Composing constraint solvers
Zoeteweij, P.

Citation for published version (APA):
Chapter 8

A Component-Based Parallel Constraint Solver

This is the second of three chapters that demonstrate the use of OpenSolver as a software component. Here we present the design and implementation of a parallel constraint solver that is composed of several autonomous OpenSolver instances. A small amount of specialized software coordinates the cooperation of the component solvers, and in our presentation we focus on the coordination aspects of the parallel solver. Since the goal of parallel processing is to reduce the turn-around time of a computation by distributing the workload, it is important to achieve a good load balance, and to ensure that communication does not dominate computation. This is realized by a time-out mechanism, implemented in the coordination layer of the solvers. By means of experiments we investigate whether the time-out mechanism, and the component-based implementation enabled by it lead to efficient parallel solvers.

8.1 Introduction

The goal of parallel processing is to reduce the turn-around time of a computation by distributing the workload over several hardware processors. Because constraint solving is computation-intensive, it can benefit from parallel processing:

- When the best known heuristics allow that problem instances of certain dimensions (number of variables, domain sizes) can be solved within acceptable time, users will probably want to solve problem instances of larger dimensions, possibly resulting from more detailed models.

- If constraint solving is used to predict the effect of some decision made in the context of a real-world problem, while the outcome of one experiment determines the parameters of the next, then being able to solve problems faster allows that more scenarios can be explored.
The obvious way to parallelize constraint solving is to explore different parts of the search tree in parallel: even for small problems, the search tree is generally large (see for example Table 7.1 on page 162), and every node of the search tree is an ECSP in itself, and can in principle be processed independently of the nodes in other subtrees.

The efficiency of a parallel computation depends on two factors: *load balancing* and the *communication overhead*. We propose to address both factors by equipping a branch-and-propagate solver with a *time-out* mechanism. When an ECSP can be solved before the elapse of a given time-out, the solver simply produces all solutions that it has found (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 subproblems that must still be explored: the *search frontier*. These subproblems are then re-distributed among a homogeneous set of 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 time-out mechanism provides 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. The time-out mechanism also gives control over the communication / computation ratio: when communication dominates, we can increase the time-out value in order that the solvers spend more time searching in between exchanging subproblems. 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 composing 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. In the particular case of OpenSolver, the coordination-layer facilitates that the time-out mechanism is implemented without modifying the solver proper. Our presentation of the scheme in Section 8.3 uses the notion of abstract behavior types, and the Reo coordination model. These are introduced in Section 8.2. Section 8.4 details the implementation, and in Section 8.5 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 8.6, together with related work and directions for future research.
8.2. Coordination and Abstract Behavior Types

As we discussed in Section 3.3.3, coordination, as a field of study in computer science, provides a perspective on component-based software engineering. From this perspective, software systems are composed from interacting component system, whose computations overlap in time. Contrary to modules and objects, which are the units of composition in the classical software engineering paradigms of modular and object-oriented programming, an instance of a prospective software component has then at least one thread of control. For the purpose of composition, the component is a black box, and we can assume that it communicates with its environment through a set of ports.

A software system that complies with the above notion of a component can be specified conveniently by an abstract behavior type (ABT) [Arb02]. ABTs are reminiscent of abstract data types (ADTs) used in modular programming:

- ADTs hide the internals of data structures. Through ADTs, data structures are characterized by the operations that are defined on them. This is the only information that is relevant for modular composition of software.

- ABTs hide the implementation of component systems, and characterize them by their behavior. This is the only information that is relevant for the composition of software systems through exogenous coordination. Because we are dealing with concurrent systems, timing is an important aspect of the behavior of a component.

Before we can introduce ABTs we first need to recall the definition of timed data streams. This notion originates from the work of Jan Rutten on co-algebras, stream calculus, and notably a co-algebraic semantics for the Reo (see below) coordination model [AR02].

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., $a(0)$, $a(1)$, $a(2)$, ... denote the first, second, third, etc. elements of the stream $a$. Also $a^{(k)}$ denotes the stream that is obtained by removing the first $k$ values from stream $a$ (so $a(0)$ is the head of the stream, and $a^{(1)}$ 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 $(\alpha, a)$, 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 $(\alpha, a)$ 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 ABTs $R_1$ and $R_2$, let the composition $R_1 \circ R_2$ denote the relation

$$\{ \langle \langle \alpha, a \rangle, \langle \beta, b \rangle \rangle \mid \text{there exists a timed data stream } \langle \gamma, c \rangle \text{ such that } \langle \alpha, a \rangle R_1 \langle \gamma, c \rangle \text{ and } \langle \gamma, c \rangle R_2 \langle \beta, b \rangle \}.$$ 

ABTs 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 for the coordination of the components.

Reo [Arb02, ABRS04] 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 8.3.2 we use Reo connectors to specify the coordination of our component solvers.

### 8.3 Specification

#### 8.3.1 Component Solver

In this section we define an ABT for a constraint solver with the time-out mechanism. In Section 2.2.6 we defined constraint solving as the transformation of extended constraint satisfaction problems, so for formalizing the notion of a solver we need a domain, or universe, of ECSPs. Let $U$ denote the set of all ECSPs, and let $U$ denote the set of all finite subsets of $U$. Also, for $p \in U$ we define the following set

$$\text{sol}_\gamma(p) = \{ p' \in U \mid \text{p' is a } \gamma \text{ solved form of } p \}$$

Next we specify that a constraint solver transforms a problem into a set of mutually incomparable problems. Let $D$ denote the data domain $U \cup U \cup \{\tau\}$, where $\tau \notin U$ 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

$$\langle \alpha, a \rangle \ BSol \langle \beta, b \rangle \equiv a < b \land S(\alpha, \beta)$$

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

- every ECSP in $R$ is a proper subproblem of $p$,
- no ECSP in $R$ is a subproblem of another ECSP in $R$, and
- $\text{sol}_\gamma(p) = \bigcup_{r \in R} \text{sol}_\gamma(r)$, for some notion of consistency $\gamma$.

The $BSol$ ABT formalizes the notion of an incomplete constraint solver of Section 2.2.6.
8.3. Specification

8.3.1. Example. An example of a software system that complies with the BSol ABT is a UNIX process that keeps reading (character encoded) ECSPs from standard input. Some time after reading each ECSP, and before reading the next, it produces on standard output one of the following (character encoded) set of ECSPs:

- the empty set if the ECSP that was read is inconsistent, or otherwise
- a set containing the first solved form found with branch-and-propagate tree search, plus an ECSP for every element of the search frontier of the branch-and-propagate algorithm.

This can be realized by repeated application of Algorithm 8.1, which is a modified version of Algorithm 2.2 on page 25.

Figure 8.1 shows the search tree for a possible execution of this algorithm for the four queens problem (see Section 4.2). We assume chronological variable selection, enumeration value selection, and depth-first leftmost-first traversal. The nodes in the figure depict the domains of the four variables as columns of four possible values (rows), where white fields are elements of the domains, and dark fields have been removed from the domains. Vertical edges denote constraint propagation, and diagonal edges denote branching. If, by propagation or branching, the domain of a variable becomes a singleton set, an X in the remaining field marks the position of the queen on the chess board.

The two leftmost leaves of the search tree are failures (they contain columns without white fields), and the algorithm backtracks twice to find the first solution in the third leaf from the left. At this point, the search frontier still contains two nodes, and the output of our process for an ECSP corresponding to the four queens problem is a set of three ECSPs, corresponding to the following configurations, a solution and two internal nodes:

\[
\begin{array}{cccc}
X & X & \square & \\
X & X & \square & \\
\end{array}
\]

The \( Str \) (streamer) ABT specifies that a 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:

\[
\langle \alpha, a \rangle \ Str \langle \beta, b \rangle \equiv a(0) = b(0) \\
\land \beta(k) = \tau \\
\land a(0) = \{\beta(0), \ldots, \beta(k - 1)\} \\
\land \langle \alpha^{(1)}, a^{(1)} \rangle \ Str \langle \beta^{(k+1)}, b^{(k+1)} \rangle
\]

where for all \( i \in \mathbb{N} \), \( \alpha(i) \in \mathcal{U} \) and \( \beta(i) \in U \cup \{\tau\} \), and \( k \) denotes \( |\alpha(0)| \), the cardinality of the (finite) set of problems at the head of stream \( \alpha \). Now the
Chapter 8. Parallel Constraint Solving

**parameters:** function *select*, function *propagate*.

**input:** an ECSP $P := \langle \mathcal{C} : x_1 \in D_1, \ldots, x_n \in D_n ; \mathcal{T}_1, \ldots, \mathcal{T}_n : A_1, \ldots, A_n \rangle$, a domain branching function $f$, a set $R$ of domain reduction functions,

**output:** a set $S$ of sequences of domains such that for all $(D'_1, \ldots, D'_n) \in S$,

$\langle \mathcal{C}[D'_1, \ldots, D'_n] : x_1 \in D'_1, \ldots, X_n \in D'_n ; \mathcal{T}_1, \ldots, \mathcal{T}_n : A_1, \ldots, A_n \rangle$

is a subproblem of $P$. If $S$ is non-empty, at least one of these subproblems is also a $\gamma$ solved form of $P$, where $\gamma$ is the notion of consistency enforced for the constraints in $\mathcal{C}$ by *propagate* and $R$.

$$F := \{ (D_1, \ldots, D_n) \}$$

$S := \emptyset$

repeat

select $D_w \in F$

$F := F - \{ D_w \}$

$D'_w := \text{propagate}(D_w, R)$

if $\neg \text{failed}(D'_w)$

then

if $\text{final}(D'_w)$

then

$S := S \cup \{ D'_w \}$

else

$F := F \cup f(D'_w)$

end

end

until $F = \emptyset$ or $S \neq \emptyset$

$S := S \cup F$

Algorithm 8.1: A first-solution search version of Algorithm 2.2
8.3. Specification

![Diagram of first-solution search for the four queens problem]

Figure 8.1: First-solution search for the four queens problem

behavior of a constraint solver component is captured by the Sol ABT, defined as

\[ \text{Sol} = \text{BSol} \circ \text{Str} \]

Thanks to the Str ABT, we can reduce the data domain from \( U \cup \mathcal{U} \cup \{\tau\} \) to \( U \cup \{\tau\} \): systems that comply with the Sol ABT deal only with ECSPs and the token \( \tau \).

8.3.2. Example. Figure 8.2 shows the input timed data stream \((\alpha, a)\) and output timed data stream \((\beta, b)\) of a system that complies with the Sol ABT. Some time \( t_s \) after an ABT for the four queens problem appears on the input timed data stream, a solution, and two subproblems followed by the token \( \tau \) appear on the output timed data stream.

Unlike our example solver, existing complete constraint solvers do not usually produce subproblems other than solutions. The search frontier is inaccessible, and 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. Also, if we model such solvers using the Sol ABT, there is typically no upper bound on the time that elapses before solutions start to appear on the output timed data stream.

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
Figure 8.2: Input stream and output stream of a solver that complies with the BSol ABT

subproblem that must still be explored. For $t \in \mathbb{R}^+$, the Sol_AB T specifies that there is an upper bound on the time needed for the solver to produce results.

$$\langle \alpha, a \rangle \text{ BSol} \langle \beta, b \rangle \equiv \langle \alpha, a \rangle \text{ BSol} \langle \beta, b \rangle \
\land \forall i \in \mathbb{N} \cdot b(i) - a(i) < t + t_c$$

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

where $t_c \in \mathbb{R}^+$ is some extra time that allows the solver to finish what it is doing, after the time-out $t$ has elapsed.

The Sol_t behavior can be realized trivially by removing the loop from Algorithm 8.1. For an input ECSP, the resulting solver then performs a single round of constraint propagation and splitting. However, in order to limit the amount of communication, for our application we want to make the solvers perform as much work as possible, within the given time-out period.

We can capture this additional requirement by adding the following condition to the BSol_t ABT:

$$\forall i \in \mathbb{N} \cdot b(i) - a(i) \geq t \lor \beta(i) = \text{sol}_t(a(i))$$

This ensures that unless the search space has been explored exhaustively, the output is produced between time $t$ and $t + t_c$.

Implementing this exact behavior is not straightforward, but an approximation of it can be realized by modifying the loop of Algorithm 8.1:

repeat
  ...
  until $F = \emptyset$ or $S \neq \emptyset$

as follows.

$t_0 := \text{clock}()$
repeat
  ...
  until $F = \emptyset$ or $\text{clock}() - t_0 \geq t$
In this modified code $t$ is a new parameter that specifies the time-out value, and `clock` is a function that returns the current time, or *wall time*. Such a function is typically made available by the operating system. The resulting solver is an approximation of the required behavior in the sense that the operations inside the loop cannot be interrupted. Especially constraint propagation may take longer than the allowed overrun time $t_c$. This is not a problem in practice.

### 8.3.2 Parallel Solver

Figure 8.3 shows a channel-based design for a (3-way) parallel solver. All channels in this design are synchronous\(^1\): read and write operations block until a matching operation is performed on the opposite channel end. The “resistors” depict Reo filters: synchronous channels that forward 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 8.3, all output of the solvers is replicated onto two filters. Channel bc filters out solutions. Its pattern $(p)$ is

$$\text{Filter}(\{ p \in P \mid p \in sol_\gamma(p)\}).$$

The channel from b to T discards all solutions. Its pattern $(q)$ is

$$\text{Filter}(\{ p \in P \mid p \notin sol_\gamma(p) \} \cup \{\tau\}).$$

The ABTs of the channels are specified in [Arb02].

Apart from the channels and the three load-balancing solvers $Sol_i$, 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. Because we focus on the component solvers instead of on the coordinating framework, we do not give full ABTs for the other elements of Figure 8.3, but only an intuitive description.

\(^1\)The synchronicity of the communication is not an important aspect of the design.
The special purpose connector $T$ implements **termination detection**. Initially, it reads a problem from its left-hand side input port. All subproblems 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 returns to its initial state, and accepts a new problem from its left-hand side input port. It should be noted that termination detection for the parallel solver is much easier than termination detection for a distributed application in general. The latter case is discussed in Section 9.2.3.

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 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 [ABRS04].

The **Store** is a channel-like connector that is specific to this application. It buffers incoming problems, and examines them 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.

### 8.4 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.

#### 8.4.1 Component Solver

A special coordination layer plug-in **StreamingIO** has been developed that configures OpenSolver as a load-balancing solver, as specified in Section 8.3. When it is equipped with this plug-in, an OpenSolver instance keeps reading configuration specifications from its standard input. These specifications are sequences of
8.4. Implementation

ASCII characters, in the language of Figure 3.2 (see Program 4.3 on page 77 for an example). The individual configuration specifications are delimited by brackets, and configure OpenSolver for solving a particular ECSP.

When a solver configuration has been read from standard input, the coordination layer plug-in instructs the solver to parse it, and starts the search for solutions. This coordinates the solver to perform a regular branch-and-propagate search, as explained at the end of Section 3.3.1. 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 flush and clear WDB commands of Section 3.3.2.

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 DRFs. These strings are then passed to the coordination layer. Normally this mechanism is used to produce the solved forms of an ECSP, but because we do not 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 solver configurations 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 directly be fed into another solver as a stream of solver configurations.

The component solvers are configured to perform a depth-first traversal of the search tree, but through an adapter, the branching operators are modified to annotate the nodes with their level in the search tree. These annotations appear in the solver configurations that are forwarded through the network, and can be interpreted by the process that implements the Store of Section 8.3.2 to impose a high-level traversal strategy on top of the depth-first traversal of the solvers.

As we discussed in Section 4.2, OpenSolver is based on copying, so the search frontier is maintained explicitly. This is a great convenience for publishing the search frontier, but we are convinced that our method extends to solvers that use trail 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.

8.4.2 Parallel Solver

Depending on the complexity of the interaction, it may make sense to use a dedicated coordination language to orchestrate the interaction of the cooperating entities in a concurrent system. For example if the population of processes is highly dynamic, the Manifold coordination language [Arb96] may be a logical choice (see also Section 9.2.1). In this case, we implemented the coordination protocol of Section 8.3.2 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 solver configuration. 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 8.4 illustrates this software architecture. In total, for an $n$-way parallel run, $2n+1$ user processes are running on $n$ processors.

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 8.3.2 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. The Manifold coordination language fully supports the design of Section 8.3.2.
8.5 Experiments

The parallel solver was tested on three combinatorial problems:

**Queens** An instance of the *n*-queens problem, as described in Section 4.2. Program 4.3 shows a solver configuration for *n* = 4. Instead of the variable based scheduler we used the default, operator-based scheduler. The results reported here are for *n* = 15, for which there are 2,279,184 solutions.

**Sat** An instance of the propositional satisfiability problem, described in Section 4.4. For these experiments we use the benchmark formula par16-2-c from the DIMACS test set\(^2\). 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, also from a 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 DRFF plug-in of Section 4.2.

In all cases, we used a fail-first variable selection strategy, selecting a variable with the smallest remaining number of alternative values. As a second criterion for **Coloring**, variables are ordered according to the degree of their corresponding nodes of the graph. In order to generate a large number of subproblems, we used an enumeration value selection strategy (see Figure 4.2 on page 70). The component solvers perform a depth-first traversal, but using the level annotation of the configurations 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 nodes. These are least likely to complete before 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.

The results reported below are for an all-solution search, and solutions are only counted, not stored or communicated. An all-solution search avoids the effect known as the *speedup anomaly*, which entails that for a non-exhaustive search, part of the speedup is due to the different traversal of the search space. For example, consider that the search space is split into two subtrees, and that the root node of the second subtree happens to reduce to a solution. Parallel search on these two subtrees would find the solution almost immediately, resulting in a *super-linear speedup* over the sequential case where the other subtree

\(^2\)available at ftp://dimacs.rutgers.edu/pub/challenge
is processed first. Because we are interested in evaluating the efficiency of our proposed parallelization of tree search, we tried to avoid demonstrating the speedup anomaly.

Table 8.1 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 a further indication that our solver is a realistic implementation, depending on the search strategy, the standard example for 15-queens in ECLIPSE 5.5 [WNS97, CHS+03] completes in 900 - 1500 sec. on the same hardware. The speedup figures (sequential runtime divided by parallel runtime) are shown in Figure 8.5. 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. These values were found to give good results in preparatory experiments, but performance did not seem overly sensitive to the actual time-out used. The master process always runs on the same node as one of the component solvers and its slave process, and competes with these processes for CPU time.

As can be seen from Figure 8.5, 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 be explained by the different sizes of the problem representations, and their associated
8.5. Experiments

<table>
<thead>
<tr>
<th></th>
<th>Seq</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queens</td>
<td>734.16</td>
<td>760.79</td>
<td>380.85</td>
<td>253.41</td>
<td>190.04</td>
</tr>
<tr>
<td>eff.</td>
<td>0.96</td>
<td>0.96</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
</tr>
<tr>
<td>Sat</td>
<td>1541.12</td>
<td>1842.5</td>
<td>931.26</td>
<td>619.65</td>
<td>466.08</td>
</tr>
<tr>
<td>eff.</td>
<td>0.84</td>
<td>0.83</td>
<td>0.83</td>
<td>0.83</td>
<td>0.83</td>
</tr>
<tr>
<td>Coloring</td>
<td>419.29</td>
<td>475.92</td>
<td>236.50</td>
<td>156.47</td>
<td>117.70</td>
</tr>
<tr>
<td>eff.</td>
<td>0.88</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queens</td>
<td>152.01</td>
<td>126.67</td>
<td>95.18</td>
<td>63.32</td>
<td>47.86</td>
</tr>
<tr>
<td>eff.</td>
<td>0.97</td>
<td>0.97</td>
<td>0.96</td>
<td>0.97</td>
<td>0.96</td>
</tr>
<tr>
<td>Sat</td>
<td>378.14</td>
<td>313.91</td>
<td>240.91</td>
<td>171.43</td>
<td>140.02</td>
</tr>
<tr>
<td>eff.</td>
<td>0.82</td>
<td>0.82</td>
<td>0.80</td>
<td>0.75</td>
<td>0.69</td>
</tr>
<tr>
<td>Coloring</td>
<td>94.31</td>
<td>78.11</td>
<td>58.23</td>
<td>38.92</td>
<td>30.56</td>
</tr>
<tr>
<td>eff.</td>
<td>0.89</td>
<td>0.89</td>
<td>0.90</td>
<td>0.90</td>
<td>0.86</td>
</tr>
</tbody>
</table>

Table 8.1: Elapsed times (sec.) and parallel efficiency

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.
8.6 Discussion

Perspectives and Limitations

As an alternative to implementing the time-out mechanism in the component solvers, we could move this mechanism into the software that coordinates them. 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 either case 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. Also saving subproblems to disk may increase the applicability of limited discrepancy search in a copying-based solver (see Section 4.1.2), especially if the I/O can be performed in the background, and does not imply busy waiting.

Another possibility is to implement in-search transformations of CSPs outside the core solver. In OpenSolver, such transformations are currently limited to deactivating reduction operators that have become redundant. For other transformation techniques, such as adding redundant constraints, or symbolic rewriting of arithmetic constraints, it is not immediately clear how to incorporate these in the branch-and-propagate search in a uniform way. Moreover, extending OpenSolver to accommodate specialized transformation techniques may make it less efficient in those cases where these techniques are not needed. By applying transformations outside the branch-and-propagate search, we risk that constraint propagation is weaker than necessary, but because of the time-out mechanism, this inefficiency will not last long, and the overall effect may be a good compromise between ease of implementation and efficiency.

Our current implementation is not suited for optimization, because new bounds for a criterion variable are not communicated between solvers. When a new bound is discovered, many of the subproblems in the Store may never be able to improve on this bound, but they have to be processed nonetheless. What is worse, the new bound remains local to the subproblems of the ECSP in which it was found. After flushing its search frontier, a solver returns to its initial state, and forgets the bound, and only the solvers that pick up one of the generated subproblems will temporarily be able to use it. This can be remedied by adding two processes to the network of Figure 8.3. One process inspects the value of the criterion variable for outgoing solutions, and sends this value to the other process, which adds a reduction operator for enforcing the current best bound to any subproblem
leaving the Store. This is illustrated in Figure 8.6.

Branch-and-bound optimization was studied from a coordination point of view in [Sta02], 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 do not expect that our component-based solution performs worse than other parallel implementations of branch-and-bound, but this should be verified by further experiments.

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. As an example, we have seen in Section 4.4 that specialized solvers for the SAT problem rely on so-called learning search algorithms, which 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 the complementing backjumping techniques.

**Related Work**

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 [MS94, Per99, Sch00, Ham05]. 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. It may well be possible, though, to use a heuristic for tuning the time-out automatically during the computation. For example, the Store of Figure 8.3 could increase the time-out value while enough subproblems are available to keep all solver busy for some time.

In [HKS01] a shared-memory scheme is described where first the original CSP is split by assigning values to variables in a generate-and-test phase, until a large set of subproblems are 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.

The approach of Disolver [Ham05] is unique in the sense that load balancing and bound sharing (in optimization) can be controlled through setting some pre-defined logical variables. In addition, several properties of the search process for which we use annotations are reflected in the domains of other pre-defined variables. Regular constraints can be now be used to activate or deactivate load balancing and bound sharing depending on properties of the search process. This way, adaptive cooperations between the parallel running solvers can be specified through constraints.

For all alternatives discussed here, a comparison of reported efficiency results is difficult, because the hardware platforms and software environments, and the benchmark problems used in each case are quite diverse. For example, the results in [Per99] for ILOG Solver apply to job-shop scheduling problems. Because these are optimization problems, the experiments are very sensitive to the traversal order, leading to speedup anomalies: for various experiments, the observed speedup ranges from 1.23 to 3.92 on two processors, and from 2.4 to 28.95 on four processors. The author mentions that running the parallel solver with one processor incurs an overhead of 2-3%, which could be comparable to the column for 1 processor in Table 8.1. This result is for a shared memory system, which may explain the low overhead compared to our approach.

Some of the results in the other references above do not suffer from the speedup anomaly, and just to indicate that despite its straightforward load balancing scheme our parallel solver is fairly efficient, Table 8.2 presents a comparison yet. For each system, the best and worst speedups among the presented set of experiments are listed. From [Sch00] we did not consider the optimization problems because of the speedup anomaly. For the same reason, from [HKS01] we cite only the results for inconsistent problems. Furthermore it should be noted that the speedup figures of [MS94] were not obtained by comparison with the best possible sequential run. They apply to the parallelized system, which is less efficient. Also [HKS01] concerns a shared memory implementation. The quoted results are for their best (dynamic) partitioning strategy only. The results reported for ECLIPS in [MS94] and for the Mozart implementation of Oz in [Sch00] are for distributed
8.7. Conclusions

<table>
<thead>
<tr>
<th>CPUs</th>
<th>OpenSolver [MS94]</th>
<th>[Sch00]</th>
<th>[HKS01]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.65–1.93</td>
<td>1.78–1.99</td>
<td>1.74–1.85</td>
</tr>
<tr>
<td>3</td>
<td>2.49–2.9</td>
<td>2.47–2.63</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4.08–4.83</td>
<td>3.12–3.51</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4.91–5.8</td>
<td>3.17–3.81</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6.4–7.71</td>
<td>4.60–7.56</td>
<td>6.32–7.44</td>
</tr>
<tr>
<td>12</td>
<td>8.99–11.59</td>
<td>5.37–10.79</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>11.01–15.34</td>
<td>9.08–13.95</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2: Comparison of speedup figures

memory (networked) systems, like ours. The last two systems both use recomputation to regenerate nodes of the search tree that have been transferred from one machine to another. For Disolver, we did not find any results that do not suffer from the speedup anomaly.

8.7 Conclusions

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 CPUs, 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.
- Separating computation and coordination, i.e., adding a protocol instead of implementing a parallel algorithm is a viable approach.

We also described how to implement parallel optimization. We do not expect that our component-based solution performs worse than other parallel implementations of branch-and-bound, but this should be verified by further experiments.