Centrum voor Wiskunde en Informatica

REPORT RAPPORT

SEN

Software Engineering

Software ENgineering

SEN

# A Component-Based Parallel Constraint Solver

P. Zoeteweij, F. Arbab

REPORT SEN-E0314 DECEMBER 17, 2003

Copyright © 2003, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

# A Component-Based Parallel Constraint Solver

ABSTRACT
As a case study that illustrates our view on coordination and component-based software engineering, we present the design and implementation of a parallel constraint solver. The parallel solver coordinates autonomous instances of a sequential constraint solver, which is used as a software component. The component solvers achieve load balancing of tree search through a time-out mechanism. Experiments show that the purely exogenous mode of coordination employed here yields a viable parallel solver that effectively reduces turn-around time for constraint solving on a broad range of hardware platforms.

*1998 ACM Computing Classification System:* D.1.3; D.1.m
*Keywords and Phrases:* component-based software engineering; coordination; parallelization; constraint solving

# A Component-Based Parallel Constraint Solver

Peter Zoeteweij* and Farhad Arbab

CWI, P.O. Box 94079, 1090 GB Amsterdam, the Netherlands
{P.Zoeteweij,Farhad.Arbab}@cwi.nl

**Abstract.** As a case study that illustrates our view on coordination and component-based software engineering, we present the design and implementation of a parallel constraint solver. The parallel solver coordinates autonomous instances of a sequential constraint solver, which is used as a software component. The component solvers achieve load balancing of tree search through a time-out mechanism. Experiments show that the purely exogenous mode of coordination employed here yields a viable parallel solver that effectively reduces turn-around time for constraint solving on a broad range of hardware platforms.

**Keywords:** component-based software engineering, coordination, parallelization, constraint solving.

## 1 Introduction

Over the past years, constraint solving has been a useful test case for coordination techniques. [3, 8, 9, 14]. One of the reasons is that because of the general nature of the constraint programming paradigm, any constraint solver inevitably supports only a specific class of constraint satisfaction problems (CSP's). By having different solvers, supporting different classes of specialized problems cooperate to solve a more general problem, a broader range of problems can be addressed. The goal of having stand-alone constraint solvers cooperate in a uniform and structured way brings solver cooperation into the area of coordination programming and component-based software engineering.

Constraint solving is an NP-complete problem. Efficient algorithms exist for some classes of CSP's, and when completeness is not important we may be able to find a solution to a CSP quickly by using local search techniques. Nevertheless, generally constraint solving comes down to tree search. In this paper, we deal with a specific mode of solver cooperation that aims at reducing the turn-around time of constraint solving through parallelization of tree search. Contrary to other modes of solver cooperation, parallel constraint solving has received little attention from a coordination point of view.

The primary aspect of our approach is to equip a tree search based constraint solver with a time-out mechanism. When a CSP can be solved before the elapse of a given time-out, such a solver simply produces all solutions that it has found

* Supported by NWO, the Netherlands Organization for Scientific Research, under project number 612.069.003.

(or *the* solution that it has found, if we are not interested in all solutions). Otherwise it also produces some representation of the work that still needs to be done. For tree search, this is a collection of (disjunct) subproblems that must still be explored: the search frontier. These subproblems are then re-distributed among a set of identical solvers that run in parallel. The initial solver is part of this set, and each solver in the set may split its input into further subproblems, when its time-out elapses. The aim of the time-out mechanism is to provide an implicit load balancing: when a solver is idle, and there are currently no subproblems available for it to work on, another solver is likely to produce new subproblems when its time-out elapses. We expect to be able to tune the time-out value such that it is both sufficiently small to ensure that enough subproblems are available to keep all solvers busy, and sufficiently large to ensure that the overhead of communicating the subproblems is negligible. The idea of using time-outs is quite intuitive, but to our knowledge, its application to parallel search is novel.

Rather than a parallel algorithm, we present this scheme as a pattern for constructing a parallel constraint solver from component solvers. The only requirement is that these components can publish their search frontiers. We believe that this requirement is modest compared to building a parallel constraint solver from scratch. Our presentation of the scheme in Section 3 uses the notion of abstract behavior types, and the Reo coordination model. These, and the relevant aspects of constraint solving are introduced in Section 2. To test the concept, we equipped a constraint solver with the time-out mechanism, and implemented the coordination pattern as a stand-alone distributed program. In Section 4 we give an account of this implementation, and in Section 5 we describe the experiments that were performed to test the parallel solver. Compared to parallelizing an existing constraint solver, the component-based approach has further benefits. These are discussed in Section 6, together with related work and directions for future research.

## 2 Preliminaries

To make the paper self-contained, in this section we provide the necessary background on constraint solving (2.1), abstract behavior types, and Reo (2.2).

### 2.1 Constraint Solving

Constraint solving deals with finding solutions to constraint satisfaction problems. A CSP consists of a number of variables and their associated domains (sets of possible values), and a set of constraints. A constraint is defined on a subset of the variables, and restricts the combinations of values that these variables may assume. A solution to a CSP is an assignment of values to variables that satisfies all constraints. Tree search in constraint solving performs a systematic exploration of assignments of values to variables: at every node of the search tree, the descendant nodes are generated by assigning different subdomains to some variable.

The search tree is expanded as a part of the traversal of the tree, but before generating a next level of nodes, we try to limit the number of possible alternatives. This is called pruning the search tree, and for constraint solving, this is done by *constraint propagation*. The purpose of constraint propagation is to remove from the variable domains the values that do not contribute to any solution. For example if two integer variables $x, y \in [0..10]$ are constrained by $x < y$, we may remove 10 from the domain of $x$, and 0 from the domain of $y$. Constraint propagation is usually implemented by computing the fixpoint of a number of reduction operators [1]. These operators are functions (domain reduction functions, DRF's) that apply to the variable domains, and enforce the constraints. If the domains of one or more variables become empty as a result of constraint propagation, the node of the search tree for which this happens is a failure. If, on the other hand, after constraint propagation all domains are singleton sets, these domains constitute a solution to the CSP. In all other cases, the node of the search tree is an internal node, and constraint solving proceeds by branching.

Important concerns in this branch-and-prune approach to constraint solving are the choice of the variable for branching, and how to construct the subdomains for that variable. In this paper we assume a *fail-first* variable selection strategy, where a variable is selected for which the number of possible assignments is minimal. Subdomains are constructed by *enumeration*: they are singleton sets, one for every value in the original domain.

Branching adds new nodes of the search tree to the set of nodes where search may continue. This set is called the *search frontier* [10]. Managing the search frontier as a stack effectively implements a depth-first traversal. Other traversal strategies exist, but depth-first keeps the search frontier small. Apart from memory requirements this is especially important for our application, because the size of the search frontier determines the communication volume of the parallel solver.

## 2.2 Coordination and Abstract Behavior Types

In our view, coordination programming deals with building complex software systems from largely autonomous component systems. The more autonomous these components are, the more it becomes justified to refer to their composition as coordination. Contrary to modules and objects, which are the counterparts of components in the classical software engineering paradigms of modular and object-oriented programming, an instance of a prospective software component has at least one thread of control. For the purpose of composition, the component is a black box, that communicates with its environment through a set of ports. Coordination programming involves writing the "glue code" to actually make the component instances cooperate. Depending on the complexity of the interaction, it may make sense to use a dedicated coordination language. For example if the population of processes is highly dynamic, the Manifold coordination language [2] may be a logical choice.

A software system that complies with the above notion of a component can be specified conveniently by an abstract behavior type (ABT) [4]. ABT's are the coordination counterpart of abstract data types, as used in classical software engineering. Before we can introduce ABT's we first need to recall the definition of timed data streams from [5].

A *stream* over some set $A$ is an infinite sequence of elements of $A$. Zero-based indices are used to denote the individual elements of a stream, e.g., $\alpha(0)$, $\alpha(1)$, $\alpha(2)$, ... denote the first, second, third, etc. elements of the stream $\alpha$. Also $\alpha^{(k)}$ denotes the stream that is obtained by removing the first $k$ values from stream $\alpha$ (so $\alpha(0)$ is the head of the stream, and $\alpha^{(1)}$ is its tail). Relational operators on streams apply pairwise to their respective elements, e.g., $\alpha < \beta$ means $\alpha(0) < \beta(0)$, $\alpha(1) < \beta(1)$, $\alpha(2) < \beta(2)$, ...

A *timed data stream* over some set $D$ is a pair of streams $\langle \alpha, a \rangle$, consisting of a data stream $\alpha$ over $D$, and a time stream $a$ over the set of positive real numbers, and having $a(i) < a(j)$, for $0 \leq i < j$. The interpretation of a timed data stream $\langle \alpha, a \rangle$ is that for all $i \geq 0$, the input/output of data item $\alpha(i)$ occurs at "time moment" $a(i)$.

An *abstract behavior type* is a (maximal) relation over timed data streams. Every timed data stream involved in an ABT is tagged either as its input or output. For an ABT $R$ with one input timed data stream $I$ and one output timed data stream $O$ we use the infix notation $I\ R\ O$. Also for two such ABT's $R_1$ and $R_2$, let the composition $R_1 \circ R_2$ denote the relation $\{\langle \langle \alpha, a \rangle, \langle \beta, b \rangle \rangle \mid \exists \langle \gamma, c \rangle \cdot \langle \alpha, a \rangle R_1 \langle \gamma, c \rangle \wedge \langle \gamma, c \rangle R_2 \langle \beta, b \rangle\}$.

ABT's specify only the black box behavior of components. For a model of their implementation, other specification methods are likely to be more appropriate, but that information is irrelevant from a coordination point of view.

Reo [4, 6] is a channel-based exogenous coordination model wherein complex coordinators, called *connectors* are compositionally built out of simpler ones. The simplest connectors in Reo are a set of *channels* with well defined behavior. In Section 3.2 we use Reo connectors to specify the coordination of our component solvers.

## 3 Specification

### 3.1 Component Solver

In this section we define an ABT for a constraint solver with the time-out mechanism. First we need some formal notion of a CSP:

Let $P$ be a finite set of *problems* and let $(P \cup \{\top\}, \sqsubseteq)$ be a partial order such that for all $p \in P$, $\top \not\sqsubseteq p$. For a problem $p \in P$, we define the sets $sub(p) = \{q \in P \cup \{\top\} \mid p \sqsubseteq q\}$ and $sol(p) = \{s \in sub(p) \mid \forall q \in sub(p) \setminus \{s\} \cdot s \not\sqsubseteq q\} \setminus \{\top\}$. Intuitively, $sub(p)$ represents the set of subproblems of a problem $p$, possibly including $\top$, which represents the deduction of a failure. The set of maximal subproblems excluding $\top$, $sol(p)$, represents the set of solutions of $p$.

Next we specify that a constraint solver transforms a problem into a set of mutually disjunct problems. Let $D$ denote the data domain $P \cup 2^P \cup \{\tau\}$, where $\tau \notin P$ is an arbitrary data element that serves as a *token.* In the following, let $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$ be timed data streams over $D$. Now the behavior of a *basic solver* is captured by the *BSol* ABT, defined as

$$\langle \alpha, a \rangle \ \mathit{BSol} \ \langle \beta, b \rangle \equiv \ a < b \wedge S(\alpha, \beta)$$

where $S$ is a relation on $P$ and $2^P$, such that for all $p \in P$ and $R \in 2^P$, $S(p, R)$ iff

- $\forall r \in R$, $p \sqsubseteq r$,
- $\forall r, s \in R$, $r \sqsubseteq s$ implies $r = s$, and
- $sol(p) = \cup_{r \in R} sol(r)$

The *Str* (streamer) ABT specifies that stream of sets of problems, as produced by a basic solver, is transformed into a stream of problems, where the sequence of problems for each input set is delimited by a token:

$$\begin{aligned} \langle \alpha, a \rangle \ \mathit{Str} \ \langle \beta, b \rangle \equiv \ & a(0) = b(0) \\ & \wedge \ \beta(k) = \tau \\ & \wedge \ \alpha(0) = \{\beta(0), \ldots, \beta(k-1)\} \\ & \wedge \ \langle \alpha^{(1)}, a^{(1)} \rangle \ \mathit{Str} \ \langle \beta^{(k+1)}, b^{(k+1)} \rangle \end{aligned}$$

where for all $i \in \mathbb{N}$, $\alpha(i) \in 2^P$ and $\beta(i) \in P \cup \{\tau\}$, and $k$ denotes $|\alpha(0)|$, the cardinality of the set of problems at the head of stream $\alpha$. Now the behavior of a constraint solver component is captured by the *Sol* ABT, defined as

$$\mathit{Sol} = \mathit{BSol} \circ \mathit{Str}$$

Our top-level model of a solver component is the composition of a basic solver and a streamer. The token $\tau$ can be thought of as the notification "no" that a PROLOG interpreter would produce to indicate that no (more) solutions have been found. If we model a typical constraint solver as a basic solver, then for any given input problem the output set corresponds to the set of solutions for that problem, i.e. $\beta(i) = sol(\alpha(i))$, and there is no upper bound on the time $b(i) - a(i)$ needed to produce this set.

In contrast, the load-balancing solver component that we propose here stops searching for solutions after the elapse of a time-out $t$. At that moment, it generates a subproblem for every solution that it has found, plus one for every node of the search tree that must still be explored. For $t \in \mathbb{R}^+$, the $\mathit{Sol}_t$ ABT defines this behavior:

$$\begin{aligned} \langle \alpha, a \rangle \ \mathit{BSol}_t \ \langle \beta, b \rangle \equiv \ & \langle \alpha, a \rangle \ \mathit{BSol} \ \langle \beta, b \rangle \\ & \wedge \ \forall i \in \mathbb{N} \cdot b(i) - a(i) < t \end{aligned}$$

$$\mathit{Sol}_t = \mathit{BSol}_t \circ \mathit{Str}$$

## 3.2 Parallel Solver

Figure 1 shows a channel-based design for a (3-way) parallel solver. All channels in this design are synchronous: read and write operations block until a matching operation is performed on the opposite channel end. The "resistors" depict Reo filters: synchronous channels that accept data items that match a certain pattern (set of allowable data items) and discard data items that do not match this pattern. At node b in Figure 1, all output of the solvers is replicated onto two filters. Channel bc filters out solutions. Its pattern (p) is $\texttt{Filter}(\{p \in P \mid sol(p) = \{p\}\})$. The channel from b to T discards all solutions. Its pattern (q) is $\texttt{Filter}(\{p \in P \mid sol(p) \neq \{p\}\} \cup \{\tau\})$. The ABT's of the channels are specified in [4].

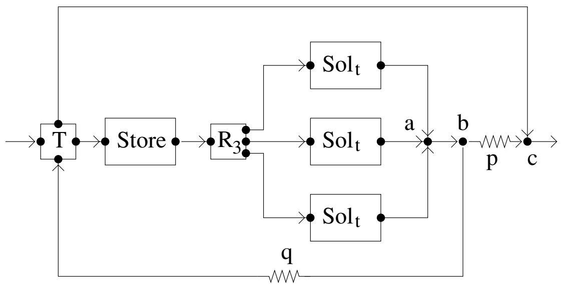

**Fig. 1.** 3-way parallel solver

Apart from the channels and the three load-balancing solvers $Sol_t$, there are three elements of the design that require further clarification: the special-purpose connector T, the 3-ary exclusive router $R_3$, and the Store. We do not give full ABT's, but an intuitive description.

The special purpose connector T implements termination detection. Initially, it reads a problem from its left-hand side input port. All problem descriptions entering T, through either input port, are forwarded immediately through its right-hand side output port to the Store. Also T counts the number of problems forwarded to the Store, and the number of tokens $\tau$ received through its bottom port (from node b). While these numbers do not match, the parallel solver is busy, and T will accept new (sub)problems from its bottom input port (connected to node b) only. As soon as the number of problems is canceled out by the number of tokens, T sends a token $\tau$ through its top port (to node c), indicating that the parallel solver has finished working on its current problem. Then it will return to its initial state, and accept a new problem from its left-hand side input port.

Connector $R_3$ is a general-purpose 3-ary exclusive router. It operates synchronously, and every data item on its input port is forwarded on exactly one of its output ports. If none of the channels connected to the output ports is able to forward a data item, the router blocks. If a data item item can be forwarded on more than one output port, a non-deterministic choice is made. Construction of the exclusive router from Reo primitives is shown in [6].

The Store is a channel-like connector that is specific to this application. Its operation is asynchronous: it buffers incoming problems, and interprets these problems to determine the level of the corresponding node of the search tree. This information can be used to enforce a global traversal strategy. When $R_3$ is ready to accept data (i.e., when one of the load-balancing solvers has become idle) it forwards a problem according to this strategy. For example, it may forward a node of the deepest available level in an attempt to implement depth-first search globally. This effectively drains the Store. Forwarding a node of the shallowest available level implements breadth-first search, filling up the Store with more subproblems.

## 4 Implementation

To test the proposed implementation of parallel search, we equipped our OpenSolver constraint solver with the time-out mechanism, and developed a distributed program to combine several such solvers into a parallel constraint solver.

### 4.1 Component Solver

OpenSolver is an experimental constraint solver that implements a branch-and-prune tree search algorithm. This algorithm is abstract in the sense that its actual functionality is determined by software plug-ins in a number of predefined categories, corresponding to different aspects of the model of constraint solving outlined in Section 2.1. For example, there are categories for variable domain types and domain reduction functions. OpenSolver has been developed with coordination in mind: a special category of plug-ins covers the *coordination layer* of the solver (Figure 2). Through a plug-in in this category, the execution of the solving algorithm can be controlled, and data can be shared with the environment. In addition to the coordination layer plug-in described below, a plug-in exists that implements a simple user interface, and a plug-in is being developed that will allow an OpenSolver to participate in a general framework for distributed constraint solving [15].

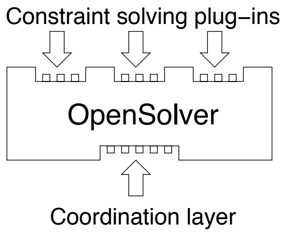

**Fig. 2.** Plug-ins determine the actual functionality of the OpenSolver

The plug-in mechanism is implemented by inheriting from C++ abstract classes for each of the categories. It is explained in more detail in [14]. The coordination layer is implemented by having the solver execute a command loop, where it continually asks the coordination layer plug-in what to do next. Examples of commands that can be given by the coordination layer plug-in are:

– from the search frontier, select a set of nodes for further exploration,
– perform constraint propagation in the nodes of the search tree that have been selected for further exploration,
– apply the branching strategy plug-in to the nodes where constraint propagation has finished, in order to expand the search tree,
– flush the nodes of the search tree: this generates a textual representation of the search frontier and of all solutions that are available. The data structures for these nodes are then deallocated.

The latter command is important for implementing the time-out mechanism. A special coordination layer plug-in `StreamingIO` has been developed, that turns an OpenSolver into a load-balancing solver, as specified in Section 3. When it is equipped with this plug-in, an OpenSolver instance keeps reading problem descriptions from its standard input. These problem descriptions are coded as sequences of ASCII characters where the individual problem descriptions are delimited by brackets. An example of a problem description is shown in Figure 3.

```
VARIABLE q1 IS DiscreteDomain {1..4};
VARIABLE q2 IS DiscreteDomain {1..4};
VARIABLE q3 IS DiscreteDomain {1..4};
VARIABLE q4 IS DiscreteDomain {1..4};
DRF DDNEQ { q1-q2 <> 0 }; DRF DDNEQ { q1-q2 <>-1 };
                             DRF DDNEQ { q1-q2 <> 1 };
DRF DDNEQ { q1-q3 <> 0 }; DRF DDNEQ { q1-q3 <>-2 };
                             DRF DDNEQ { q1-q3 <> 2 };
...
DRF DDNEQ { q3-q4 <> 0 }; DRF DDNEQ { q3-q4 <>-1 };
                             DRF DDNEQ { q3-q4 <> 1 };
```

**Fig. 3.** Part of a problem description for the 4-queens puzzle

The problem descriptions themselves contain instructions for the solver to activate plug-ins for variables, DRF's, etc. When a problem description has been read from standard input, the coordination layer plug-in instructs the solver to parse it, and subsequently starts the search for solutions. When the time-out elapses, or when the search frontier becomes empty, the `StreamingIO` plug-in stops issuing commands that drive the search for solutions. Instead it issues the command to flush the nodes of the search tree. Our implementation is an approximation of the $Sol_t$ ABT of Section 3.1 in the sense that the subproblems appear on the output stream just *after* the time-out elapses. In theory this could take unacceptably long, for example if suddenly the workload of the system increases. In practice the approximation is realistic.

Every plug-in implements a method to write itself into a character string. When executing the command to flush the search tree, this method is called for all plug-ins that define a particular node of the search tree, notably the variable domains and the DRF's. These strings are then passed to the coordination layer. Normally this mechanism is used to produce the solutions of a CSP, but because we don't perform an exhaustive search, in this case it also produces the search frontier. This information is used by the `StreamingIO` coordination layer plug-in to construct new problem descriptions, that are written to standard output. After the flushing operation is complete, the coordination layer plug-in generates a character-encoded token $\tau$, and proceeds by reading a new problem specification from standard input. Except for the token, the output of this coordination layer plug-in can be directly fed into another solver as a stream of problem descriptions.

The component solvers are configured to perform a depth-first traversal of the search tree, but through a special category of plug-ins, problem descriptions are annotated with the level of the corresponding node in the search tree. This allows the master process, implementing the Store of Section 3.2, to impose a high-level traversal strategy on top of the depth-first traversal of the solvers.

An important aspect of a constraint solver implementation is the construction of the data structures, notably the variable domains, that define the node of the search tree where search continues. While hybrid methods have been studied, the main options are [11]:

**Copying** When the search tree is expanded by branching, the data structures that define the current node are copied for all new nodes. These copies are then modified to construct subproblems. At potentially high memory costs, every node of the search frontier is immediately available for further exploration.

**Trailing** Only the current node of the traversal is maintained, but all changes (deletions of values) to the domains of variables leading up to this node are registered. Backtracking is implemented by undoing changes to reach an internal node of the search tree, from which search can progress along an alternative branch. Trailing is the predominant method used in current constraint solvers.

**Recomputation** Internal nodes are represented by the branching choices that were made in order to arrive at that node. Instead of unwinding a trail of changes, the internal nodes are reconstructed from a shallower internal node by repeating a part of the traversal of the search tree.

OpenSolver is based on copying, so the search frontier is maintained explicitly (but plug-ins in the variable domain type category can implement a copy-on-write policy). Admittedly, this is a great convenience for publishing the search frontier, but we are convinced that our method extends to solvers that use trailing or recomputation. Especially when searching for all solutions, every node of the search tree must be generated eventually, so no extra work is involved if this is done for the current search frontier when the time-out elapses.

## 4.2 Glue Code

We implemented the coordination protocol of Section 3.2 as a master-slave distributed program coded in C using the MPI message passing interface. Without the facilities for gathering statistics, the size of this glue code is just a little more than 600 lines. The slave processes fork a new UNIX process to start the component solvers, and a pair of pipes is connected to the standard input and output of these processes to facilitate the character-based implementation of the timed data streams.

The channels of the coordination model are implemented by directed send and receive MPI calls. Upon reception of a token $\tau$, a new subproblem is sent to the solver that generated the token. For this purpose, the character-based encoding of the token contains the identity of this solver. Also the number of solutions counted for each subproblem is piggybacked on the token.

When reading from the pipe that is connected to the standard output of a solver, the slave processes perform some parsing to recognize the beginning of a new problem description. At this point, an entire problem is sent to the master process as a character string. The master process implements the distribution and gathering of the problems. Figure 4 illustrates this software architecture.

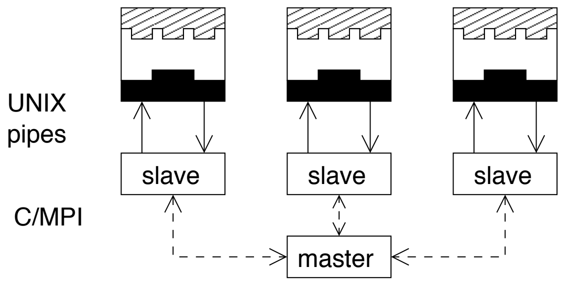

**Fig. 4.** Software architecture of the parallel solver

Note that the component solvers are still stand-alone applications that rely on character-based standard I/O only. Our primary goal was a performance evaluation of the time-out mechanism, and from that perspective, a master-slave implementation is acceptable. However, the channel-based design of section 3.2 has many advantages over this rigid scheme. In particular, the decision where to send the next subproblem is now taken on the basis of solver output, whereas a true implementation of the exclusive router would be able to detect that a solver is idle when the channel connecting to that solver is ready to accept new data. This has the benefit of a better separation of concerns and of a reusable solution.

## 5 Experiments

The parallel solver was tested on three combinatorial problems:

*Queens* An instance of the $n$-queens problem, where $n$ queens must be placed on an $n \times n$ chess board, such that no two queens attack each other. Figure 3 shows a problems description for $n = 4$. The results reported here are for $n = 15$, for which there are 2279184 solutions.

*Sat* The problem is to find an assignment of truth values to propositional variables such that a given propositional formula in conjunctive normal form is satisfied. Such a formula is a conjunction of clauses, where a clause is a disjunction of literals (a propositional variable or its negation). In our model for this problem we use a constraint for every clause of the formula, which states that at least one of the literals of that clause must be true. A special-purpose DRF plug-in has been developed, which is initialized by a sequence of positive literals and a sequence of negative literals. When all except one literal have been assigned the value false, this DRF plug-in removes the value that would render the last literal false from the domain of that literal's variable. For these experiments we used formula par16-2-c from the dimacs test set[1]. This formula has 1392 clauses on 349 variables.

*Coloring* This is a graph coloring problem. In general, the problem is to find an assignment of colors to the vertices of a graph, such that two vertices that are connected by an edge have different colors. Here we verify that no 9 coloring exists for graph DSJC125.5[2] from the dimacs test set, having 125 nodes and 3891 edges. In our model we use a variable for every node, and a disequality constraint for every edge. The disequalities are implemented using the `DDNEQ` DRF plug-in of Figure 3.

In all cases, we used a fail-first variable selection strategy, selecting a variable with the smallest number of alternative values. As a second criterion for *Coloring*, variables are ordered according to the degree of their corresponding nodes of the graph. The component solvers perform a depth-first traversal, but using the level annotation of the problem descriptions generated by the solvers, the master switches between breadth-first and depth-first traversal, depending on the number of available subproblems. If this number is below a certain threshold value (512, for these experiments) priority is given to the shallowest available

[1] available from e.g. `http://www.lri.fr/~simon/satex` (the Sat-Ex website).
[2] available from e.g. `http://mat.gsia.cmu.edu/` (Michael Trick's Operations Research Page.)

nodes. These are least likely to complete within the time-out, and can thus be expected to increase the number of problems available to the master, making it easier to keep all solvers busy. Also, when the full problem is first submitted to the first solver, this solver uses a very small time-out in order to generate work for the other solvers quickly. Finally it should be mentioned that in our test runs, solutions are only counted, not stored or communicated.

Table 1 shows the sequential and parallel runtimes (elapsed time) for our test problems, as well as the parallel efficiency, which is the actual speedup divided by the number of processors. As an indication that our solver is a realistic implementation, depending on the search strategy, the standard example for 15-queens in ECLiPSe [3] 5.5 completes in 900 - 1500 sec. on the same hardware. The speedup figures (sequential runtime divided by parallel runtime) are shown in Figure 5. All elapsed times shown are averages of 10 repeated runs on a Beowulf cluster built from 1200 MHz Athlon nodes. The entries for parallel runs on 1 processor are an indication of the overhead of the time-out mechanism. For *Queens* and *Sat* we used a time-out value of 3200ms. For *Coloring* we used 9600ms. The master process always runs on the same node as one of the component solvers.

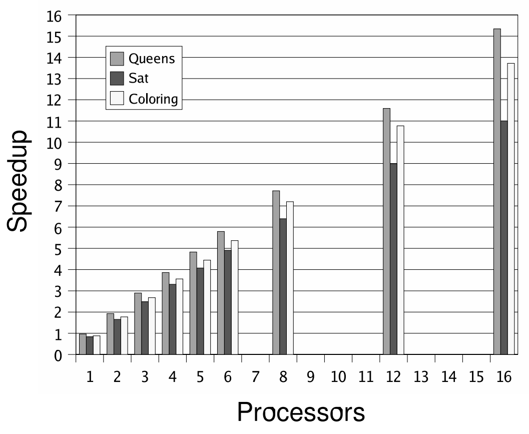

**Fig. 5.** Speedup figures

As can be seen from Figure 5, our parallel solver scales well. For *Queens* and *Coloring*, the parallel efficiency remains practically constant for the numbers of processors that we have tested with, and the scalability can be expected to extend to higher numbers of processors. The difference in efficiency for these two series of runs, and for the *Sat* runs on lower numbers of processors can

[3] ECLiPSe Constraint Logic Programming System. See `http://www-icparc.doc.ic.ac.uk/eclipse`

| | Seq | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 12 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|
| *Queens* | 734.16 | 760.79 | 380.85 | 253.41 | 190.04 | 152.01 | 126.67 | 95.18 | 63.32 | 47.86 |
| eff. | | 0.96 | 0.96 | 0.97 | 0.97 | 0.97 | 0.97 | 0.96 | 0.97 | 0.96 |
| *Sat* | 1541.12 | 1842.55 | 931.26 | 619.65 | 466.08 | 378.14 | 313.91 | 240.91 | 171.43 | 140.02 |
| eff. | | 0.84 | 0.83 | 0.83 | 0.83 | 0.82 | 0.82 | 0.80 | 0.75 | 0.69 |
| *Coloring* | 419.29 | 475.92 | 236.50 | 156.47 | 117.70 | 94.31 | 78.11 | 58.23 | 38.92 | 30.56 |
| eff. | | 0.88 | 0.89 | 0.89 | 0.89 | 0.89 | 0.89 | 0.90 | 0.90 | 0.86 |

**Table 1.** Elapsed times (sec.) and parallel efficiency

be explained by the different sizes of the problem representations, and their associated communication costs.

For *Sat*, parallel efficiency drops after 8 processors. The reason is that because the variable domains are binary, the search frontiers are smaller than for the other two problems, and the master has difficulty keeping all solvers busy. Also the problem seems to have a less balanced search space: submitting a shallow subproblem to one of the solvers is less likely to generate new nodes than for *Queens* and *Coloring*. We hope to remedy the problem of the binary search trees by using a special-purpose branching strategy plug-in, which instantiates several variables at the same time, thus generating larger search frontiers. However, this strategy will also generate assignments that would otherwise have been prevented by constraint propagation, so it is hard to predict the overall effect.

The *Queens* experiments have also been run overnight on several (mostly idle) workstations connected by a local area network. While a detailed analysis of these experiments has not been made, here too we saw good speedup and scalability. Our approach seems well suited for such an environment: because no solver will work longer than the specified time-out before sharing work with other solvers, the proposed implementation of parallel search will likely be insensitive to the existing load and heterogeneity of the hardware. Because good results were obtained on a cluster (distributed memory), the parallel solver can also be expected to perform well on shared memory machines.

## 6 Discussion and Directions for Future Work

In [14] we described a constraint solver based on the coordination protocols of [8, 3]. This solver implements distributed constraint propagation using channel-based communication. In [15] we presented a design for extending this system to support a large variety of solver cooperation schemes, including parallel search for which we suggested the time-out mechanism that is evaluated in the present paper. OpenSolver is intended to play the role of the main component solver in this extended system. In addition to the coordination protocol of [8], [9] describes an IWIM implementation of a system for the coordination of heterogeneous solvers.

Other approaches to parallel constraint solving often use a scheme where the parallel solvers exchange nodes of the search tree only when one of them becomes idle, see for example [10, 12]. For such schemes, solvers can potentially run for a long time without having to respond to a request for work from other solvers, but once a solver becomes idle, it may be more difficult to find another solver that is willing to share part of its search frontier. In contrast, our approach aims at having a large repository of work, assuming that the time-out can be tuned such that publishing the search frontier is relatively cheap. From a software engineering point of view it is simpler, and better suited for a component-based implementation, but from a user's point of view, our scheme is more complicated because it introduces a tuning factor.

In [7] a shared-memory scheme is described where first the original CSP is split by assigning values to variables in a generate and test phase, in order that a large set of subproblems is available. These problems are then solved in a data-parallel way, using either a static or dynamic partitioning. We expect that scheme to be more sensitive to load imbalance because it is possible that most of the work is concentrated in only a few of the generated subproblems. For all alternatives discussed here, a comparison of reported efficiency results is difficult, because the hardware platforms and the benchmark problems used in each case are simply too diverse.

As an alternative to implementing the time-out mechanism in the component solver, we could move this mechanism into the glue code. It would be equally easy to modify a constraint solver to respond to some interrupt, and somehow an interrupt mechanism seems less alien to constraint solving than a time-out mechanism. In both cases the solver must be able to publish the state of its search algorithm, for which we use a character-based encoding. There are other advantages to enabling a solver to publish its search frontier. For instance, it allows user interaction in constraint solving, e.g. for computational steering, and supports a mechanism for checkpoints. When the set of subproblems held by the master process is saved to disk at regular intervals, and subproblems are not discarded until their results have been processed, the solver can restart from the last saved set of subproblems after, for example, a power failure has occurred.

Constraint solving was used as an example application, but our method can probably be applied to other problems that involve tree search. This is not surprising, because for many such problems, there exists a more or less efficient encoding as a constraint satisfaction problem. However, some problems that involve tree search have special requirements. For example in optimization we try to minimize some cost function during search. This can be implemented as a branch-and-bound algorithm, to prevent the exploration of subtrees that cannot improve the current best value found for the cost function. Our first goal is to adopt our method for branch-and-bound. Such an algorithm has been studied from a coordination point of view in [13], but in our work, the emphasis is on the component side rather than on the coordination framework, and on the demonstration of a realistic implementation. We expect that our parallel solver can be adapted for branch-and-bound by inserting a dedicated solver into the loop of Figure 1, to record the best value for the cost function found for a solution, and to filter out any nodes produced by the other solvers that will not improve on this bound. Some special care should be taken to communicate new bounds to other solvers.

As a further example, specialized solvers for the propositional satisfiability problem rely on so called learning search algorithms, that derive new constraints during the traversal of the search tree. These constraints are redundant, but when they are made explicit they achieve a stronger pruning of the search tree. It is not directly clear how our method should be extended to facilitate learning solvers, and this is a subject for future research.

## 7 Conclusion

In this paper we proposed an implementation of parallel tree search in constraint solving based on time-outs. Instead of a parallel algorithm, we presented and implemented the method as a protocol for the coordination of multiple instances of a component solver. After equipping a constraint solver with the time-out mechanism, some 600 lines of C/MPI code were sufficient to coordinate several of these component solvers to perform parallel search. Experiments showed that a good speedup is obtained on 2 to 16 CPU's, which indicates a good load balance. We conclude that:

– The time-out mechanism is an effective way to implement parallel search in constraint solving.
– Once a solver is able to publish its search frontier, building a parallel constraint solver becomes a matter of component-based software engineering.
– The OpenSolver plug-in mechanism made it very easy to meet this requirement.

## References

1. K.R. Apt. The Rough Guide to Constraint Propagation. In Jaffar (ed.) *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, LNCS 1713, pp. 1–23, Springer-Verlag, 1999.
2. F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In Ciancarini, Hankin (eds.) *Coordination Languages and Models*, LNCS 1061, pp. 34–56, Springer-Verlag, April 1996.
3. F. Arbab, E. Monfroy. Distributed Splitting of Constraint Satisfaction Problems. In Porto, Roman (eds.) *Coordination Languages and Models*, LNCS 1906, pp. 115–132, Springer-Verlag, 2000.
4. F. Arbab. Abstract Behavior Types: A Foundation Model for Components and Their Composition. In De Boer, Bonsangue, Graf, De Roever (eds.) *Formal Methods for Components and Objects*, LNCS 2852, Springer-Verlag, 2003.
5. F. Arbab, J.J.M.M. Rutten. *A Coinductive Calculus of Component Connectors*. Technical report SEN-R0216, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, September 2002.

6. F. Arbab, C. Baier, J.J.M.M. Rutten, M. Sirjani. Modeling Component Connectors in Reo by Constraint Automata (extended abstract). In *Proceedings of the 2nd International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2003)*, to appear on ENTCS.
7. Z. Habbas, M. Krajecki, D. Singer. *Shared Memory Implementation of Constraint Satisfaction Problem Resolution*. Parallel Processing Letters, Vol. 11, No. 4 (2001), pp. 487–501.
8. E. Monfroy. A Coordination-based Chaotic Iteration Algorithm for Constraint Propagation. In Carroll, Damiani, Haddad, Oppenheim (eds.) *Proceedings of the 2000 ACM Symposium on Applied Computing*, pp. 262–269, ACM Press.
9. E. Monfroy, F. Arbab. Constraints Solving as the Coordination of Inference Engines. In Omicini, Zambonelli, Klusch, Tolksdorf (eds.) *Coordination of Internet Agents: Models, Technologies and Applications*, Springer-Verlag, 2001.
10. L. Perron. Search Procedures and Parallelism in Constraint Programming. In Jaffar (ed.) *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, LNCS 1713, pp. 346–360, Springer-Verlag, 1999.
11. C. Schulte. Comparing Trailing and Copying for Constraint Programming. In De Schreye (ed.) *Proceedings of the Sixteenth International Conference on Logic Programming*, Las Cruces, NM, USA, pp. 275–289, The MIT Press, November 1999.
12. C. Schulte. Parallel Search Made Simple. In Beldiceanu et al. (eds.) *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, September 2000.
13. A. Stam. A Framework for Coordinating Parallel Branch and Bound Algorithms. In Arbab, Talcott (eds.) *Coordination Models and Languages*, LNCS 2315, pp. 332–339, Springer-Verlag, 2002.
14. P. Zoeteweij. A Coordination-Based Framework for Distributed Constraint Solving. In O'Sullivan (ed.) *Recent Advances in Constraints*, LNAI 2736, Springer-Verlag, 2003.
15. P. Zoeteweij. Coordination-Based Distributed Constraint Solving in DICE. In *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC)*, Melbourne, FL, USA, pp. 360–366, ACM, 2003.