Towards Effective Parallel Logic Programming

Vítor Santos Costa, Inês de Castro Dutra,
Felipe França, Marluce Pereira, Patrícia Vargas[*]
Cláudio Geyer[†]**, Jorge Barbosa, Cristiano Costa**[‡]
Priscila Lima, Fátima Dargan, António Branco[§]
Juarez Muylaert Filho[¶]**Gopal Gupta**[‖]**, Enrico Pontelli**[**]
Manuel Correia, Ricardo Lopes, Ricardo Rocha, Fernando Silva[††]

To Doris Ferraz de Aragon — in memoriam

Abstract

One of the advantages of logic programming (LP) and constraint logic programming (CLP) is the fact that one can exploit *implicit* parallelism in logic programs. Logic programs have two major forms of implicit parallelism: *or-parallelism* (ORP) and *and-parallelism* (ANDP). In this work we survey some of work that has been taking place within the $CLoP^n$ project towards fast execution of logic programs. We first briefly present our work in comparing several data representations for ORP: copying, copy-on-write, and the sparse binding array. Next, we demonstrate the usefulness of ORP concepts in the context of an important extension of LP, tabling. We then discuss how the process of solving a set of constraints can be dividided by several workers. We conclude with ongoing work on language issues towards the multiparadigm language Holo.

1 Introduction

One of the advantages of logic programming (LP) and constraint logic programming (CLP) is the fact that one can exploit *implicit* parallelism in logic programs. Implicit

[*]COPPE/Sistemas, Universidade Federal do Rio de Janeiro, Brasil
[†]Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brasil
[‡]Departamento de Computação, Universidade Católica de Pelotas, Brasil
[§]ILTC-Universidade Federal Fluminense, Brasil
[¶]IPRJ-Universidade Estadual do Rio de Janeiro, Brasil
[‖]Department of Computer Science, University of Texas at Dallas, USA
[**]Department of Computer Science, New Mexico State University, USA
[††]LIACC, Universidade do Porto, Portugal

parallelism reduces the effort required to speedup logic programs through parallelism. Moreover, implicit parallel systems alleviate the user from the actual details of work management, which can be quite difficult to program for the irregular problems commonly addressed in LP applications. Logic programs have two major forms of implicit parallelism: *or-parallelism* (ORP) and *and-parallelism* (ANDP). Given an initial query to the logic programming system, ORP results from trying several different alternatives simultaneously. In contrast, ANDP stems from dividing the work required to solve the alternative between the different processors.

In this work we survey some of work that has been taking place within the $CLoP^n$ project towards fast execution of Prolog programs. We chose to present four of the several contributions of the research being performed in the project. As regards ORP, we first briefly present our work in comparing several data representations for ORP: copying, copy-on-write, and the sparse binding array. Our work shows that the several models can perform well, although they have different properties and can adapt well to different forms of parallelism. This work should be seen in the context of a recent revival of ORP, both as a form of parallelism that can perform well for standard shared memory machines [24, 1, 37], hardware based shared memory [15, 35], and clusters [43] and as a very general form of parallelism in LP. Indeed, section 3 demonstrates the usefulness of ORP concepts in the context of an important extension of LP, tabling. We show how ORP can be used with little effort to obtain scalable speedups on a novel application of LP, *model checking* [28]. Next, in section 4 we show how a different form of implicit parallelism can be exploited. Namely, we discuss how the process of solving a set of constraints can be dividided by several workers [27], and present exciting early results for this work in a cluster-based setting. We conclude with ongoing work on language issues towards optimal exploitation of parallelism in a distributed setting, and briefly present the multiparadigm language Holo.

2 Or-Parallelism

Arguably, *Or-parallel* systems, such as Aurora [24] and Muse [1], have been the most successful parallel logic programming (PLP) systems so far. One first reason is the large number of logic programming applications that require search. Examples of Prolog applications that perform search include structured database querying, expert systems and knowledge discovery applications. Also, parallelising search can be quite useful for an important extension of Prolog, the constraint-based systems, commonly used for decision-support applications.

Two major issues must be addressed to exploit ORP. First, one must address the *multiple bindings* problem. This problem arises because alternatives being exploited in parallel may give different values to variables in shared branches of the search tree. Several mechanisms have been proposed for addressing this problem [17]. Second, the ORP system itself must be able to divide work between processors. This *scheduling* problem is made complex by the dynamic nature of work in ORP systems.

2.1 Copying

Most modern parallel LP systems, including SICStus Prolog [7] or Yap [36] use copying as a solution to the multiple bindings problem. Copying was made popular by the Muse ORP system, a system derived from an early release of SICStus Prolog. The key idea for copying is that workers work in shared memory, but in separate stacks. Whenever a processor, say W_1, wants to give work to another, say W_2, W_1 simply copies its own stacks to W_2. As we shall see later, the actual implementation of copying requires quite a few more details.

One major advantage of copying is that it has a low overhead over the corresponding sequential system. Moreover, the actual overhead of copying was shown to be quite low in Muse [1]. On the other hand, copying makes it harder to exploit several forms of parallelism. Also, suspending branches in Copying-based PLP systems requires copying the branch to a separate area, and is therefore expensive.

2.2 Copy-On-Write

This model, named αCOWL, has been proposed by Santos Costa [33] with a view to overcome limitations of the environment copying model to support and/or parallelism. The model makes use of the copy-on-write technique that has proven so effective in Operating Systems.

In the αCOWL, and similarly to the environment copying, each computing agent (worker) maintains a separate environment. As in copying, the sharing of work is represented by or-frames in a separate shared data-area. The key idea of αCOWL is as follows: whenever a worker Q wants to share work from a different worker P, it simply *logically* copies all execution stacks P.

The insight is that although stacks will be logically copied, they should be *physically* copied only on demand. To do so, the αCOWL depends on the availability of a Copy-On-Write mechanism on the host Operating System. The αCOWL has two major advantages. First, it is independent of what we are copying, that is, we need not know what to copy, as we logically copy everything. Thus, instead of standard Prolog stacks, we may copy the environment for a constraint solver, or a set of stacks for ANDP computations. Second, because copying is done on demand, we do not need to worry about the overheads of copying large stacks. This is particularly a problem for ANDP computations.

The main drawback of the αCOWL is that the actual setting up of the COW mechanism can be itself quite expensive, and in fact, more expensive than just copying the stacks. In the next sections we discuss an implementation and its performance results.

2.3 Sparse Binding Arrays

The *sparse binding array* (SBA) derives from the binding arrays model. Binding arrays were originally proposed by Warren for the SRI model [44]. In this model execution

stacks are distributed over a shared address space, forming the so-called cactus-stack. In more detail, workers expand the stacks in the part of the shared space they own, and they can directly access the stacks originally created by other workers. In BA based systems, workers initially do not inform the system that they have created new alternatives, and thus have exclusive access to them. This is called *private work*. At some point they may be requested to make this work available. They therefore must make the work *public*.

Most, but not all, accesses to both private and public work are read-only. The major updates to public and private work are for bindings of variables. Bindings to the public part of the tree are temptative, and in fact different alternatives of the search tree may give different values, or even no value, to the same variable. These bindings are called *conditional bindings*, and they are also stored in the *Trail* data-area, so that they can later be undone.

Conditional bindings cannot be stored in the shared tree. The major contribution of the Sparse Binding Array (SBA) is that each worker has a private virtual address space that shadows the system shared address space. This address space is mapped at the same fixed location for each worker in the system. Data structures and unconditional bindings are still stored in the shared address space. Conditional bindings are stored in the shadow area, that is consulted before consulting the shared area. The SBA thus solves the multiple bindings problem.

Note that the shadow area inherits the structure of the shared area. This simplifies implementation, reduces sequential overheads, and allows sharing of the complex stack structure created by ANDP. On the other hand, still requires some modifications to the original Prolog engine and requires more memory than the original BA, thus increasing task-switching overhead.

2.4 Evaluation

To address the question of how these systems fare against copying for ORP we experimented with YapOr, an ORP copying system using the Yap engine [30], and we implemented the SBA and the αCOWL over the original system. The three alternative systems share schedulers and the underlying engine: they do only differ in their binding scheme. We then used a set of well-known ORP all-solutions benchmarks to evaluate how did they perform comparatively.

The results for the αCOWL are quite good, considering the very simple approach we use to share work. The αCOWL performs well for smaller number of processors and for coarse-grained applications. As granularity decreases the overhead of the `fork()` operation becomes more costly, and in general system performance decreases versus other systems. As implemented, the αCOWL is therefore of interest for parallel workstations or for applications with large running times, which are indeed the goal for this study.

Our results also confirm that the SBA is a valid alternative to copying. Although the SBA is slightly slower than copying and cannot achieve as good speedups, it is an

Programs	2 workers			3 workers			4 workers		
	YapOr	αCOWL	SBA	YapOr	αCOWL	SBA	YapOr	αCOWL	SBA
queens12	2.00	1.99	2.00	3.00	2.87	2.99	4.00	3.75	3.99
queens10	1.98	1.84	1.99	2.90	1.90	2.93	3.86	2.03	3.91
cubes7	2.00	1.99	2.02	2.99	2.91	3.03	3.98	3.65	4.05
cubes5	2.00	1.89	2.02	2.97	2.52	3.04	3.95	2.18	4.02
puzzle	1.98	1.95	1.91	2.97	2.38	2.84	3.96	2.74	3.79
nsort	2.00	2.02	1.92	3.01	2.95	2.86	4.02	3.86	3.82
ham	1.99	1.88	1.98	2.95	2.70	2.94	3.90	2.50	3.85
Average	1.99	1.94	1.98	2.97	2.60	2.95	3.95	2.96	3.92

Table 1: Speedups for the three models on the PC Server.

interesting alternative for the applications where copying does not work so well. As an example we are using the SBA to implement IAP.

3 Parallel Tabling

Ideally, one would want Prolog programs to be written as logical statements first, and for control to be tackled as a separate issue. In practice, the operational semantics of Prolog is given by SLD-resolution with depth-first search, a refutation strategy particularly simple and that matches current stack-based machines particularly well. Unfortunately, the limitations of SLD-resolution mean that Prolog programmers must be very aware of the Prolog computation rule throughout program development. For instance, logically correct programs will may have infinite loops.

Several proposals have been put forth to improve the declarativeness and expressiveness of Prolog. One such proposal that has been gaining in popularity is the use of *tabling* or *memoing*. In a nutshell, tabling consists of storing intermediate solutions to a query so that they can be reused during the query execution process. It can be shown that tabling-based computational rules can have better termination properties than SLD-based models, and indeed termination can be guaranteed for all programs with the *bounded term-size property* [9].

Work on SLG-resolution [10], as implemented in the XSB System [32], proved the viability of tabling technology for applications such as natural language processing, knowledge-base systems and data-cleaning, model-checking, and program-analysis. Tabling also facilitates the implementation of several extensions to Prolog, including support for negation [32] that allows for non-monotonic reasoning.

Tabling can work for both deterministic and non-deterministic programs, but quite a few interesting applications of tabling are by nature non-deterministic. This rises the question of whether further efficiency would be possible by running several branches of the search tree in parallel. Freire and colleagues were the first to propose that tabled goals could be a source of parallelism [16]. In this model, each tabled subgoal is computed independently in a single computational thread, a *generator thread*, that is

responsible for fully exploiting its search tree and obtain the complete set of answers. A generator thread dependent on other tabled subgoals will asynchronously consume answers as the correspondent generator threads will make them available. This model is limitative in that it restricts parallel exploitation to having several generator threads running concurrently. Parallelism arising from non-tabled subgoals or from execution alternatives to tabled subgoals is not exploited.

In contrast, we argue that the same mechanism can be used to exploit or-parallelism from both tabled and non-tabled subgoals. By doing so we can both extract more parallelism, and we can reuse the technology presented for or-parallelism and tabling. Towards this goal, we have proposed two computational models, the *Or-Parallelism within Tabling (OPT)* and *Tabling within Or-Parallelism (TOP)* [29]. In this paper we present the implementation of the OPTYap system, based on the OPT model. The OPT model considers tabling as the base component of the system, that is, each computational worker behaves like a full sequential tabling engine. The or-parallel component of the system is only triggered when a worker runs out of alternatives to exploit. The OPT model gives the highest degree of orthogonality between or-parallelism and tabling, thus simplifying initial implementation issues.

3.1 Or-Parallelism within Tabling

We use the example in Figure 1 in order to illustrate how parallelism can be exploited in the OPT model. The example assumes two workers, $\mathcal{W}_1$ and $\mathcal{W}_2$, and it represents a possible evaluation for a tabled program with `?- a(X).` as the query goal.

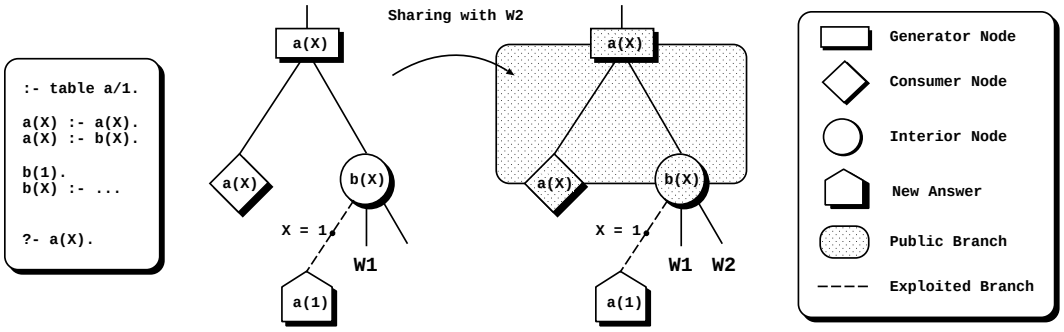

Figure 1: Exploiting parallelism in the OPT model.

Consider that worker $\mathcal{W}_1$ executes the query goal. It first inserts an entry for the tabled subgoal `a(X)` into the table space and creates a generator node for it. The execution of the first alternative leads to a recursive call for `a(X)`, and thus it creates a consumer node for `a(X)` and because there are no available answers, it backtracks. The next alternative finds a non-tabled subgoal `b(X)` for which an interior node is created. The first alternative for `b(X)` succeeds and an answer for `a(X)` is therefore found: `a(1)`. The worker inserts the newly found answer in the table and then starts exploiting the next alternative for `b(X)`. At this point, worker $\mathcal{W}_2$ moves in to share

work. Consider that worker $\mathcal{W}_1$ decides to share all of its private nodes. The two workers will share three nodes: the generator node for `a(X)`, the consumer node for `a(X)` and the interior node for `b(X)`. Worker $\mathcal{W}_2$ takes the next unexploited alternative of `b(X)` and from now on, both workers can find further answers for `a(X)` and any of them can restart the shared consumer node.

In our model, each worker physically owns a copy of the environment, that is, its stacks, and shares an area storing tabling and scheduling data. A set of independent workers executes a tabled program by traversing the corresponding search tree in search of nodes that are candidate entry points for parallelism. To reduce overheads, the search tree is implicitly divided into public and private regions. Workers in the private region execute nearly as in sequential tabling. Workers exploiting the public region of the search tree must to synchronise in order to ensure the correctness of the tabling operations.

When a worker runs out of alternatives to exploit, it enters in scheduling mode. The YapOr scheduler is used to search for busy workers with unexploited work. Alternatives should be made available for parallel execution, regardless of whether they originate from generator, consumer or interior nodes. A worker is said to have sharable work if it contains private nodes with unexploited alternatives or with unconsumed answers. We use incremental copy technique to share work, that is, we only copy the *differences* between stacks.

Parallel execution requires significant changes to the SLG-WAM. Synchronisation is required when backtracking to a public generator or to an interior node to take the next available alternative; when backtracking to a public consumer node to take the next unconsumed answer; and when inserting new answers into the table space. Parallel support changes the completion mechanism, computation resumption, and the fixpoint check procedure. First, in parallel systems the relative positions of generator and consumer nodes are not easily determined, hence resulting in more complex algorithms to determine whether a node can be a leader node and to determine whether a SCC can be completed. Second, the condition of being a leader node is not, by itself, sufficient to perform completion.

3.2 Performance

Table 2 presents the speedups for OPTYap with 2, 4, 6, 8, 12, 16, 24 and 32 workers using batched scheduling. The environment for our experiments was *oscar*, a Silicon Graphics Cray Origin2000 parallel computer from the Oxford Supercomputing Centre. *Oscar* consists of 96 MIPS 195 MHz R10000 processors each with 256 Mbytes of main memory (for a total shared memory of 24 Gbytes) and running the IRIX 6.5.12 kernel. The first three tabled benchmark programs were obtained from the XMC system [28]. XMC is a model checker implemented atop the XSB system which verifies properties written in the alternation-free fragment of the modal μ-calculus [18] for systems specified in XL, an extension of value-passing CCS [25]. The speedups correspond to the best speedup obtained in a set of 3 runs. The table is divided in two

main blocks: the upper block groups the benchmarks that showed potential for parallel execution, whilst the bottom block groups the benchmarks that do not show any gains when run in parallel.

	Number of Workers							
Program	**2**	**4**	**6**	**8**	**12**	**16**	**24**	**32**
sieve	2.00	3.99	5.99	7.97	11.94	15.87	23.78	31.50
leader	2.00	3.98	5.97	7.92	11.84	15.78	23.57	31.18
iproto	1.72	3.05	4.18	5.08	7.70	9.01	8.81	7.21
samegen	1.94	3.72	5.50	7.27	10.68	13.91	19.77	24.17
lgrid/2	1.88	3.63	5.29	7.19	10.21	13.53	19.93	24.35
Average	1.91	3.67	5.39	7.09	10.47	13.62	19.17	23.68
lgrid	0.46	0.65	0.69	0.68	0.68	0.55	0.46	0.39
rgrid/2	0.73	0.94	1.01	1.15	0.92	0.72	0.77	0.65
Average	0.60	0.80	0.85	0.92	0.80	0.64	0.62	0.52

Table 2: OPTYap speedups.

The results show superb speedups for the XMC *sieve* and the *leader* benchmarks up to 32 workers. These benchmarks reach speedups of 31.5 and 31.18 with 32 workers! Two other benchmarks in the upper block, *samegen* and *lgrid/2*, also show excellent speedups up to 32 workers. Both reach a speedup of 24 with 32 workers. The remaining benchmark, *iproto*, shows a good result up to 16 workers and then it slows down with 24 and 32 workers. Globally, the results for the upper block are quite good, especially considering that they include the three XMC benchmarks that are more representative of real-world applications.

4 Constraints

Finite domain Constraint Satisfaction Problems (CSPs) usually describe NP-complete search problems. Algorithms exist, such as arc-consistency algorithms, that help to eliminate inconsistent values from the solution space. They can be used to reduce the size of the search space, allowing to find solutions for large CSPs.

Still, there are problems whose instance size make it impossible to find a solution with sequential algorithms. Concurrency and parallelisation can help to minimise this problem because a constraint network generated by a constraint program can be split among processes in order to speed up the arc-consistency procedure.

Parallelisation of constraint satisfaction algorithms brings two advantages: (1) programs can run faster, and (2) large instances of problems can be dealt with because of the amount of resources (memory and cpus).

Andino *et al* [31] implemented a parallel version of the AC-5 algorithm [40] for a logically shared memory architecture, the Cray T3E, a high cost parallel platform.

Our work is based on Andino's *et al* implementation. We seek to obtain good performance on on-the-shelf low cost clusters of PCs. We adapted their algorithms and data structures to run on a distributed-shared memory platform using TreadMarks, a software Distributed Shared-Memory (DSM) system [2].

Our results show that arc-consistency algorithms can achieve good speedups on distributed-shared memory systems. One of our applications achieves superlinear speedups due to the distributed labeling.

4.1 PCSOS and PCSOS-TMK

PCSOS was implemented on a logically shared memory platform, the Cray T3E.

Our first step on porting the PCSOS to a DSM platform was to study its data structures, understand them, and separate private data from shared data. We also needed to adapt the data structures to the software DSM we used, since the PCSOS system relied on the SHMEM library [39] to access logically shared data on remote nodes. We then established the right synchronisation points in the source code in order to obtain a parallel correct code.

Besides porting the original PCSOS code to a software DSM platform, we implemented two kinds of labeling: *sequential* and *distributed*, and two kinds of partitioning of indexicals: *round-robin*, and *block*. The sequential labeling assumes that each processor can apply the labeling procedure over any variable. The distributed labeling partitions the set of variables in subsets of equal size, and each processor can execute the labeling procedure over its own subset. The round-robin partitioning of indexicals assumes that each indexical is allocated to each process at a time. The partition of indexicals in blocks assumes that a block of consecutive indexicals is allocated to each process. This partitioning is done in the beginning of the computation upon reading the input data file that contains the constraint network.

4.2 Arithmetic

Arithmetic is a synthetic benchmark. It is formed by sixteen blocks of arithmetic relations, $\{B_1, \ldots, B_{16}\}$. Each block contains fifteen equations and inequations relating six variables. Blocks B_i and B_{i+1} are connected by an additional equation between a pair of variables, one from B_i and the other one from B_{i+1}. Coefficients were randomly generated. The goal is to find an integer solution vector. This kind of constraint programming is very much used for decomposition of large optimisation problems.

This application has a constraint graph weakly connected whose pattern of connection is regular, i.e., for each pair of blocks there are only two edges connecting them: one edge connecting a block of equations to the next, and one edge connecting the next block to the previous. We explored the structure of this graph in order to do a better distribution of indexicals and labeling among processors. Since the problem is structured in blocks of equations, we distributed the indexicals in blocks. Besides,

each processor labels its own subset of variables, communicating only when pruning a variable which connects different blocks allocated to different processors.

Number of Processors	Execution Times (sec.)
1	4.74
2	0.39
4	0.37
8	0.47

Table 3: *Arithmetic*: Execution Times for 1, 2, 4 and 8 Processors

Table 3 shows the execution times of the application *Arithmetic* for 1, 2, 4 and 8 processors. This application presents superlinear speedups when we explore the constraint graph and distribute the labeling among processors. Note that on parallelising the sequential algorithm we naturally achieve distributed labeling, because we take advantage of the constraint graph structure. For two processors, the execution time is 12.14 times less than the execution time for one processor. The lowest speedup we obtain with this application is 10.01, for 8 processors related to the execution time for one processor. From 2 to 4 processors we obtain a discrete improvement in performance, and from 4 to 8 processors we obtain a slowdown. Because of the cache coherence protocol (invalidate) and the memory consistency model (LRC) used by TreadMarks, the overheads for this application are too high, when we increase the number of processors.

4.3 PBCSP

Our second application, *Parametrizable Binary Constraint Satisfaction Problem* (PBCSP), has a constraint graph that is strongly connected, and does not present a regular connection pattern, i.e., the edges are randomly generated to form the final constraint graph. Therefore, most of our experiments with this application, were done using sequential labeling. Our choice for the partitioning of indexicals was round-robin. We ran this application with two input data, containing 100 variables (PBCSP_1), and 200 variables (PBCSP_2).

Table 4 shows the execution times and speedups, respectively of PBCSP_1 and PBCSP_2. The minimum execution time for PBCSP_1 was achieved with 2 processors, which yielded a speedup of 1.31 related to the execution time for one processor. We manage to keep a discrete speedup related to one processor up to 4 processors. The performance degrades for 8 processors.

Number of Processors	Execution Times (sec.)	
	PBCSP_1	PBCSP_2
1	26.55	39.36
2	20.13	29.54
4	26.38	26.03
8	65.22	50.41

Table 4: PBCSP_1 and PBCSP_2: Execution Times for 1, 2, 4 and 8 Processors

5 The Holoparadigm

The last few years have seen interest in programming languages that that can support several paradigms [3, 4, 5, 11, 26]. The researchers in this area propose novel languages that integrate basic paradigms, such as the imperative, logic, functional, and object oriented. Their goals are to overcome the issues with each individiual paradigm and to exploit the synergie between different paradigms.

The imperative paradigm difficults the automatic exploitation of parallelism, as it depends on control commands, memory positions, and variables that allow for destructive updates. Imperative programs may exhibit control and data dependencies. Other basic paradigms display implicit sources of parallelism, allowing for automatic exploitation of parallelism, We have seen the advantages of logic basic languages for ORP and ANDP. Object-oriented programming also allows both inter and intra-object parallelism.

A second motivation for novel languages is the advantges in microlectronics, which have reduced hardware cost, and in networking, which allows for fast interconections. The rapid growth of the *Internet* has also made it clear that computational systems must move towards distributed computing. Novel computational languages that can support distributed computing are therefore of interest.

5.1 The Paradigm

The Holoparadigm is a multiparadigm model dedicated towards automatic exploitation of parallelism and distribution. The model assumes as modelling unity *beings* and as data unit *symbols*. Beings may be *static*, programs, or *dynamic*, execution. Beings are created through cloning. There are three forms of cloning: *static* cloning is similar to inheriti8ng in object oriented languages, transition cloning creates a dynamic being based on a static being, and dynamic cloning creates a new dynamic being from a static being. A further organisationl classification distinguishes being according to their structure as either *atomic* or *composed*.

An atomic being consists of three components, an interface, behavior and history. The interface describes the possible relationships with the other beings. The behavior

contains operations that implement the being's functionality. The history is a space of shared memory within the being. The Holoparadigm proposes using symboling processing as the main technique for information processing. This characteris is inherited from logic programming. In this sense, the logic variable and the unification are the foundations for symbolic processing.

A composed being is organised in much the same way as an atomic being. Regardless, it supports other beings in its composition (the component beings). Each being has its own history. The history is encapsulated in the being and, in the case of composed beings, its is shared by all component beings. We therefore allow several levels of sharing between stories.

A being may assume two states of distribution: it may be centralised (that is, located at a single node of the system), or distributed, that is, located at several nodes. In the latter case history acts as shared distributed memory, or DSM. Mobility is the capacity that allows beings to move. The Holoparadigm supports two forms of mobility: logical and physical. Logical mobility relates with movement as the modelling level, that is, regardless of the execution platform. In this context, a being moves when it crosses one or more borders between beings. The *physical* mobility relates with moving between nodes of a distributed computer. Note that after moving the being is not allowed further access to the source history, although it is allowed access to the history in the target being. Note also that physical mobility requires for the source and target beings to be allocated at physically different nodes. Last, we remark that a node may move physically without moving logically.

We propose a *blackboard* as the coordination model. The *Knowledge Sources* (KSs) are beings and the *blackboard* is the history. Our model assumes implicit invocation, meaning, the *blackboard* performs communication and synchronisation between KSs. To address the limitations in implicit invocation we also allow for explicit invocation in the Holoparadignm.

5.2 The Language

The Holo Language integrates the imperative, logic and object oriented paradigm. The language uses a symbolic representation, unification, and is untyped, as in logic programming. Holo allows for higher-level programming. A Holo program consists of descriptions of beings. Each description includes a head and a body.

Holo uses program transformation as an implementation technique. We propose using Java as the ideal intermediate language for building the Holo compiler. We therefore built the HoloJava system, a program transformer that can from Holo to Java. HoloJava is based on JavaCC, a compiler generator for Java. We also use two class libraries, JIProlog (a Java Prolog) to implement logical actions and Jada (Java blackboards) to implement the history.

6 Conclusions

We have presented four of the main strands of research being pursued in the $CLoP^n$ project. From the beginning, the project's goals were to achieve high-performance and scalable performance for logic and constraint programming. Other research not presented here due to space limitations includes support for sequential performance [34, 8], hardware for computation reuse [14, 13], neural networks [19, 42], system performance analysis [35, 15], independent and-parallelism [12], granularity-based scheduling [6, 41], and novel computation models for parallelism [23, 22, 21, 20]. One of the goals of the $CLoP^n$ project was to support real applications: we discussed model checking and equation solving, other work included Inductive Logic Programming [38]. We plan to continue and consolidate this work in the last year of the project, with emphasis on application work.

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] Krzysztof R. Apt and Andrea Schaerf. The alma project, or how first-order logic can help us in imperative programming. In *Correct System Design*, pages 89–113, 1999.

[4] Jorge Luis Victória Barbosa and Cláudio Fernando Resin Geyer. Software Multiparadigma Distribuído. *Revista de Informática Teórica e Aplicada (RITA)*, December 1999.

[5] Jorge Luis Victória Barbosa and Cláudio Fernando Resin Geyer. Uma Modelo Multiparadigma para Desenvolvimento de Software Paralelo e Distríbuido. In *I Workshop on High Performance Computing Systems, SBC*, 2000.

[6] Jorge Luis Victória Barbosa, Patrícia Kayser, Cláudio Fernando Resin Geyer, and Inês C. Dutra. GRANLOG: An Integrated Granularity Analysis Model for Parallel Logic Programming. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages*, July 2000.

[7] Mats Carlsson and Johan Widen. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, 1988. SICS Research Report R88007B.

[8] Luís Fernando Castro and Vítor Santos Costa. Understanding Memory Management in Prolog Systems. In *Proceedings of ICLP'01, to be published*, November 2001.

[9] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[10] W. Chen and David S. Warren. Query evaluation under the well-founded semantics. In *Proc. of 12th PODS*, pages 168–179, 1993.

[11] A. Ciampolini, E. Lamma, P. Mello, and C. Stefanelli. Distributed logic objects: a fragment of rewriting logic and its implementation.

[12] Manuel Eduardo Correia. *On the Implementation of AND/OR Parallel Logic Programming Systems*. PhD thesis, Universidade do Porto, August 2001.

[13] Amarildo T. Costa, Felipe M.G. França, and Eliseu M. Chaves Filho. "exploiting reuse with dynamic trace memoization: Evaluating architectural issues". In *XII Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), SBC*, pages 163–172, October 2000.

[14] Amarildo T. Costa, Felipe M.G. França, and Eliseu M. Chaves Filho. "the dynamic trace memoization reuse technique". In *Proc. of The International Conference on Parallel Architectures and Compilation Techniques - PACT 2000*, pages 92–99, Philadelpha, PA, USA, October 2000.

[15] Inês C. Dutra, Vítor Santos Costa, and Ricardo Bianchini. The Impact of Cache Coherence Protocols on Parallel Logic Programming Systems. In *Proceedings of CL'2000, LNAI 1861*, pages 1285–1299, July 2000.

[16] Juliana Freire, Rui Hu, Terrance Swift, and David S. Warren. Exploiting Parallelism in Tabled Evaluations. In *7th International Symposium PLILP*, pages 115–132, 1995.

[17] Gopal Gupta and Bharat Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.

[18] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[19] Priscila M. V. Lima. "a neural propositional reasoner that is goal-driven and works without precompiled knowledge". In *Proc. of The IVIth Brazilian Symposium on Neural Networks*, Rio de Janeiro, Brasil, October 2000.

[20] Ricardo Lopes. *An Implementation of the Extended Andorra Model*. PhD thesis, Universidade do Porto, September 2001.

[21] Ricardo Lopes, Vítor Santos Costa, and Fernando Silva. A novel implementation of the extended andorra model. In *Proceedings of PADL01: The 2001 Conference on the Practical Aspects of Declarative Languages*, pages 199–213, Las Vegas, NV, USA, 2001.

[22] Ricardo Lopes and Vítor Santos Costa. A Performance Analysis of the BEAM Memory Manager. In *Proceedings of AGP00: The 2000 Joint Conference on Declarative Programming*, December 2000.

[23] Ricardo Lopes, Fernando Silva, Vítor Santos Costa, and Salvador Abreu. The RAINBOW: Towards a Parallel Beam. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages, CL2000*, pages 38–54, July 2000.

[24] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross A. Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Péter Szeredi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumił Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[25] R. Milner. *Communication and Concurrency*. Preentice-Hall, 1989.

[26] M. Muller, T. Muller, and P. Van Roy. Multiparadigm programming in oz, 1995.

[27] Marluce Rodrigues Pereira. Paralelização de Algoritmos de Consistência de Arcos em um Cluster de PCs. Master's thesis, COPPE/Sistemas, UFRJ, August 2001.

[28] C. R. Ramakrishnan, I. V. Ramakrishnan, S. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Proceedings of Computer Aided Verification*, 2000.

[29] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. Or-parallelism within tabling. *Lecture Notes in Computer Science*, 1551:137–151, 1999.

[30] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. In *LNAI 1695, Proceedings of EPPIA'99: The 9th Portuguese Conference on Artificial Intelligence*, pages 178–192. Springer-Verlag LNAI Series, September 1999.

[31] Alvaro Ruiz-Andino, Lourdes Araujo, Fernando Sáenz, and José J. Ruz. Parallel execution models for constraint programming over finite domains. In *LNCS 1702, Proceedings of PPDP'99*, pages 134–151. Springer-Verlag, September 1999.

[32] Konstantinos F. Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, May 1998.

[33] Vítor Santos Costa. Cowl: Copy-on-write for logic programs. In *Proceedings of the IPPS/SPDP99*, pages 720–727. IEEE Computer Press, May 1999.

[34] Vítor Santos Costa. Optimising bytecode emulation for prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.

[35] Vítor Santos Costa, Ricardo Bianchini, and Inês C. Dutra. Parallel Logic Programming Systems on Scalable Architectures. *Journal of Parallel and Distributed Computing*, 60(7):835–852, July 2000. http://www.idealibrary.com/links/toc/jpdc/60/7/0.

[36] Vítor Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*, 2000. http://www.ncc.up.pt/˜vsc/Yap.

[37] Vítor Santos Costa, Ricardo Rocha, and Fernando Silva. Novel Models for Or-Parallel Logic Programs: A Performance Analysis. In *Proceedings of EuroPar2000, LNCS 1900*, pages 744–753, September 2000.

[38] Vítor Santos Costa, Ashwin Srinivasan, and Rui Camacho. A note on two simple transformations for improving the efficiency of an ILP system. In *Proceedings of ILP'2000, LNAI 1866*, pages 225–242, July 2000.

[39] S. L. Scott. Synchronization and Communication in the t3e Multiprocessor. In *ASPLOS7*, pages 26–36, October 1996.

[40] Pascal Van Hentenryck, Yves Deville, and C. M. Teng. A Generic Arc-Consistency Algorithm and its Specifications. *Artificial Intelligence*, 57(2–3):291–321, October 1992.

[41] Patrícia Kayser Vargas, Jorge Luis Victória Barbosa, Débora Nice Ferrari, Cláudio Fernando Resin Geyer, and Jacques Chassin. Distributed OR Scheduling with Granularity Information. In *XII Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), SBC*, pages 253–260, October 2000.

[42] Iara Vilela and Priscila Lima. "conjecturing the cognitive plausibility of an ann theorem-prover". In *Proc. of The International Work-conference on Artificial and Natural Neural Networks (IWANN'2001)*, Granada, Spain, June 2001.

[43] Karen Villaverde, Hai-Feng Guo, Enrico Pontelli, and G. Gupta. High Performance (Constraint) Logic Programming on the Beowulf Architecture. In *Proceedings of ICLP01, to be published*, November 2001.

[44] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.