A High Level Model and Implementation of Labeling in Distributed (C)LP Systems*

J. M. Gómez
jgomez@fi.upm.es

School of Computer Science
Technical University of Madrid (UPM)

Abstract

Distributed programming pursues the execution of the subtasks a program can be divided into, using a set of (possibly heterogeneous) computers. We describe preliminary work on the design and implementation of a particular case of the more general Distributed Constraint Logic Programming paradigm, which encompasses a wider range of solving methods and constraint domains. This work is intended to exploit the disjunctive parallelism contained in Constraint Logic Programming over finite domains, trying to optimize execution time. The search tree of the labeling process is split into subtrees, intending to achieve parallelism by distributing them across the network, among a number of workers. During exploration these subtrees are processed, and at the same time new work is produced. A dynamic scheduling that fosters a solidary model of work sharing is, in general, expected to offer more optimal results than static strategies. The target of this scheduling is to keep all the workers equally busy, in order to maximize the exploitation of parallelism as well as to minimize communication overhead.

1 Introduction

Current communication lines are becoming quite reliable and of high bandwidth. This is certainly so in local networks, and it is becoming the norm in wide-area networks and in the whole of Internet. This opens the practical possibility of using networks as *computation farms*, and it has been exploited in efforts such as Mersenne prime search using the Entropia architecture [CCEB03], which relies on duplicating work in several personal computers in order to overcome possible wrong answers and computer or network failures, or, more recently, the GRID proposals [FKNT99] in its various inceptions.

It is to be expected that, as distributed applications appear and become more common, their internal complexity will grow: from relatively straightforward (although not trivial) hand-coded load and task distribution decisions to the need of automatic granularity analysis [LGHD96], and from algorithms which are in principle easy to split into separate, independent sections to algorithms which are irregular in nature. It is also to be expected that the need for higher-level programming paradigms which arose in sequential programming will be carried on to large-scale distributed programs in order to solve problems which are both big in terms of size of the data, and complex in terms of the algorithms.

Studying and assessing distribution policies which can take advantage of a set of machines connected in a network when executions of irregular algorithms are to be performed turns out to

*This work has been partially funded by the European Union IST program under contract IST-2001-34717, Amos, and by Spanish MCYT project TIC 2002-0055, CUBICO.

be necessary. In particular, we aim at developing and studying platforms for the distributed execution of constraint logic programs [MS98, JL87], applicable when a natural (or advantageous) formulation of a problem uses constraint-based programs, and when, at the same time, the size of the input data makes it necessary to distribute load among different processing units. Constraint logic languages offer some advantages for distributed (and parallel) execution since its sequential semantics is relatively easy to reconcile with parallel/distributed execution, and it makes implementation of automatic analysis tools (for, e.g., independence or granularity) easier than in other languages.

Among the different possibilities of combination of program control and load distribution (which are not unlike those already investigated for parallel execution [GPA$^+$01]) two of them (and their combinations) have been recognized as the more interesting ones: and-parallelism, where goals in the body of a clause are executed in parallel, and or-parallelism, where different clauses (in general, branches of a search process) of a predicate are executed in parallel. In general, and-parallelism is more difficult to implement due to the need to ensure that conditions to avoid speed-down with respect to sequential execution hold. This can always be done at run-time, and also (but only partially) at compile-time. Besides, implementing a correct and fast backtracking in an and-parallel systems is a difficult task. Or-parallelism is easier to implement (as witnessed by the systems offering it since time ago) due to the independence of the different branches to be explored.

In this piece of work we present a high-level, or-parallel, distributed implementation applied to the *finite domain* (FD) constraint system [Van89a, Van89b]. Some parts of the execution mechanism of FD (and, in general, of other constraint systems) can be executed in an and-parallel fashion, but solving an FD program usually includes a part (the *labeling* or *enumeration*) which can be executed in an or-parallel way quite easily. The range of high-level comes from using as implementation basis a sequential implementation of an FD solver in the Ciao Prolog [HBC$^+$00, HBC$^+$99] system which allows seeing the constraints and constraint stores as objects to be communicated between the processing units participating in an execution.

The initial aim of the implementation is to serve as initial base to develop more efficient implementations, and also to be used as testbed to further develop distribution and granularity algorithms. We evaluate the efficiency of the implementation and its behavior in a non-distributed architecture (a parallel machine), but using the same mechanisms that would be used in a distributed execution. Therefore, and module communication speed failures, which can however be simulated by introducing artificial delays of arbitrary length in the communication channels, we expect to obtain a behavior which reflects that of a distributed system.

2 Introduction to CLP($\mathcal{FD}$)

Constraint Logic Programming is an extension of Logic Programming, usually (but not necessarily) taking the Prolog language as base, which augments LP semantics with constraint (e.g., equation) handling capabilities, including the ability to generate constraints dynamically (e.g., at run time) to represent problem conditions and also to solve them by means of internal, well-tuned, user-transparent constraint solvers. Constraints can come in very different flavors, depending on the *constraint system* supported by the language. Examples of well-known constraint systems are linear [dis]equations, either over $\mathcal{R}$ or over $\mathcal{Q}$ [JM87], $\mathcal{H}$ (equations over the Herbrand domain, finite trees), $\mathcal{FD}$ ([dis]equations over variables which range over finite sets with a complete order among their elements, usually represented as integers [Van89a]).

$\mathcal{FD}$ is one of the more widely used constraint domains, since the finiteness of the domain of the variables allows, in the worst case, a complete traversal of each variable range when searching for a solution. This gives complete freedom to the type of equations an $\mathcal{FD}$ system can handle.[1] Figure 1 shows a toy CLP($\mathcal{FD}$) program, with the same overall structure of other larger CLP($\mathcal{FD}$) programs. Briefly, the declarative reading of the program is that it succeeds for all values of X, Y

[1]Note that many constraint systems do not have a complete solution procedure.

and Z such that

$$X, Y, Z \in \mathbb{N} \land 1 \leq X, Y, Z \leq 5 \land X - Y = 2Z \land X + Y \geq Z$$

and fails if no values satisfies all of these constraints. Operationally, and also from the viewpoint of a programmer, the program first declares initial ranges for variables `X`, `Y`, and `Z`,[2] then a set of relationships are set up among them, and finally a search procedure is called to bind the variables to *definite* values. The first phase (setting up equations) fires a process called *propagation*, in which some values can be removed from the domain of the variables (e.g., from $1 \leq X, Y, Z \leq 5$ and $X + Y \leq Z$, the values 4 and 5 can be removed from the domain of X and Y). Usually this stage does not end with a definite value for each variable, which is sought for in a second search process, called *labeling* or *enumeration*: variables in the problem are assigned values within their domains until all of them have a unique value satisfying all the equations in the problem. In this process, if some assignment is inconsistent with the equations, backtracking is performed looking for a different mapping from variables to values. These two stages are radically different in that propagation is a deterministic process, while *labeling* is non-deterministic. In fact, after each assignment made by the labeling process, a series of propagation steps can take place. In a real program several propagation / labeling stages can be freely intertwined.

```
main(X,Y,Z) :-
    [X, Y, Z] in 1..5,
    X - Y .=. 2*Z,
    X + Y .>=. Z,
    labeling([X,Y,Z]).
```

Figure 1: A short CLP($\mathcal{FD}$) program

In the example, the initial propagation phase, before the labeling, reduces the domains of the variables to be:

$$\texttt{s0} : X \in \{3, 4, 5\} \land Y \in \{1, 2, 3\} \land Z \in \{1, 2\}$$

Different propagation schemes can have different power and yield domains more or less tight. This is not a correctness / completeness problem, as labeling will eventually remove inconsistent values. Removing as much values as possible is advantageous, since this will make the search space smaller, but the computational cost of a more precise domain narrowing has to be balanced with the savings in the search. The state `s0` can be taken as the initial root where the search starts from, as the propagation is deterministic and does not leave branches to explore.

If we assume variables are *labeled* in lexicographical order, the next search step will generate three different nodes, resulting from instantiating `X` to the values in its domain. Each of these instantiations will in turn start a propagation (and simplification) which will lead to the following three states:

$$\begin{array}{rl}\texttt{s01}: & X = 3 \land Y = 1 \land Z = 1 \\ \texttt{s02}: & X = 4 \land Y = 2 \land Z = 1 \\ \texttt{s03}: & X = 5 \land Y \in \{1, 2, 3\} \land Z \in \{1, 2\}\end{array}$$

`s01` and `s02` are completely determined, and are final solutions. If only one solution were needed, the execution could have finished when $X = 3$ was executed. If more solutions are required, further exploration can be performed starting at `s03`, resulting in the children states `s031`, `s032`, and `s033`, where Y is instantiated to the values in its domain:

$$\begin{array}{rl}\texttt{s031}: & X = 5 \land Y = 1 \land Z = 2 \\ \texttt{s032}: & X = 5 \land Y = 3 \land Z = 1\end{array}$$

[2]Large default ranges are automatically selected if this initialization is not present.

At this point, no more search either propagation is possible, and all the solutions to the constraint problem have been found. Note that the combination $X = 5, Y = 2$ leads to an inconsistent valuation, and is not shown.

An interesting property of the search in the subtrees is their independence (at this level of abstraction): exploration in each of the branches does not need to communicate values with the exploration in other branches. Of course, if different agents are assigned to each branch and only one solution is needed, signaling when an alternative has found a solution is needed so that no resources are wasted in the other branches. How different nodes are mapped to each node is a scheduling problem to be tackled at a higher level. Dashed boxes in Figure 2 reflect a possible assignment of sets of nodes to agents.

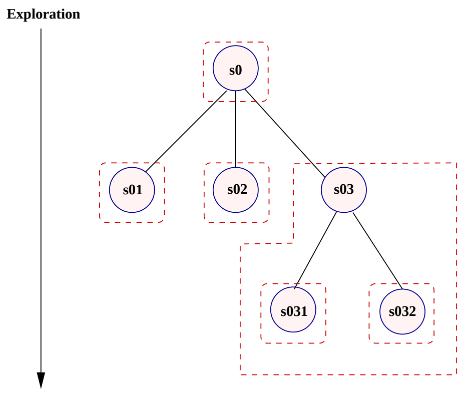

Figure 2: Search tree for the program in Figure 1

The size (depth and width) of the search depends on the number of variables and the domain size of each of them. Although propagation prunes search by removing values before they are assigned by the labeling process (which can in fact reduce greatly the time needed to solve large combinatorial problems), the combined cost of search and consistency maintenance can be high. However, each selection of a value from a variable domain generates a different subtree which can be explored independently from the other subtrees (in an or-parallel fashion), while selecting variables one after another can be executed in an and-parallel fashion. Both execution mechanisms are amenable to be distributed, as in [Leu93, YDIK98]. However, the relative independence of search subtrees should need, in principle, exchanging fewer messages, while at the same time being more resilient to communication delays and even link failures if only one solution (and no preference among different solutions) is needed.

3 Architectural Design

We will describe now, from a high level point of view, a design and interaction scheme to solve CLP($\mathcal{FD}$) problems in a distributed way.

3.1 Worker-Manager Design

We have chosen a client-server architecture (figure 3), where a designated agent acts as *Manager*, coordinating the search space of the other *Worker* agents, which perform the actual search. The workers communicate with each other by message passing using the manager as intermediate communication point. Although this may cause the manager to become a communication bottleneck, experiments presented in section 7 show that this is not the case. Additionally, the simplicity of the design helps keeping the communication and distribution algorithms simple and concentrating on the distribution of constraints and search trees.

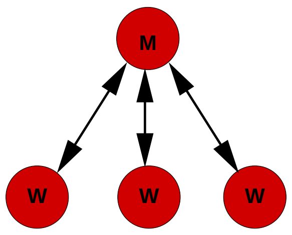

Figure 3: Manager-workers relationship

All workers start in an *Initial* state (corresponding to no activity) and, when work is available, evolve to one of the *busy* states (Figure 5). Messages are by default processed in arrival order, intending to mimic the behavior of a sequential search process.

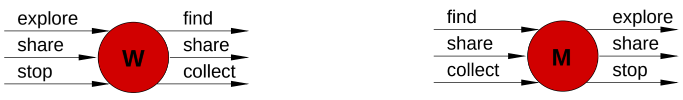

Figure 4: Messages exchanged between workers and manager

3.2 Messages

The overall execution is driven by the manager by means of messages sent to and from the workers. The direction of these messages is shown in Figure 4, and the meaning of the messages is as follows:

Explore(Nodes): the manager sends this message to an idle worker requesting the exploration of the node(s) included in the message.

Find: when the current subtree has been fully explored without finding a solution and the worker becomes idle, a *Find* message is sent to the manager requesting for more work. The manager forwards the request to the non-idle workers.

Share(Nodes): the busy workers send back to the manager one or several of its nodes pending to be explored.

Explore(Nodes): the manager sends to the requesting worker the work received from some currently busy worker.

Collect(Solution): when a worker finds a solution it is sent to the manager in a *Collect* message.

Stop: the manager asks a worker to stop searching (for example, because only one solution was needed and it has already been found).

Scalability: While in the preliminary experiments the manager did not become a performance problem, when the number of workers grows the amount of messages and information exchanged can clearly become and unbearable overhead which slows down the overall process. We plan to overcome this situation by extending the manager protocol to support hierarchical, tree-shaped communication networks (Figure 6) which keeps messages (for example, work requests) as local as possible, and forwarding them to upper levels only when no answer is found within a manager subtree.

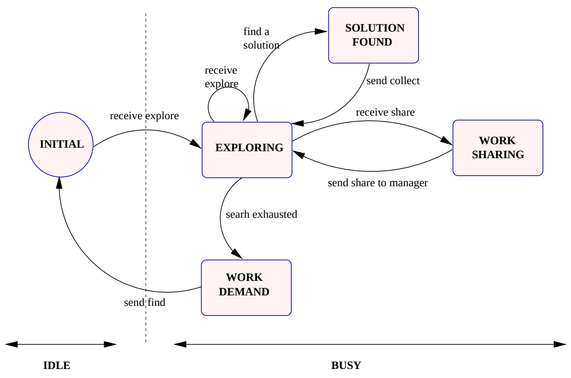

Figure 5: Worker states

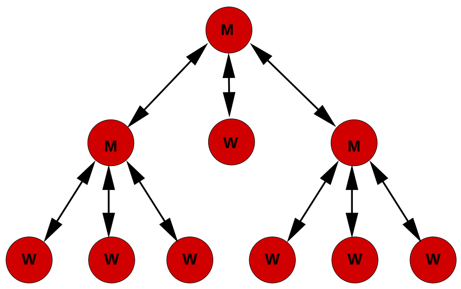

Figure 6: Scaling the architecture

3.3 Store Communication Strategies

A subtree rooted in a node is defined in terms of the state of the constraint store S_1 associated to that node.When an agent has to start exploring at some node, the computation state can be transmitted by sending the whole store or by sending a difference between some previous store S_0 (possibly the initial one) already known to the receiving node and S_1.

Since our approach is based on a CLP($\mathcal{FD}$) implementation which handles constraints at a high-level, using attributed variables [Hol92] which contain first-order terms, sending the store can be made simply by term transmission (which can be optimized by marshaling and compression). Differences can be sent as mere store differences, but this becomes complicated because of the intermediate store simplifications which may have taken place.[3] An approach to overcome this problem is to encode the difference as the (program) operations which were applied to S_0 to yield S_1. These operations are either equation additions (and the associated propagation steps) and labeling steps. Note that the set of labeling steps to be applied to S_0 to give S_1 is deterministic and does not need search. This approach, termed recomputation (in contrast with the former, termed copying) has already been used elsewhere [Sch02, Clo87] both in shared memory and distributed memory implementations of declarative languages with choices, and is, in principle, more amenable to distribution, since constraint stores are usually large. However, granularity considerations can be used both at compile time and at run time to decide which communication method should be used: recomputing from a distant state can become very costly, even if no search is actually performed.

4 Search Tree Generation

In the initial stage of exploration, the search tree is composed by a single node containing the original constraint store. During search, and while there are variables still not enumerated, intermediate nodes can produce new children by (non-deterministically) selecting a variable and (again, non-deterministically) refining its domain. These reductions cause new constraint stores which correspond to new nodes. Domain reduction can be made, in its simplest form, by making assignments of the form $Variable = Value$, where $Value \in Domain(Variable)$, with more constraint addition (e.g., $Variable \leq Mid \vee Variable > Mid$, where, for example, $Mid = \lfloor \frac{\min(Domain(Variable)) + \max(Domain(Variable))}{2} \rfloor$). In general, choosing this kind of domain reduction policy helps to keep the tree balanced, which is desirable for a distributed / parallel computation.

[3]This is, in some sense, similar to the techniques used, at a lower level, in the Muse or-parallel system [AK90], but in that case the heap differences are computed traversing the choicepoint stack, and the Herbrand constraints, even after simplification, have a simpler representation than other domains.

According to the scenario described in section 3, exploration can be seen as a two dimensional process (figure 7), where the first dimension takes place at the manager establishing the basis of the interaction among workers in terms of scheduling, i.e. splitting the search tree into subtrees, deciding which worker explores which subtree, and providing a protocol to obtain another subtree once the current one is exhausted. The other dimension of search is local to each worker and specifies the way in which the search space currently assigned to the worker is explored. Analogously to the sequential standard exploration process in CLP systems, in our implementation, by default though customizable, we have chosen to perform exploration in a width−first manner in the first dimension and depth−first in the local search dimension.

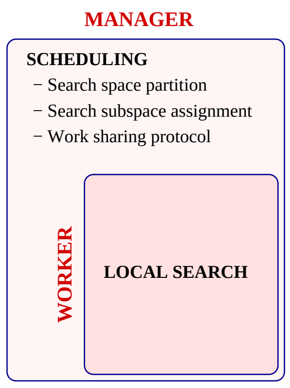

Figure 7: Two dimension search

5 Work Sharing

As a matter of fact, not necessarily do search subtrees in which the scheduling process divides the whole search space have the same size in terms of exploration steps. So, during exploration, when a worker exhausts search in its local scope, either because all the variables in its store are ground or because all the branches have already been (unsuccessfully) explored, it requests more work, which will be provided by another worker in a busy state.

As cooperative entities, whatever decision workers take will always be focused on maximizing global performance. In this case, the worker that gives in part of its search tree to another worker has in fact to decide which this part must be so that the whole system will obtain as much benefit as possible from this transaction. If it is not a good choice, the requesting worker will probably ask for more work in a short time, hence adding extra message passing time overhead by respawning the *share* process each time it goes through its particular search tree. This decision can be:

- Share the most recent node in the search tree
- Share the least recent node
- Share a combination of nodes

The most recent node in a worker's current search space is by definition, given the depth−first search policy used by default, the one corresponding to the branch of the local search tree that has been more explored so far and therefore this node will be due to have the smallest amount of exploration work left to be done. So, in general it will not be a good practice to share it. In contrast, the least recent node will also be the least explored one and, subsequently, the node whose exploration has more search steps left to be taken. This reasoning matches the expected behavior in general but there can be exceptions. Therefore, more intelligent work sharing techniques can be used consisting on drafting, based on a number of heuristics, a combination of new and old nodes that may optimize behavior.

The larger amount of information exchange between system entities (workers and manager) takes place during the process of work sharing. This fact highlights the importance of reducing the size of messages as much as possible. As messages mainly contain constraint stores representing nodes of the search tree, it is necessary to minimize the size of the information used to describe constraints. This is also a good reason to use *Recomputing* instead of *Copying* as it is always cheaper, in terms of bytes transferred, to send a description, in the appropriate representation, of the operations performed over the original store from a previous snapshot to the present state than the whole current store.

6 Implementation

The implementation proposed of this high level model for distributed labeling lies on the (still in development) Ciao Prolog CLP($\mathcal{FD}$)library, and a good number of already existing distributed programming facilities provided by Ciao Prolog as *concurrency* [CH99] and *active modules* [BC01] libraries, as well as some others which are still in development but already offer the required functionality, as library *remote* [CH02].

Under a procedural point of view, this implementation consists of three main entities: *agencies*, *workers*, and *manager*, that interact with one another, as shown in figure 8.

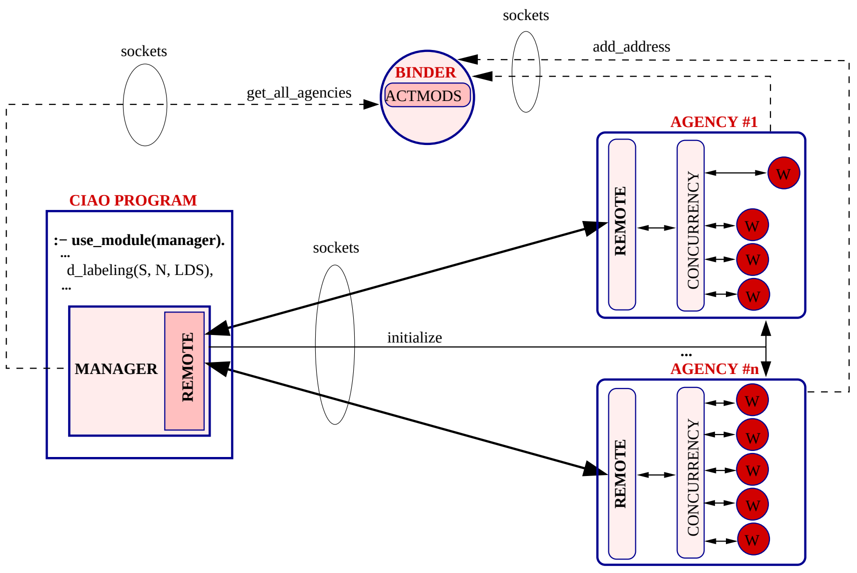

Figure 8: Process interaction in distributed labeling

- An agency is a standalone executable which gives support for and encapsulates any number of workers. Thanks to this intermediate level in the implementation design, it is possible to manipulate workers during execution, i.e. include/create new workers into the system, remove workers from execution and, in general, scale the system.

- Workers kept in an agency use it as a proxy to communicate with the rest of the components, i.e. the manager or the rest of the workers, participating in the distributed labeling process. Workers are implemented as Ciao's threads thanks to the primitives provided in *concurrency*.

- The manager is implemented as a Ciao module which exports a predicate called `d_labeling/3`. This predicate permits to perform distributed labeling on a constraint store upon invocation in an user program. Its first argument is a list containing the variables to be labeled, the second is the number of solutions demanded (e.g, one, 100, all) and the third one is an LDS parameter specifying the number of search steps allowed to be taken against the default worker local search heuristic.

Additionally, it has also been necessary to implement a *binder*, consisting of an active module that connects agencies with the manager and vice-versa. It is a Ciao service reachable across the network which stores the IP addresses of the agencies available. Agencies register themselves in the binder during startup thanks to the predicate `add_address/2` so that, upon user request, the manager can initialize the execution of distributed labeling. The manager, using the binder as a proxy, obtains the agencies addresses via the binder service `get_all_agencies/1`, provides them with its IP address, which agencies keep stored, and makes them initialize the local side of search, i.e. launch the requested workers enabling them to execute the commands sent by the manager.

This implementation offers not only conceptual but also actual physical distribution of execution over the network. Thanks to Ciao libraries *remote* and *active modules*, entities can communicate with one another across the network in a transparent way (applications in grid computing).

7 Empirical results

We will now compare the speedup obtained using an increasing number of workers, including the relation between the execution time of the distributed scheme with only one worker and the sequential case, represented with dashed lines in the figures. In these benchmarks we have chosen recomputing instead of copying as the computation state representation technique, though comparing both is definitely an interesting experiment as in [Sch02], given it helps to provide a more precise vision of the system's behavior. Copying generally needs a more expensive communication between manager and workers than recomputing, due to the size and process time of the messages exchanged (they include the whole constraint store), hence being more sensitive to issues like communication bandwidth availability, which depends on physical environment factors like network and CPU speed. Also, the cost of keeping recomputing information per worker is much lower in time and memory usage than copying.

We have used three different benchmarks to perform a preliminary assessment of the proposed implementation. The first one is the FD version of the N-queens program, where performance for searching one and all solutions was measured. The second one is the cryptoarithmetic problem *DONALD + GERALD = ROBERT* (DGR), where each letter stands for a different digit and they compose three numbers (DONALD, GERALD and ROBERT) which must meet the previous equation. Finally, the third benchmark is a program to find magic squares: square grids of side N which hold numbers from 1 to N^2 and which must add up to the same amount in every horizontal, vertical and (main) diagonal; in this case we found all solutions (for a 3×3 magic square) and one solution (for a 4×4 magic square). All benchmarks have been written trying not to use tricks which would speed up the program for a certain solver; they have all been executed on a 10-processor Sun SparcCenter machine running Solaris 2.5.

The Queens Benchmark Finding all of the solutions (724) of a 10-Queens board (Figure 9) needs a considerable amount of search; therefore, workers are seldom idle and work request messages (*share* and *find*) are scarce. The speedup approaches linearity when the number of workers is small (2 to 3) and starts to decay, due to the increased numbers of messages exchanged among the workers.

Asking for just one solution with, in general, chessboards of any size N can be enough to change completely the behavior of the program (Figure 10). This is due to the large number of solutions of the problem, scattered along the search tree: the first solution to be found sequentially is close enough to the start of the search that using more workers does not help to achieve more speed — this is also partly due to the initial exchange of messages aimed at distributing load among the workers.

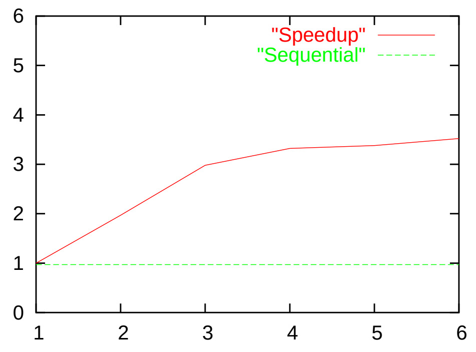

Figure 9: 10 Queens (all solutions)

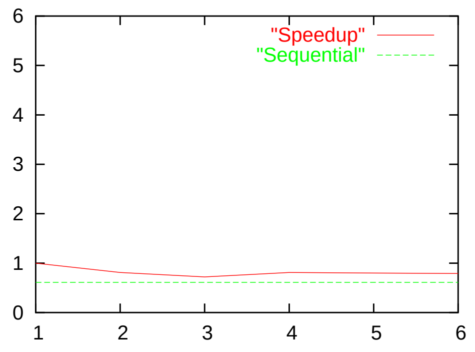

Figure 10: N Queens (1 solution)

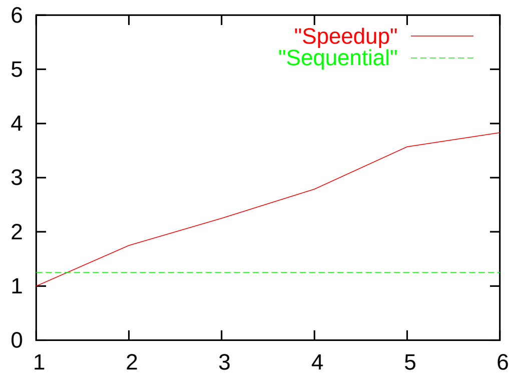

Figure 11: DONALD + GERALD = ROBERT

DONALD + GERALD = ROBERT This problem has only one solution, but a high number of alternatives to be tried during labeling — i.e., the initial propagation cannot cut off many alternatives, and most of the work has to be performed with labeling. The overall situation is similar to that in the 10-Queens case (Figure 9), where all the solutions were found by backtracking. An early work sharing among workers allows an almost independent execution (i.e., few messages are exchanged) and relatively good speedups.

Magic Squares The magic square benchmark for a side of size 3 and all solutions required is a especially good benchmark to show up to what point it is convenient or not to use a certain amount of parallelism during execution, as well as the relationship between speedup and the overhead induced by the work share out process. In this case, all solutions are demanded but this is quite a low number, only eight. Thus, as depicted in figure 12, there is a point where the addition of workers does not suppose an increase of speedup anymore but, on the contrary, it decays. This happens because from five workers and on there is literally no more work amenable to be shared, it has all been already assigned, and the overhead added by the share protocol can not be made up for anymore.

Similarly to the case of searching for one solution in the queens problem, the search space of magic squares has plenty of possible solutions (for example, 7040 for a square of size N = 4). So, it is almost straightforward for any worker to find one in its portion of the search tree and, as can be seen in Figure 13, the increase of workers hardly has any impact in the evolution of speedup, which keeps nearly constant and equal to one, this meaning there is practically no acceleration.

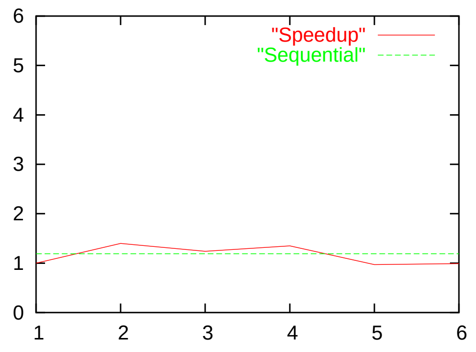

Figure 12: Magic square, (N=3, all solutions)

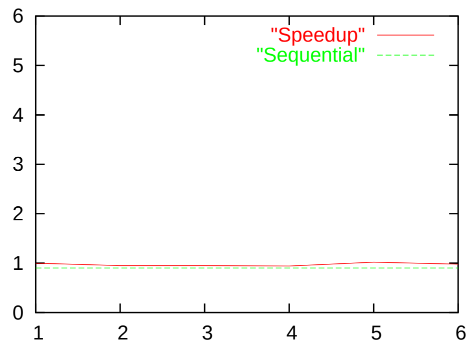

Figure 13: Magic square, (any N, 1 solution)

8 Conclusions and Future Work

During the phase of labeling the search tree of the problem is explored and can be divided into independent subtrees which are amenable to be explored in parallel. We have implemented an scheme which executes in a distributed fashion the “or” part of the exploration performed by the labeling stage of the execution of an CLP($\mathcal{FD}$) program. Subtrees of the search tree are mapped onto agents which can be physically distributed over a network and cooperatively explored. New work generated by the agents can be shared, according to a number of different policies, with the then idle workers. Workers communicate by means of a manager which is sent messages regarding availability of work and requests of work, and which forwards them to the corresponding workers. The manager also keeps track of the overall state of the computation, i.e., whether a solution has been found or not, and whether more solutions are requested.

Work scheduling is a two-layered process which influences the order the tree is explored. Part of the decisions take place in the manager, which establishes which particular worker is assigned a certain subtree. In turn, each worker decides which not-yet-explored branches are to be delegated to other workers. Workers can also decide which search strategy is to be applied to their local tree.

In our experience, recomputing has given us better results than copying in terms of speed and memory consumption. Besides, taking into account which part of the tree was explored by every worker allows making smarter decisions regarding which part of the subtree is to be explored.

Finally, benchmarks which are amenable to speed up, due to their granularity and distribution of solutions, do show a reasonable speedup. This seems to be the case of search trees with few (or just one) solution: when the time to succeed by one worker is not enough as to perform task distribution among workers, the distribution does not have any chance to work in parallel. This is just another incarnation of task granularity problems.

In a future we plan to extend the local search strategy with more possibilities, including different heuristics (e.g., Limited Discrepancy Search [WDH95]). As long as the heuristic do not infinitely visit the same nodes, completeness is ensured thanks to the finiteness of the search space.

Research is also being conducted on an and-parallel scheme where both variables and constraints are distributed over a set of agents across a network.

References

[AK90] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[BC01] F. Bueno and J. Correas. Active module service. Technical Report CLIP4/2001.1, Facultad de Informática, UPM, 2001.

[CCEB03] Andrew A. Chien, Brad Calder, Stephen Elbert, and Karan Bathia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel Distributed Computing*, 2003.

[CH99] M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, USA, November 1999.

[CH02] M. Carro and M. Hermenegildo. A simple approach to distributed objects in prolog. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[Clo87] William Clocksin. Principles of the delphi parallel inference machine. *Computer Journal*, 30(5), 1987.

[FKNT99] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[GPA+01] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

[HBC+99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HBC+00] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The Ciao Logic Programming Environment. In *International Conference on Computational Logic, CL2000*, July 2000.

[Hol92] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, August 1992.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, pages 196–219. University of Melbourne, MIT Press, 1987.

[Leu93] H.F. Leung. *Distributed Constraint Logic Programming*, volume 41. World-Scientific, 1993.

[LGHD96] P. López-García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[MS98] Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[Sch02] Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302. Springer, 2002.

[Van89a] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.

[Van89b] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming. In *Sixth International Conference on Logic Programming*, pages 165–180, Lisbon, Portugal, June 1989. MIT Press.

[WDH95] Matthew L. Ginsberg William D. Harvey. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, pages 607–615, Montréal, Québec, Canada, August 20-25 1995. Morgan Kaufmann, 1995.

[YDIK98] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.