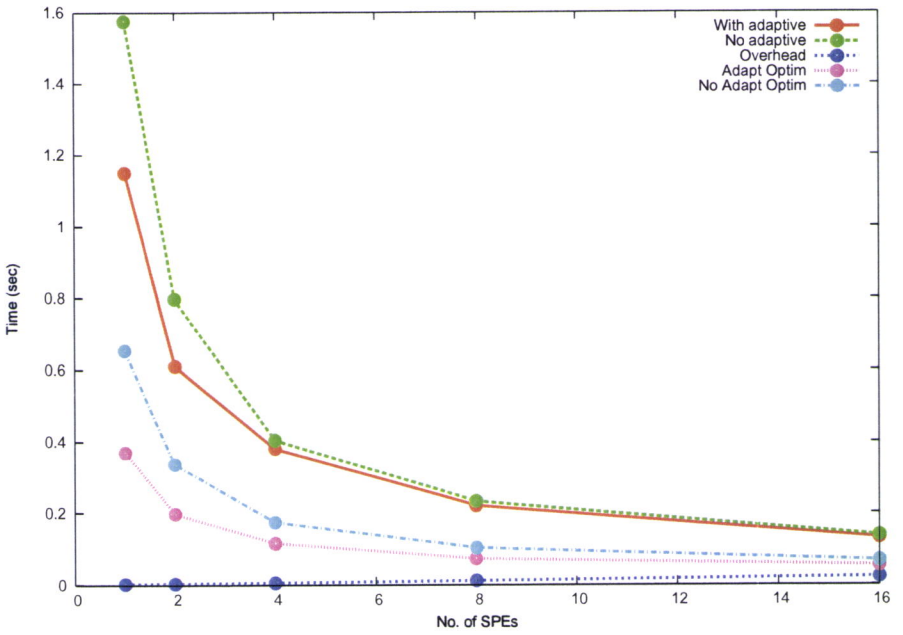

Figure 6.3.3: Queens8 plot

SendMoreMoney

The Overhead continues very similar across all tests, including with the $SEND + MORE = MONEY$ test. Table 6.3.3 presents the overhead values.

Table 6.3.3: SEND+MORE=MONEY Overhead

Number of workers	1	2	4	8	16
Overhead	0.001977	0.003006	0.005298	0.010726	0.022625

The timing results for this test show nice and encouraging results considering the size of this problem when compared with the other tests. The results obtained provide meet the initial expectations. The solution (this test only has one) is obtained much faster when more workers are involved and much faster when using the hybrid method (from 2.163 to 0.059 secs).

Figure 6.3.4 illustrates the results obtained. In all scenarios there is an improvement every time more workers are added and the hybrid search always behaves better than the complete search.

Table 6.3.4: SEND+MORE=MONEY Results

Number of workers		1	2	4	8	16
No Optimization	With adaptive	0.517	0.265	0.145	0.105	0.059
	Without adaptive	2.163	1.087	0.551	0.283	0.158
With Optimization	With adaptive	0.161	0.086	0.051	0.035	0.033
	Without adaptive	1.097	0.573	0.351	0.211	0.096

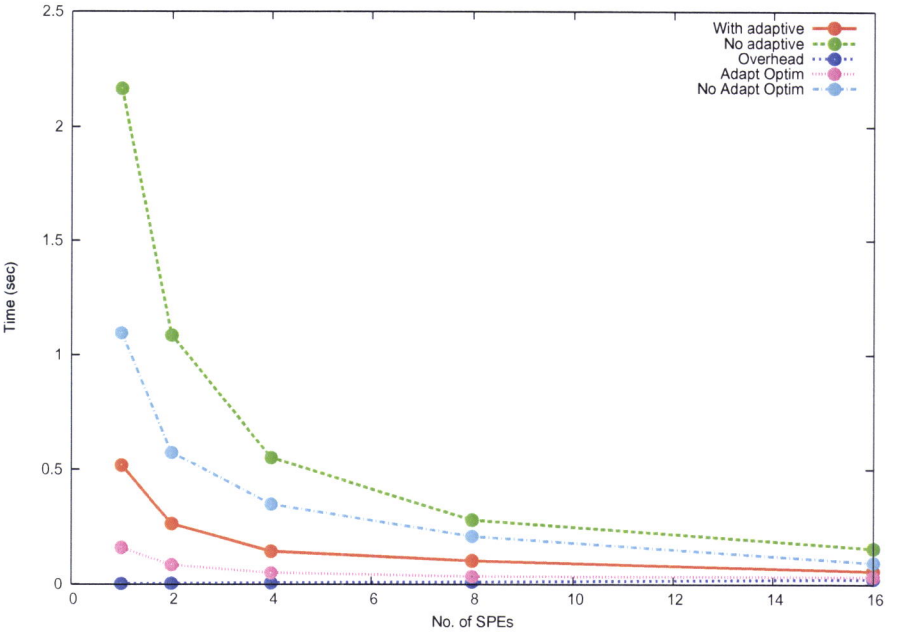

Figure 6.3.4: Money plot

Golomb Rulers

The table 6.3 shows the measured overhead for the Golomb Ruler test. They are slightly different from the previous tests specially with less workers but the biggest overhead (with 16 workers) is not much greater than before in spite of having much greater complexity.

Table 6.3.5: Golomb Ruler Overhead

Number of workers	1	2	4	8	16
Overhead	0.00334	0.004303	0.006842	0.0129	0.024491

Table 6.3 presents the measurements obtained for the Golomb Ruler test. The

parallelization increases the performance but only without adaptive search. With the hybrid approach (with adaptive search), the results are disappointing as if there was no parallelization at all.

Table 6.3.6: Golomb Ruler Results

Number of workers		1	2	4	8	16
No Optimization	With adaptive	25.55	37.369	40.132	32.892	40.767
	Without adaptive	12.977	6.495	2.891	1.454	0.885
With Optimization	With adaptive	6.033	4.349	7.127	4.877	6.294
	Without adaptive	6.353	3.183	1.577	0.810	0.413

Without adaptive search, the performance increase gained by adding more workers is visible: from almost thirteen seconds with one worker (12.977) to less than a second with sixteen workers (0.885). Here again, the optimized code reduces the time needed to the half of non-optimized code, doing 0.413 seconds with sixteen workers.

Figure 6.3.5 illustrates the results obtained excluding the scenario with adaptive and no optimization.

The results obtained with the hybrid approach are totally disappointing. In this test, the hybrid approach takes practically always the same time to complete the task, no matter how many workers are participating. With no code optimization, the time needed by the hybrid approach is very large when compared with the rest of the scenarios. Figure 6.3.6 illustrates the results obtained in all scenarios.

6.4 Results interpretation

In general the results are encouraging but not excellent. Of course it is an evolving prototype and performance results obtained are important only to show a direction and are not at all definitive.

After running the tests and extracting some performance information, three overall conclusions can be safely drawn:

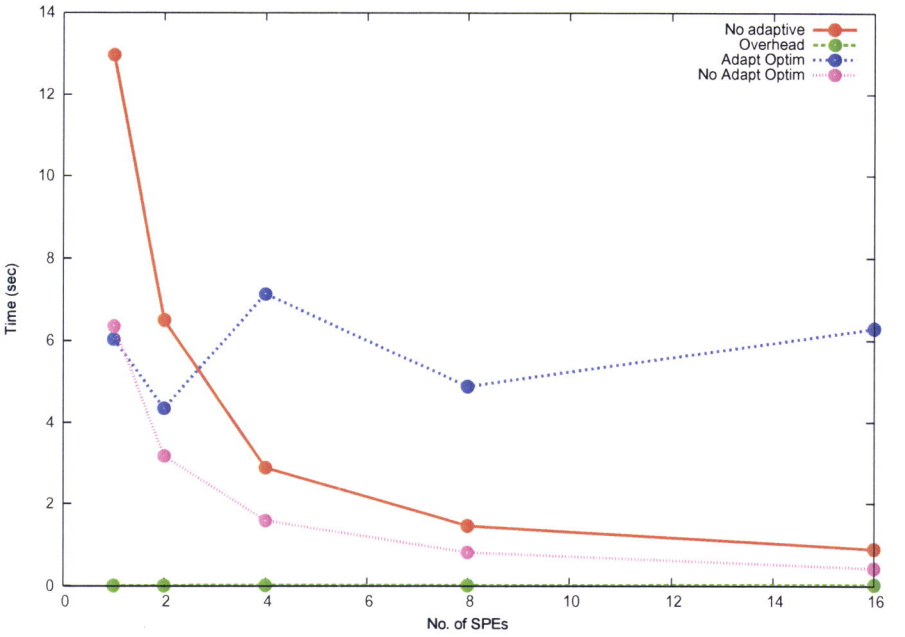

Figure 6.3.5: Modified Golomb Ruler plot

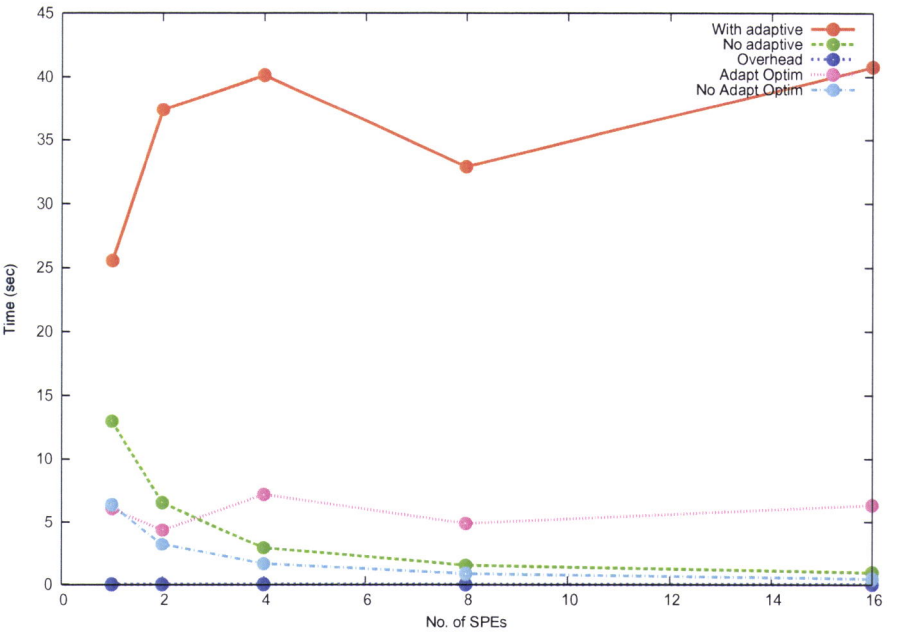

Figure 6.3.6: Golomb Ruler plot

- adaptive search and the hybrid scheme need more tuning since by including heuristics, the performance depends on their quality
- adding more workers increases performance
- there's still some space for optimization

All three Queens tests present different results. In **Queens4**, up to four Workers, there are two cases where the performance increases: both with adaptive search with and without compiler optimization. The performance increase is considerable from 1 to 2 Workers with no code optimization. More Workers means more time needed due to Overhead increase. This means although the problem is a bit complex its size is limiting. After fours Workers (or two in some scenarios) , the overhead takes control and the program is slower. With optimization from the compiler, the difference from one to four workers is not so evident but is not so surprising since the program is working between 1.5 and 2 hundredths of a second. A little surprisingly, running with adaptive search is slower than complete search.

Queens6 behaves much better and takes more advantage of the parallelization. In this test, most of the scenarios take advantage up to eight Workers and adaptive without optimization even benefits from all sixteen Workers. But here again, adaptive search performs worse than complete search.

From all Queens tests, **Queens8** is the biggest and the one which presents results closer to what was expected. It fully takes advantage of parallelization, with the program increasing its performance as more Workers are added and the adaptive search is also, though slightly, faster than the complete search.

Since all three Queens tests solve exactly the same problem, the variations must depend on the size of the Problem which is where the tests differ. The better results are due to bigger problem size where are workers work on some Store and don't just start and finish without performing any real work.

The other difference has to do with adaptive search. Adaptive search performs better

with Queens8 and worse with Queens4 and Queens6. Both sides have something in common: the number of solutions. Queens8 is a problem with many solutions whereas Queens4 and Queens6 have less solutions (two and four respectively). When looking for many solutions, complete search takes more time to finish and adaptive search has the advantage since it will only return one solution for each store it starts with. With less solutions, complete search is faster acknowledging when there are no more solutions left while adaptive search tries to find a solution where there's none until its iteration limit is consumed.

The **SEND+MORE+MONEY** test corroborates what was seen so far with the previous tests and adds some extra information. This test is bigger and thus requires more work. The time needed by one worker is now in the order of seconds.

As the number of workers is augmented, the time needed to get to a solution (the only one is this case) diminishes, with a good speedup. This happens in all scenarios.

The SEND+MORE+MONEY test implements the equation constraint in a inefficient way. The propagation always succeeds until the last instant that is, when all are ground but incorrect then propagation fails. This turns propagation-based search much slower and in fact it is slower when compared with adaptive search scenarios. Before, with the Queens tests, whenever a problem had less solutions, adaptive search was slower than complete search because of unnecessary work in Stores without solution. This time this doesn't happen, although SEND+MORE+MONEY only has one solution. This leads to the conclusion that adaptive search's current implementation is not as fast as it could. The only reason why adaptive search is faster is because complete search works with a very inefficient constraint. Moreover and to verify this, a change in adaptive was made. By increasing the limit number of iterations, the degradation on the hybrid approach was more obvious with adaptive search trapped too often in local minima cases and repeatedly re-doing work.

Finally, the Golomb Ruler test. In what concerns results interpretation, the Golomb ruler provides information in two scopes: it agrees with was concluded so far for

some parts and adds some more information to better understand the frameworks' behaviour.

On one hand, the complete search behaves well and as expected, agreeing with what was seen so far since there is a performance increase every time more workers are added. This test program, by requiring much more time to complete - around 13 seconds with one worker - provides a better view of the benefit by parallelizing the problem solving, needing less than one second with 16 Workers. Of course, the time needed could be much less since a Golomb ruler with 7 marks is still a relatively small problem but once again, the results are encouraging.

On the other side, the program shows deceiving performance when the hybrid approach is used. Not only the time needed to reach only one solution is extremely larger than the time needed by complete search, also the parallelization has no effect. The adaptive search performs practically the same way with any number of Workers or better, it always performs like having only one Worker. And in fact that is what is happening. The first Worker to get a Store is the only one which gets a Store to work on. All other just start and finish because there is no work in the work list. The first split on the search tree done by the Controller only puts one Store in the work list, in contrast with all the previous tests, where the work list was initially more populated.

The reason for this poor performance is the conjunction of 3 factors: the problem's nature, the adaptive search's implementation and the current work-flow of a Worker. First the problem's nature: the problem starts with a search tree with only one branch so there will be only one to be put in the work list. Also the propagation fails very often because of the problem's tight constraints therefore having less Stores transfered to the work list. As already mentioned, the adaptive search needs work and in the Golomb Ruler's test this is evident. It takes to much time to find a solution and sometimes doesn't find one. Finally and more importantly, the Worker's behaviour. As explained in chapter 5, the Worker loops until there's work to be

done. Since the work list starts almost empty, all Workers except the first exit without performing work. This is why parallelization has no effect and the program behaves always like it has only one Worker.

In summary, the overall results from the different tests are similar. The first observation is to the test's size. Although they provide insight on the framework's behaviour, they revealed themselves small for taking advantage of the parallelization among sixteen workers. The positive side is that the framework performs better than initially expected. With Queens8, SEND+MORE+MONEY and specially Golomb Ruler, the effects of the parallelization are more visible.

The hybrid model is still a work in progress. The current implementation of the adaptive search still needs some work in order to obtain performance gains such as those reported on in [8]. This can be seen in the Queens4, Queens6 and definitely in the Golomb Ruler as when increasing the iterations' limit. It gives less solutions and takes more time.

The Overhead is pretty constant among all different tests with a slight fluctuation with optimized and non-optimized code.

One sign for possible future performance increase is given by the use of optimization flags and comparing its results with non-optimized code. As mentioned before, Cell specific code is needed to get more performance and some optimization done by the compiler can give a preview of a performance gain window. The difference between optimized and non-optimized code is definitely visible. Needless to say that compiler's optimization are different from the optimizations done in code by the programmer but what it means is that there is still space to better and faster code.

A last point that is worth a comment relates to the experience gained with developing and doing the evaluation described in this chapter. More specifically, the difference in performance between using debugging output and not using it. When using debugging output - which happens most of the time when one is developing a

prototype - the execution time increases in a great factor. This happens due to the fact that each request for I/O from the SPE must be handled by the PPE thread. Although we knew that I/O is handled this way, we did not know that it would lead to such a great difference. Thus, most of the development time was done with a wrong and only late cleared assumption. Fortunately it lead to better performace measurements after removing the debugging output but nevertheless we worked with false assumptions and that was one lesson learned from this experimentation.

Chapter 7

Conclusion

In this thesis CASPER was presented, a Parallel Hybrid Constraint Programming Library.

Today's hardware tendency is to go multicore, progressively making end-user computers similar to high-performance and scientific machines.

More than solving hardware limitations, this shift in computing has a strong software impact. Programmers have now several cores at their disposition which can possibly increase their application's performance. The catch is that taking advantage of this performance increase requires complex changes to the software structure, which needs to be explicitly aware of the performance-motivated underlying hardware architectural changes.

The need of research for new methods and models that are suitable for this hardware change is high and industry as well as academia are heavily focusing on this.

This thesis' work looks exactly at this hardware tendency and one particular programming paradigm, Constraint Programming, which is a high-level and declarative programming approach where programs are stated as a series of relations (constraints) between variables.

The work starts by two given points: the Cell/B.E. and the AJACS model. The

Cell/B.E. is a very innovative architecture that reveals much of the characteristics of future architectures and includes itself in the current multicore trend. On the other hand, the AJACS model for constraint programming already proved some interesting results in a distributed environment.

CASPER results from matching these two points and by extending it with local search capabilities. It is an experimentation of how well both work together. As result a prototype was developed.

Despite the prototype's early state of development, some tests were developed to initially assess the viability of the whole framework. The tests results look promising and point at some issues to be dealt with. The AJACS model is interesting and suits the Cell/B.E. architecture, specially in what concerns to the model's parallel execution architecture where controller and workers fits nicely with PPE and SPEs.

The results are not yet excellent but also not frustrating. They provide a good incentive for further work. Some tests are small to take full advantage of parallelization but the bigger ones already provide some good results. They take advantage of the parallel work done by the workers and faster results with the hybrid scheme in some scenarios.

7.1 Future work

The present design and implementation of the prototype is effectively a work in progress. It is the result of experimentation and the starting point for the investigation of constraint programming capabilities and limits in a novel architecture.

Naturally, the current codebase will be subject of modification in order to address current limitations.

This first implementation is very "naive", making use of static structures and some hard-coded values. This has obviously to change since it restricts the variety of problems to solve.

Also related to this static nature, the currently used data structures might suffer changes, reducing dependencies and memory footprint - specially for the SPEs. It might depart from the initial AJACS model and evolve on itself. More analysis and options have be produced.

The current implementation is simple. More work on optimizing the code must be done, more importantly on the SPE side, using wherever possible SIMD code and reducing branching as the two most striking improvements. The objective, together with performance, is to reduce the SPE code-size in the LS. Although this is very architecture dependent it will be worthy when trying to solve more real and heavy problems.

The algorithms too, might benefit from some redesign namely the algorithm for Adaptive search which still iterates too much and requires extensive individual tuning in some cases in order to extract good speedups.

It is also an objective to enhance the declarativiness of user-programs by providing a richer API, possibly including a language pre-processor to provide a measure of syntact sugaring. The whole point is to describe problems and its constraints as well as to extend the solver in a very simple way, hiding hardware complexity and control.

Besides implementation details and improvements, the design still has issues which are worth further experimentation and are interesting for new revision and extension.

Although thought of since the beginning, **single-source** was not accomplished. This is particularly relevant for programming propagators and other constraint procedures, which must be usable in both kinds of context, controller and workers. This may well require the development of a tool responsible for a pre-processing phase, which then feeds the different compilers.

A more radical design change passes by **differentiating SPEs** responsibilities. Currently, each worker (from the AJACS model), running on an SPE carries out the same kind of work. It would be interesting to have workers with different roles, like

selector and propagator, where each would do a simpler task instead of the whole procedure as presently. They would communicate between each other, creating a pipeline and exploit the Cell/B.E. inner bus (EIB), which has very high bandwidth, much greater than memory access. This redesign would address several problems:

- locks: there would be no need to synchronize the access to the indexes via atomic operations, a current bottleneck.
- access to memory: reduce greatly the number of accesses to main memory which are slower that communication between processor elements
- space used in the LS: reduce each SPEs code size would save important LS space for data
- simplify the code: simpler code leads to less bugs and is easier to optimize

This design is currently being reasoned to put into practice.

Another point, is the class of problems that can be modelled using the framework i.e. whether the problems which can fit into SPEs have a sufficiently complex processing associated with them to result in a significant performance gain for the overall constraint solving goal. This includes developing more and increasingly complex tests and look at real problems which would benefit from the performance gain.

A more long-term line would be looking at networks of Cell/B.E. blades, creating another layer. There are already controllers and workers and this can be extended to teams or departments, in a distributed memory model.

Bibliography

[1] M. Rinard A. Saraswat and P. Panangaden. Semantic foundations of concurrent constraint programming, 1990.

[2] Gabriele Jost Barbara Chapman and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press, 2007.

[3] Roman Barták. Online guide to constraint programming. http://ktiml.mff.cuni.cz/ bartak/.

[4] Roman Barták. Constraint programming: In pursuit of the holy grail. *Proceedings of the Week of Doctoral Students*, pages 555–564, June 1999.

[5] David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

[6] Constraint handling rules website. http://www.cs.kuleuven.be/ dtai/projects/CHR/.

[7] Guido Tack Christian Schulte and Mikael Z. Lagerkvist. http://www.gecode.org/presentations/INFORMS%20-%20Gecode.pdf.

[8] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. *Lecture Notes in Computer Science*, 2264:73–90, 2001.

[9] Al Geist et al. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[10] Rohit Chandra et al. *Parallel Programming in OpenMP.* Morgan Kaufmann, 2000.

[11] P. van Beek F. Rossi and T. Walsh. *Handook of Constraint Programming*, 2006.

[12] Lígia Ferreira and Salvador Abreu. Design for AJACS, yet another java constraint programming framework. *Elsevier Electronic Notes in Theoretical Computer Science*, 2001.

[13] Lígia Ferreira and Salvador Abreu. Towards a distributed implementation of ajacs, 2004.

[14] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

[15] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming.* Springer, 2003.

[16] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[17] F. Glover and M. Laguna. Tabu search. Kluwer Academic Publishers, 1997.

[18] Michael Gschwind. Chip multiprocessing and the cell broadband engine. *Computing Frontiers 2006*, March 2006. Keynote Speech and Invited Paper.

[19] Michael Gschwind. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35:233–262, June 2007.

[20] Chris Jefferson Ian P. Gent and Ian Miguel. Minion: A fast, scalable, constraint solver. *The European Conference on Artificial Intelligence 2006*, 2006.

[21] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Fourtheenth Annual ACM Symposium on Principles of Programming Languages*, page 111, 1987.

[22] Michael J. Maher Joxan Jaffar. Constraint logic programming: A survey. *Journal of Logic Programming*, 1994.

[23] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *AAAI/IAAI*, pages 169–174, 2000.

[24] Leif Kornstaedt. Alice in the land of Oz – an interoperability-based implementation of a functional language on top of a relational language. In *Proceedings of the First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Electronic Notes in Computer Science*, volume 59, Firenze, Italy, September 2001. Elsevier Science Publishers.

[25] Andrew See Laurent Michel and Pascal Van Hentenryck. Distributed constraint-based local search. *Lecture Notes in Computer Science*, 4204:344–358, 2006.

[26] Rui Machado Luis Almas and Salvador Abreu. Design for a parallel and distributed hybrid constraint programming library. In *proceedings of the 7th International Colloquium on Implementation of Constraint and Logic Programming Systems*, 2007.

[27] Kim Marriot and P.J. Stuckey. *Programming with constraints.* MIT Press, 1998.

[28] Yukio Watanabe Michael Gschwind H. Peter Hofstee Brian Flachs, Martin Hopkins and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26:10–24, March 2006.

[29] L. Michel and P. Van Hentenryck. Comet in context. In *Paris C. Kanellakis Memorial Workshop*, pages 95–107, 2003.

[30] R. Mohr and T.C. Henderson. Arc and path consistency revised. *Artificial Intelligence*, 28:225–233, 1986.

[31] Laurent Perron Pascal Van Hentenryck and Jean-Francois Puget. Search and strategies in opl. TOCL, October 2000.

[32] J.-F. Puget. A c++ implementation of clp. *Ilog Solver Collected papers*, 1994.

[33] Peter Van Roy, editor. *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.

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

[35] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[36] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 2002.

[37] A. Philips S. Minton, M. Johnston and P. Laird. Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[38] IBM Systems and Technology Group. *Cell Broadband Engine Programming Handbook*. IBM, April 2006.

[39] Saraswat et al. Van Hentenryck. Strategic directions in constraint programming. *ACM Computing Surveys*, 28(4), 1996.

[40] Mark Wallace. Survey: Pratical applications of constraint programming. Technical report, Imperial College, 1995.