Design and Implementation of a Parallel Constraint Satisfaction Algorithm

Marco Platzner and Bernhard Rinner

Institute for Technical Informatics, Graz University of Technology, AUSTRIA

Abstract

Constraint satisfaction techniques are applied in areas like design, diagnosis, truth maintenance and scheduling. This paper describes the design and implementation of a parallel constraint satisfaction algorithm for *qualitative simulation*. In qualitative simulation, constraint satisfaction problems (CSPs) must be solved very often which in turn dominates the runtime of the most prominent qualitative simulator QSim.

The presented parallel algorithm is based on partitioning the search space of a CSP into independent subspaces. These subspaces are then searched for solutions in parallel with a backtracking algorithm. Finally, the solutions of the subspaces are used to generate the overall result of the CSP.

We introduce a new partitioning method, called *variable-based partitioning (VBP)*. This method discards many subspaces from further processing by revealing local inconsistencies in the subspaces. Speedup limits for the parallel algorithm are analytically derived and proved by experimental results. The parallel CSP algorithm is implemented on a MIMD multiprocessor system and evaluated using benchmark-CSPs derived from the qualitative simulator QSim.

keywords: parallel constraint satisfaction,
search space partitioning,
qualitative simulator QSim

1 Introduction

Constraint satisfaction is a well-known term for a variety of techniques used in artificial intelligence (AI) and related disciplines. Constraint satisfaction techniques are applied in areas like design, diagnosis, truth maintenance and scheduling [4]. One of the application areas is *qualitative simulation* [7] which is a rather new and challenging simulation paradigm. In qualitative simulation, physical systems are modeled on a higher level of abstraction than in other simulation paradigms, like in continuous simulation. Qualitative simulation is mainly used in applications where a detailed description of a physical system is not required or not known; major application areas are design, monitoring and fault diagnosis. QSIM is the most prominent algorithm for qualitative simulation and was developed by Kuipers [8]. One drawback of current QSIM implementations is poor execution speed. In our research project [15] [16] [17], a special-purpose computer architecture has been developed to improve the performance of QSIM. This is achieved by parallelizing the constraint satisfaction algorithm of QSIM and mapping the parallel algorithm onto a multiprocessor system. This paper describes the design and implementation of this parallel constraint satisfaction algorithm.

A *constraint-satisfaction problem (CSP)* is defined as a triple $\langle V, D, C \rangle$ consisting of

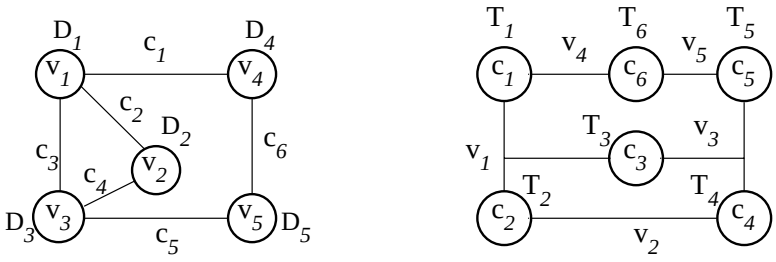
- a set $V = \{v_1, \ldots, v_l\}$ of variables,
- a set $D = \{D_1, \ldots, D_l\}$ of domains, such that each D_i is a set of values for the variable v_i and
- a set $C = \{c_1, \ldots, c_m\}$ of constraint relations, where each c_j refers to some subset of the variables V. The constraint relations determine subsets of the Cartesian product of the domains of the variables involved. Every v_i in V appears in some c_j in C.

The assignment of a unique domain value to each variable of some subset of variables

is called *instantiation*. An instantiation is said to be legal if it does not violate any of the relevant constraints. A legal instantiation of all variables V is called a *solution* of the CSP. Therefore, a solution is one element in the search space $S = D_1 \times \cdots \times D_l$ of the CSP.

CSPs are often represented as constraint networks. In these networks, the nodes correspond to the variables, and the edges correspond to the constraints between the variables. Figure 1(a) shows the constraint network of a simple CSP consisting of 5 variables and 6 constraints. Each constraint relation refers to 2 variables. Hence, the arity of these constraints is two. Constraints which share a variable are *adjacent*; these constraints are *attached* to the shared variable. In Figure 1(b), the dual constraint network of the CSP is shown. In this representation, the nodes correspond to the constraints, and the edges correspond to the shared variables. The domain of node i in the dual representation is the Cartesian product of the domains of the attached variables. This set is called *set of tuples* T_i. Hence, the overall tuple set of the CSP is $T = \{T_1, \ldots, T_m\}$. The domain of node i can be reduced by applying the constraint relation C_i to the tuple set T_i; a CSP with reduced tuple sets is said to be *node-consistent*. An instantiation of tuples of adjacent nodes is only valid, if the value of the shared variable is the same in all tuples. Table 1 summarizes the relationship between a constraint network and its dual representation.

In general, finding a solution of a CSP is *NP-complete* [12]. CSPs are often solved by backtracking algorithms which explore the search space of the CSP by a depth-first search. Many improvements to simple backtracking algorithms have been developed to solve CSPs more efficiently. These improvements range from preprocessing steps, like arc- and path-consistency algorithms [13] [14], to advanced search techniques [18]. In QSim, the basic algorithm to solve the CSP is *simple backtracking* [18]. Increased performance is achieved by an arc-consistency algorithm that reduces the search space and by a heuristic



(a) constraint network (b) dual constraint network

Figure 1: A CSP is graphically represented as constraint network (a) consisting of 5 nodes $v_1, \ldots, v_5$ and 6 constraint relations $c_1, \ldots, c_6$. Each node is assigned with the domain of the variable $D_1, \ldots, D_5$. In the dual constraint network (b), the nodes correspond to the constraint relations and the edges to the variables, respectively. In this representation, the nodes are assigned with the sets of tuples $T_1, \ldots, T_6$.

	constraint network	*dual representation*
nodes	V	C
edges	C	V
domains	D	T
search space	$D_1 \times \cdots \times D_l$	$T_1 \times \cdots \times T_m$

Table 1: Relationship between a constraint network and its dual representation.

ordering of the constraint relations for the backtracking step. The CSP in QSIM has some specific properties which make it different from many other CSPs. These properties influence the parallel implementation:

- The backtracking algorithm in QSIM solves the CSP by finding the solutions of the dual constraint network. Therefore, the overall search space of the QSIM CSP is given by $S = T_1 \times \cdots \times T_m$.
- Constraint satisfaction in QSIM requires *all* solutions of the CSP.
- In QSIM, the arity of the constraint relations ranges from 1 to 3, and the domain of the variables is limited by 4. Therefore, the maximum number of tuples per constraint is given by 4, 16 and 64, respectively.

The remaining part of this paper is organized into the following sections. Section 2 reviews closely related work in the area of parallel constraint satisfaction. Section 3 introduces a partitioning algorithm for QSIM CSPs using four different heuristics. Section 4 presents an analytical estimation of the speedup limits and a comparison of the different partitioning heuristics. In Section 5, the parallel implementation and the experimental results are presented. A discussion of the results concludes the paper.

2 Related Work

2.1 Classification of Parallel CSP Algorithms

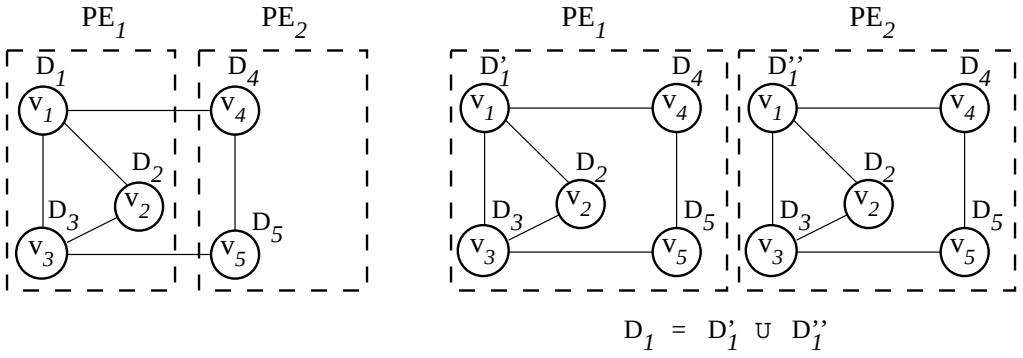
Luo, Hendry and Buchanan [11] have classified the most common parallel CSP algorithms as *distributed-agent-based (DAB)*, *parallel-agent-based (PAB)* and *function-agent-based (FAB)*. In all these strategies, the CSP is split into a number of subproblems that are distributed over a number of processors. Important features of these strategies can be summarized as follows:

DAB. In the *distributed-agent-based* strategy, the CSP is split by assigning one or more variables with their complete domains to the individual processors. All processors work together in one shared search space. During the search, the processors have to communicate to satisfy the constraint relations, i.e., to resolve conflicts between variables located in different processors. The control mechanisms to resolve these conflicts can be centralized or decentralized. A serious drawback of DAB is that the control mechanism usually introduces a lot of communication overhead.

PAB. In the *parallel-agent-based* strategy, the CSP is split by partitioning the domains of the variables. Each processor solves a part of the complete search space that involves all variables and is independent of other partial search spaces. Therefore, each processor solves a unique CSP, and no conflicts between processors must be resolved. PAB can directly use any sequential CSP algorithm, requires little communication overhead and is amenable to established global heuristic search strategies.

FAB. The *function-agent-based* strategy exploits the control-parallelism of constraint satisfaction. Luo, Hendry and Buchanan describe this strategy based on a parallel implementation of a *forward-checking* algorithm. In this implementation, the complex forward-check operation is partitioned into several processing elements by spawning individual subprocesses. These subprocesses manipulate the data, i.e., the variables' values, of the parent process.

Figure 2 presents the two most important parallelization strategies DAB and PAB. The inherent communication of the DAB strategy (a) is visualized by the edges (constraints) connecting variables assigned to different processors. The independent subproblems of the PAB strategy are shown in (b). Due to the reduced domain sizes of some variables, each of the individual subspaces is smaller than the original search space. As described by [11], a FAB strategy is only suitable where complex (forward-) check operations can be



(a) Distributed-agent-based (b) Parallel-agent-based

Figure 2: Two important parallelization strategies for CSPs. Two processing elements PE_1 and PE_2 are used to solve the CSP. In the DAB strategy (a), the CSP is partitioned into two subproblems based on the variables. In the PAB strategy (b), independent subspaces of the overall search space of the CSP are generated based on the domains of the variables.

partitioned. The checks in QSIM's backtracking algorithm are simple comparisons. In the following section, the PAB strategy is investigated in more detail because it is well suited for parallel constraint satisfaction where all solutions are required and the scalability of the parallel implementation is important [11].

2.2 Parallel Algorithms Using Search Space Partitioning

A lot of results have been published in the area of parallel constraint satisfaction. Some of the papers deal with the parallelization of the preprocessing steps, i.e., arc-consistency algorithms. The approaches in these papers range from a massively parallel implementation based on VLSI [3] to more theoretical ones [5] [23]. In the remainder of this section, closely related work in the area of parallel algorithms using search space partitioning by PAB strategies is described in some detail.

In Burg [1], the search space is divided among a set of processors each executing a sequential forward checking (FC) algorithm [18]. The search space is partitioned into the same number of subspaces as processors are available. The number of processors of the used multiprocessor system is a power of two. Due to static load balancing, no communication between the processors is required. An extension which utilizes dynamic load balancing is given in [2]. However, dynamic load balancing introduces communication among the processors.

In Lin and Yang [9] [10], a forward checking algorithm is executed on each processor. The partitioning strategy differs from Burg's in that the search space is divided into d partitions, where d is the domain size of the first variable of the CSP. Only one variable is considered for the partitioning. The subspaces are statically mapped onto the processing elements. To achieve a balanced mapping, a simple probabilistic analysis is used to estimate the amount of search required in each subspace. The parallel algorithm is evaluated by simulation on a single-processor system.

Rao and Kumar [19] have evaluated the efficiency of parallel backtracking algorithms. They have presented analytical models and experimental results on the average case behavior of two parallel backtracking algorithms — simple backtracking and ordered backtracking. The difference between these two algorithms is that ordered backtracking uses heuristics for ordering the variables and pruning nodes of the search space. Dynamic load balancing is applied to reduce processor idle times.

The related work differs from the work presented in this paper in (i) the partitioning method and (ii) the number of generated subspaces. In the related work, the number of subspaces is restricted — either by the number of processing elements which has to be a power of two or by the domain size of the first variable. Restricting the number of subspaces and processing elements to a power of two is impractical for our work. This is because QSIM's CSPs do not naturally split up into a number of subspaces that is a power of two and the resulting multiprocessor would only be scalable in very coarse steps. Furthermore, as the domain size of a variable in qualitative simulation is limited by 4, the partitioning method of Lin and Yang would lead to a multiprocessor with at most 4 processing elements. The results of Rao and Kumar also do not directly apply to constraint satisfaction in qualitative simulation. Their evaluation is based on CSPs which require only *one* solution, i.e., the CSP algorithm terminates when a solution is found in any subspace. On average, a solution is found earlier in a subspace with many solutions than in a subspace with few solutions. When the solutions are nonuniformly distributed among the subspaces, the average speedup is superlinear. However, qualitative simulation requires *all* solutions of the CSP, i.e., all subspaces must be completely processed. If no subspace can be discarded a priori, at most a linear speedup can be achieved independent of the distribution of solutions in the subspaces.

3 The Parallel Constraint Satisfaction Algorithm

This section presents the design of the parallel constraint satisfaction algorithm for qualitative simulation. First, we give a general outline of the algorithm; then we present our new partitioning method. Issues concerning scheduling and the workload distribution are discussed, and finally, four different partitioning heuristics are presented.

3.1 Algorithm Outline

In our work, a PAB strategy consisting of the following three consecutive steps is used:

1. Partitioning the overall search space into independent subproblems.
2. Solving the subproblems in parallel.
3. Merging the results of the subproblems to the overall result.

As already stated in the previous section, a subproblem in the PAB strategy can be solved by any sequential CSP algorithm. To allow a fair evaluation of our parallel implementation, the same simple backtracking algorithm as in the original QSIM implementation is used. The overall result of the CSP is the union of all results of the subproblems. The essential part for achieving a high performance of the parallel implementation is therefore step 1, the partitioning of the search space.

In QSIM, the backtracking algorithm finds the solutions of the CSP based on the dual constraint network. Therefore, in our PAB algorithm the tuple sets must be split for the partitioning. A subproblem is defined as

$$P_i = T_{1i} \times \cdots \times T_{mi}, \tag{1}$$

where T_{ki} represents a subset of T_k. For a valid partitioning, the union of all p subproblems must be the overall search space S of the CSP. Hence, the following equation must hold:

$$\bigcup_{i=1}^{p} P_i = T_1 \times \cdots \times T_m = S \tag{2}$$

3.2 Variable-Based Partitioning (VBP)

In general, the tuple sets can be divided into subsets in an arbitrary manner. However, our method for partitioning the tuple sets takes into account the dependencies between adjacent constraints. The tuple sets of adjacent constraints are connected by the domain of the constraints' shared variables. A partitioning of the domain of a shared variable automatically induces a partitioning of the tuple sets of all attached constraints. We call this partitioning method *variable-based partitioning (VBP)*; it is depicted in Figure 3. By dividing the domain of variable v_a into k subdomains, the tuple sets of all attached constraints are divided into k subsets. In this advantageous partitioning, each subproblem of the CSP is characterized by the considered subdomains of the variables. All individual tuple subsets of one subproblem consist only of tuples using exactly those variable values given by the characterizing subdomain. The advantage of VBP compared to partitioning the tuple sets in an arbitrary manner is that many subproblems can be discarded from further processing. VBP avoids the generation of subproblems with tuple subsets using mutually exclusive variable values. These subproblems violate local consistency conditions, i.e., *arc-consistency* [13]. There cannot be any solution in these subproblems. The number of subproblems which must be searched with VBP is given by the number of subdomains. If k subdomains are generated for a variable with q attached constraints, only k subproblems must be searched for solutions. In contrast, if each tuple set of q constraints is partitioned into k subsets in an arbitrary manner, no subproblem can be discarded a priori. In this case, q^k subproblems must be processed. To generate more subproblems than the domain size of one variable, VBP is extended to other variables of the constraint network.

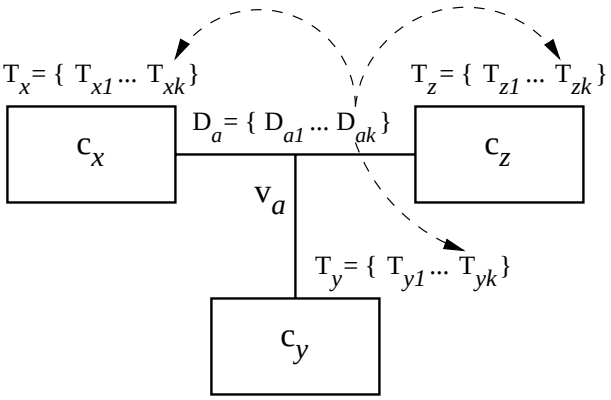


Figure 3: Variable-based partitioning. The partitioning of the domain $D_a = \{D_{a1}, \ldots, D_{ak}\}$ of the shared variable v_a induces a partitioning of the tuple sets of all attached constraints c_x, c_y and c_z.

The VBP method is described in more detail based on a pseudo code representation. VBP consists of two procedures. The first procedure `vbp` recursively divides the complete search space into candidate subspaces. The second procedure `generate-subproblem` checks each candidate subspace for local consistency and stores the corresponding tuple sets for each consistent subspace to generate a subproblem.

VBP uses the following global data structures: the variable `maxsub` stores the number of requested subproblems, the variable `gensub` stores the number of generated subproblems and the array `actval` stores the subdomains of all processed variables. The initialization of these data structures and the first call of the procedure `vbp` is shown in Figure 4. The pseudo code for the procedures `vbp` and `generate-subproblem` is presented in Figure 5 and Figure 6. The recursive procedure `vbp` uses the actual variable `var` and the number of already generated candidate subspaces `nsub` as parameters. The terminal condition for the recursion is the comparison in line 3. If a further partitioning is required, `nsub` is increased (line 4) and a subdomain is generated for each value of the actual variable `var`. In line 9, `vbp` is called recursively with the next variable and `nsub` as parameters. If a sufficient number of candidate subspaces has been generated or no further partitioning is possible, each candidate subspace is checked for local consistency in line 12, and the number of generated subproblems is updated (line 13).

In lines 7–12 of `generate-subproblem`, the tuple set of each constraint is filtered using the array `actval`. For this filtering, the function `tuple-val(tup,var)` is used which returns the value of the variable `var` in the tuple `tup`. In line 15, each consistent tuple is stored to generate the subproblem. If no tuple of a constraint survives, filtering is aborted, the subproblem is discarded and FALSE is returned (lines 18–21).

There may be some cases in which the VBP method generates less than `maxsub` subproblems. This is caused for two reasons. First, `vbp` generates only `maxsub` candidate subspaces, if the product of the domain sizes of all processed variables equals `maxsub`.

Second, the procedure `generate-subproblem` discards candidate subspaces that violate local consistency conditions.

3.2.1 Example

The VBP method is demonstrated in the following example. The variables and constraints are given by the constraint network of Figure 1 ($V = \{v_1, \ldots, v_5\}, C = \{c_1, \ldots, c_6\}$). The tuple sets are defined as follows:

$$\begin{aligned}
T_1 &= \{a_1d_1,\ a_2d_1,\ a_3d_1\} \\
T_2 &= \{a_1b_1,\ a_1b_2,\ a_2b_1,\ a_2b_2\} \\
T_3 &= \{a_1c_1,\ a_1c_2,\ a_2c_1,\ a_2c_2\} \\
T_4 &= \{b_1c_1,\ b_1c_2\} \\
T_5 &= \{c_1e_1,\ c_3e_1,\ c_1e_2,\ c_3e_2\} \\
T_6 &= \{d_1e_1,\ d_2e_1,\ d_1e_2,\ d_2e_2\}
\end{aligned}$$

The CSP results in 4 solutions:

$$\begin{aligned}
L_1 &= \{a_1b_1c_1d_1e_1\} \\
L_2 &= \{a_2b_1c_1d_1e_1\} \\
L_3 &= \{a_1b_1c_1d_1e_2\} \\
L_4 &= \{a_2b_1c_1d_1e_2\}
\end{aligned}$$

In this example, the complete search space of the CSP, $(T_1 \times \ldots \times T_6)$, is partitioned by VBP into at most 6 subproblems (`maxsub=6`); the order of the variables is determined by their indices, i.e., v_1 to v_5. Thus, the procedure `vbp` is initially called with the parameters v_1 and 1.

Because variable v_1 has 3 values, `vbp` is called recursively three times with the parameters v_2 and 3. Recursion continues until variable v_3. At this point, the number of generated candidate subspaces equals `maxsub`, and the procedure `generate-subproblem` starts to generate the partitioned tuple sets. Overall, this procedure is called 6 times.

```
1     maxsub ← requested number of subproblems
2     gensub ← 0
3     for all constraints con do
4      for all variables var do
5       actval[con,var] ← NIL
6      endfor
7     endfor
8     vbp(first-var(), 1)
```

Figure 4: Pseudo code for the initialization of the data structures and first call of the procedure `vbp`.

```
1     procedure vbp(var, nsub)
2     begin
3      if ((var ≠ NIL) ∧ (nsub < maxsub)) then
4       nsub ← nsub*(domain size of var)
5       for all values val of variable var do
6        for all attached constraints con of var do
7         actval[con,var] ← val
8        endfor
9        vbp(next-var(var), nsub)
10      endfor
11     else
12      if (generate-subproblem() = TRUE) then
13       gensub ← gensub + 1
14      endif
15     endif
16    end
```

Figure 5: Pseudo code for the procedure `vbp`.

candidate subspace	*subdomains*		*tuple sets*				*solutions*
i	D_{1i}	D_{2i}	T_{1i}	T_{2i}	T_{3i}	T_{4i}	L_j
1	a_1	b_1	a_1d_1	a_1b_1	$a_1c_1,\ a_1c_2$	b_1c_1	L_1, L_3
2	a_1	b_2	a_1d_1	a_1b_2	$a_1c_1,\ a_1c_2$	$\emptyset$	
3	a_2	b_1	a_2d_1	a_2b_1	$a_2c_1,\ a_2c_2$	b_1c_1	L_2, L_4
4	a_2	b_2	a_2d_1	a_2b_1	$a_2c_1,\ a_2c_2$	$\emptyset$	
5	a_3	b_1	a_3d_1	$\emptyset$			
6	a_3	b_2	a_3d_1	$\emptyset$			

Table 2: Candidate subspaces, subdomains of the partitioned variables, tuple sets and solutions for Example 3.2.1. Dividing the domains of variables v_1 and v_2 into subdomains causes a partitioning of the tuple sets of all attached constraints. The tuple sets T_5 and T_6 remain unchanged.

```
1    procedure generate-subproblem()
2    begin
3      for all constraints con do
4        tuple-found ← FALSE
5        for all tuples tup of con do
6          tuple-ok ← TRUE
7          for all variables var of tup do
8            aval ← actval[con,var]
9            if ((aval ≠ NIL) ∧ (aval ≠ tuple-val(tup,var)))
10             tuple-ok ← FALSE
11           endif
12         endfor
13         if (tuple-ok = TRUE) then
14           tuple-found ← TRUE
15           save tuple for subproblem
16         endif
17       endfor
18       if (tuple-found = FALSE) then
19         discard subproblem
20         return FALSE
21       endif
22     endfor
23     return TRUE
24   end
```

Figure 6: Pseudo code for the procedure `generate-subproblem`.

However, `generate-subproblem` returns only 2 subproblems for further processing; 4 candidate subspaces are discarded. Table 2 summarizes the subdomains and partitioned tuple sets for all 6 candidate subspaces.

3.3 Scheduling and Workload Distribution

For a generated number of subproblems, the overall parallel execution time is mainly influenced by the schedule of the individual subproblems. The execution times of the subproblems cannot be exactly determined in advance. Only a worst-case estimation can be given, but this estimation normally differs from the the actual execution time by orders of magnitude. Therefore, the scheduling algorithm cannot utilize execution times to generate the schedule. In our parallel implementation, *task attraction scheduling* is used. Whenever a processing element is idle, the next subproblem is assigned to this processing element and it runs there without interruption. An unbalanced workload distribution occurs, when subproblems with extraordinary long execution times are scheduled at the end. Our scheduling strategy does not allow the improvement of an unbalanced workload by an additional partitioning and redistribution of subproblems already running on processing elements. Furthermore, due to redundancies in the independent subproblems, the total workload of all subproblems can be greater than the workload of the unpartitioned search space. These considerations lead to two important requirements for the partitioning method. An efficient partitioning method keeps the total workload small and generates equally sized subproblems.

The choice of VBP is a first step in keeping the total workload small, because VBP avoids the generation of subproblems that obviously do not lead to solutions. Due to the rejection of these subproblems, the total workload of all subproblems can even be smaller than the workload of the unpartitioned search space. Another point to be investigated is the order in which the variables are processed by VBP.

3.4 Partitioning Heuristics

The order in which the variables are processed influences the number and the workload of the generated subproblems and, in turn, the achievable speedup of the parallel implementation. To access the first and next variable, the functions `first-var()` and `next-var(var)` are used (see pseudo code in Figure 5). The actual implementation of these functions reflects the chosen order of the variables. We consider four different heuristics:

VBP-INST. This heuristic processes the variables in the same order as the sequential QSIM algorithm instantiates the variables. The original QSIM algorithm reorders the constraints to avoid backtracking at early stages of the CSP algorithm — i.e., variables with one value are instantiated first. VBP-INST starts with splitting the domains of variables that are instantiated directly after the one-value variables of the original QSIM algorithm.

VBP-CON. The number of attached constraints of a given variable defines the order of this heuristic. The domains of variables which are shared by many constraints are partitioned first. Thus, many tuple sets are divided into subsets.

VBP-DOM. The cardinalities of the domains determine the order of the variables. The tuple sets of constraints attached to variables with large domains are partitioned first. This results in the generation of many subsets of the tuple sets.

VBP-TUP. The order of variables is based on the total number of tuples of all attached constraints. This heuristic divides the largest tuple sets first.

All four of these heuristics are advantageous for certain types of CSPs. However, we are interested in an efficient parallel algorithm for qualitative simulation. Hence, we must determine which heuristic results in both a minimum total workload and equally

sized subproblems for typical CSPs of qualitative simulation. The next section includes a comparison of the partitioning heuristics based on speedup estimations using QSIM CSPs.

4 Complexity Analysis and Speedup Estimation

4.1 Complexity Analysis

The asymptotic complexity of our parallel constraint satisfaction algorithm is determined by the complexities of the three consecutive steps of this algorithm: partitioning the search space with VBP, solving the subproblems in parallel and merging the results of the subproblems to the overall result. The complexity of solving a CSP or a polynomial number of subproblems in parallel is NP-complete [12]. The complexity of the final step is linear with the number of solutions, since the overall result is simply merged by the union of the results of the subproblems.

For VBP, we first derive the worst-case complexity for general CSPs and then refine the results by taking into account the properties of QSIM CSPs. The elementary operation in this complexity analysis is defined as the local check in the procedure `generate-subproblem` (lines 8–11 in Figure 6). Hence, the number of local checks is given by

$$(\text{calls to } \texttt{generate-subproblem}) \cdot (\text{local checks in } \texttt{generate-subproblem}).$$

`generate-subproblem` is called by the procedure `vbp` to check a candidate subspace. The number of recursion levels in `vbp` is limited by $|V|$; at each level a variable v_i is partitioned which induces $|D_i|$ recursive calls. Hence, the number of candidate subspaces cannot exceed $\prod_{i=1}^{|V|} |D_i|$.

The number of local checks within `generate-subproblem` is given by

$$|C| \cdot (\text{no. of tuples per constraint}) \cdot (\text{no. of variables per constraint}).$$

For general CSPs, the number of variables attached to a constraint cannot be restricted, i.e., the worst-case constraint arity equals $|V|$. Thus, the number of local checks is given by

$$|C| \cdot (\prod_{i=1}^{|V|} |D_i|) \cdot |V|. \tag{3}$$

The total number of local checks for general CSPs is determined by combining the number of candidate subspaces and the number of local checks per procedure call:

$$|C| \cdot |V| \cdot (\prod_{i=1}^{|V|} D_i)^2 \tag{4}$$

In QSim CSPs, the constraint arity is limited by 3 and the domain size is limited by 4. This means, a QSim constraint has at most 64 tuples and a maximum of 3 variables are attached to it. The maximum number of candidate subspaces is given by $4^{|V|}$ which leads to a total number of local checks of:

$$192 \cdot |C| \cdot 4^{|V|} \tag{5}$$

This exponential complexity is caused by our worst-case assumption that `vbp` partitions the domains of all variables. In such a case, the generated candidate subspaces are already candidate solutions, i.e., the domain size of each variable equals 1. These candidates are then checked by `generate-subproblem`; the successive backtracking algorithm becomes obsolete. This situation is comparable to solving the original CSP by enumerating all possible combinations of variable values and then checking each combination for local consistency.

The key to our implementation of the VBP method is that we control the number of candidate subspaces by the parameter `maxsub`. This parameter is defined by the user and the choice is mainly influenced by the number of processing elements. Particularly, `maxsub` is independent of the properties of the CSP, e.g., $|C|$ and $|V|$. Hence, the complexity of our VBP method is given as

$$192 \cdot \texttt{maxsub} \cdot |C|, \tag{6}$$

which is linear in the number of constraints and in the number of requested subproblems.

4.2 Speedup Model

The speedup estimation of the parallel constraint satisfaction algorithm is based on worst-case and best-case execution times of the parallel implementation. In this section, we derive analytically a model for the speedup. In the next section, we compare the four partitioning heuristics using execution times from partitioning and running QSim CSPs on one processing element as input for this model.

The speedup is defined as the ratio of the sequential execution time t_{seq} and the parallel execution time t_{par} using n processors:

$$S(n) \quad = \quad \frac{t_{seq}}{t_{par}(n)} \tag{7}$$

The parallel execution time is the sum of the execution time of the partitioning algorithm t_{vbp}, the overall execution time of all subproblems on n processors t_{exe} and the execution time of merging the partial results t_{merge}:

$$t_{par}(n) \quad = \quad t_{vbp} + t_{exe}(n) + t_{merge} \tag{8}$$

In this speedup model, no communication times are considered. To achieve high speedups with the parallel CSP algorithm, the sequential parts of the overall runtime, t_{vbp} and t_{merge}, must be small compared to t_{exe}. In such a case, a further simplification of our model by neglecting t_{vbp} and t_{merge}, i.e., $t_{vbp} = t_{merge} = 0$, is justified. The analysis of the asymptotic complexity in Section 4.1 supports this simplification.

The parallel execution time of all subproblems t_{exe} is determined by the order in which the subproblems are executed. Worst-case and best-case parallel execution times, t_{exe-wc} and t_{exe-bc}, can be derived from the sequential execution time of the unpartitioned CSP t_{seq} and the execution time of all its subproblems t_i. We define the following execution

times:

$$\begin{array}{ll} t_{seq} & \text{execution time of the unpartitioned problem} \\ t_{tot} = \sum_{i=1}^{p} t_i & \text{total execution time of all } p \text{ subproblems} \\ t_{max} = max_{i=1}^{p}\{t_i\} & \text{execution time of the subproblem with the maximum workload} \end{array} \tag{9}$$

The worst-case order is given if the subproblem with the maximum workload is scheduled last and all other subproblems are equally distributed among the processors. If the number of processors is large, i.e., $n \geq p$, the parallel execution time is limited by t_{max}. The worst-case execution time t_{exe-wc} using n processors is given as

$$t_{exe-wc}(n) \;=\; \begin{cases} \frac{t_{tot}-t_{max}}{n} + t_{max} & \text{if } n < p \\ t_{max} & \text{otherwise.} \end{cases} \tag{10}$$

The best-case execution time t_{exe-bc} is determined as follows. If the number of processors is smaller than $\lceil \frac{t_{tot}}{t_{max}} \rceil$, all subproblems are equally distributed among the processors. Otherwise the parallel execution time is limited by t_{max}. More formally, the best-case execution time is given as

$$t_{exe-bc}(n) \;=\; \begin{cases} \frac{t_{tot}}{n} & \text{if } n < \lceil \frac{t_{tot}}{t_{max}} \rceil \\ t_{max} & \text{otherwise.} \end{cases} \tag{11}$$

The parallel execution time t_{par} on n processors is limited by

$$t_{exe-bc}(n) \;\leq\; t_{par}(n) \;\leq\; t_{exe-wc}(n). \tag{12}$$

The resulting limits for the speedup $S(n)$ are

$$\frac{t_{seq}}{t_{exe-wc}(n)} \;\leq\; S(n) \;\leq\; \frac{t_{seq}}{t_{exe-bc}(n)}. \tag{13}$$

By substituting the parallel execution times, the speedup limits can be expressed only by t_{seq}, t_{tot} and t_{max}:

$$\left.\begin{array}{ll} \frac{n \cdot t_{seq}}{t_{tot}+(n-1)\cdot t_{max}} & \text{if } n < p \\ \frac{t_{seq}}{t_{max}} & \text{otherwise} \end{array}\right\} \leq S(n) \leq \left\{\begin{array}{ll} n \cdot \frac{t_{seq}}{t_{tot}} & \text{if } n < \lceil \frac{t_{tot}}{t_{max}} \rceil \\ \frac{t_{seq}}{t_{max}} & \text{otherwise} \end{array}\right. \quad (14)$$

Given a CSP with the sequential execution time t_{seq} and a pair of parameters $\langle$*partitioning heuristic*, `maxsub`$\rangle$ determining t_{tot} and t_{max}, the speedup model in Equation 14 estimates the range of the achievable speedup as a function of n, the number of processors.

4.3 Comparison of the Partitioning Heuristics

The partitioning heuristics are compared using simulation models from QSim. These models are called Starling (STLG) [8] and RCS [6] and represent CSPs with different complexities. Simulation of the STLG model results in less complex CSPs (17 variables and 18 constraints); simulation of the RCS model results in more complex CSPs (45 variables and 48 constraints). The execution times of the CSPs from these two models differ by orders of magnitude [20].

The execution times t_{seq}, t_{tot} and t_{max} are determined by the following procedure using one processing element (digital signal processor TMS320C40 from Texas Instruments). First, the execution time of the unpartitioned CSP t_{seq} is measured. After the partitioning of the CSP, the individual execution times for searching each subspace t_i are measured. With these individual execution times, t_{tot} and t_{max} are determined.

Table 3 shows the execution times t_{tot} and t_{max} for the different heuristics of the VBP method for 16, 64 and 256 requested subproblems. The sequential execution times of the CSPs are $t_{seq} = 6.82$ ms for the STLG model and $t_{seq} = 805.63$ ms for the RCS model. The execution times from Table 3 are derived by summing up the execution times from several CSPs of the simulation models. As discussed in Section 3.3, the requirements for a good partitioning heuristic are (i) to keep the total workload small and (ii) to generate equally sized subproblems. It can be seen from Table 3 that VBP-DOM leads in all

heuristic	*model*	`maxsub` = 16		`maxsub` = 64		`maxsub` = 256	
		t_{tot} [ms]	t_{max} [ms]	t_{tot} [ms]	t_{max} [ms]	t_{tot} [ms]	t_{max} [ms]
VBP-INST	STLG	6.96	3.49	6.55	3.28	6.34	3.18
	RCS	739.48	497.93	746.44	495.45	865.42	259.95
VBP-CON	STLG	6.02	3.29	5.60	1.99	9.40	1.53
	RCS	1370.53	104.29	826.66	67.86	825.96	30.71
VBP-DOM	STLG	34.96	3.15	24.73	1.99	18.12	1.70
	RCS	3932.79	265.16	15119.29	252.02	4739.37	244.82
VBP-TUP	STLG	7.65	2.15	5.60	1.99	9.40	1.53
	RCS	1401.59	103.31	779.47	51.08	780.95	44.95

Table 3: Comparison of the four heuristics of the variable-based partitioning. The execution times of the partitioned subproblems t_{tot} and t_{max} are given in ms.

cases to remarkably higher total workloads than the other partitioning heuristics. For the other heuristics, t_{tot} is in some cases smaller than t_{seq}. On the basis of the first requirement, a small total workload, only VBP-DOM can be discarded. The influence of the number of requested subproblems on t_{tot} and t_{max} is interesting. While an increased number of requested subproblems reduces t_{max} in all cases, the total execution time t_{tot} does not show such regular behavior. Moreover, in a parallel implementation a high number of subproblems increases the communication. Hence, the parameter `maxsub` has to be adjusted in relation to the number of processors and cannot be used to select or discard a partitioning heuristic in general. Concerning the second requirement, equally sized subproblems, our investigations revealed that the heuristic VBP-CON performs best on the widest range of QSim CSPs. Although for some particular CSPs, VBP-TUB and even VBP-INST also perform well. For the implementation of the parallel CSP algorithm, the partitioning heuristic VBP-CON was chosen.

A graphical comparison of the speedup limits of the heuristics VBP-INST and VBP-CON for the model RCS is shown in Figure 7. The speedups are presented for up to 8 processors and for 16, 64 and 256 requested subproblems. Figure 7 shows clearly that VBP-INST leads to a saturation of the achievable speedup due to the high values of t_{max}.

5 Multi-Processor Implementation

5.1 MIMD Architecture

The parallel constraint satisfaction algorithm for QSim is implemented on a MIMD multiprocessor system consisting of digital signal processors (DSP) of type TMS320C40 [21]. The processing elements are connected in a tree structure via their communication ports. Each processing element is operated at a clock frequency of 50 MHz and is equipped with a local memory of 8 MByte. The root processing element of the tree structure (master

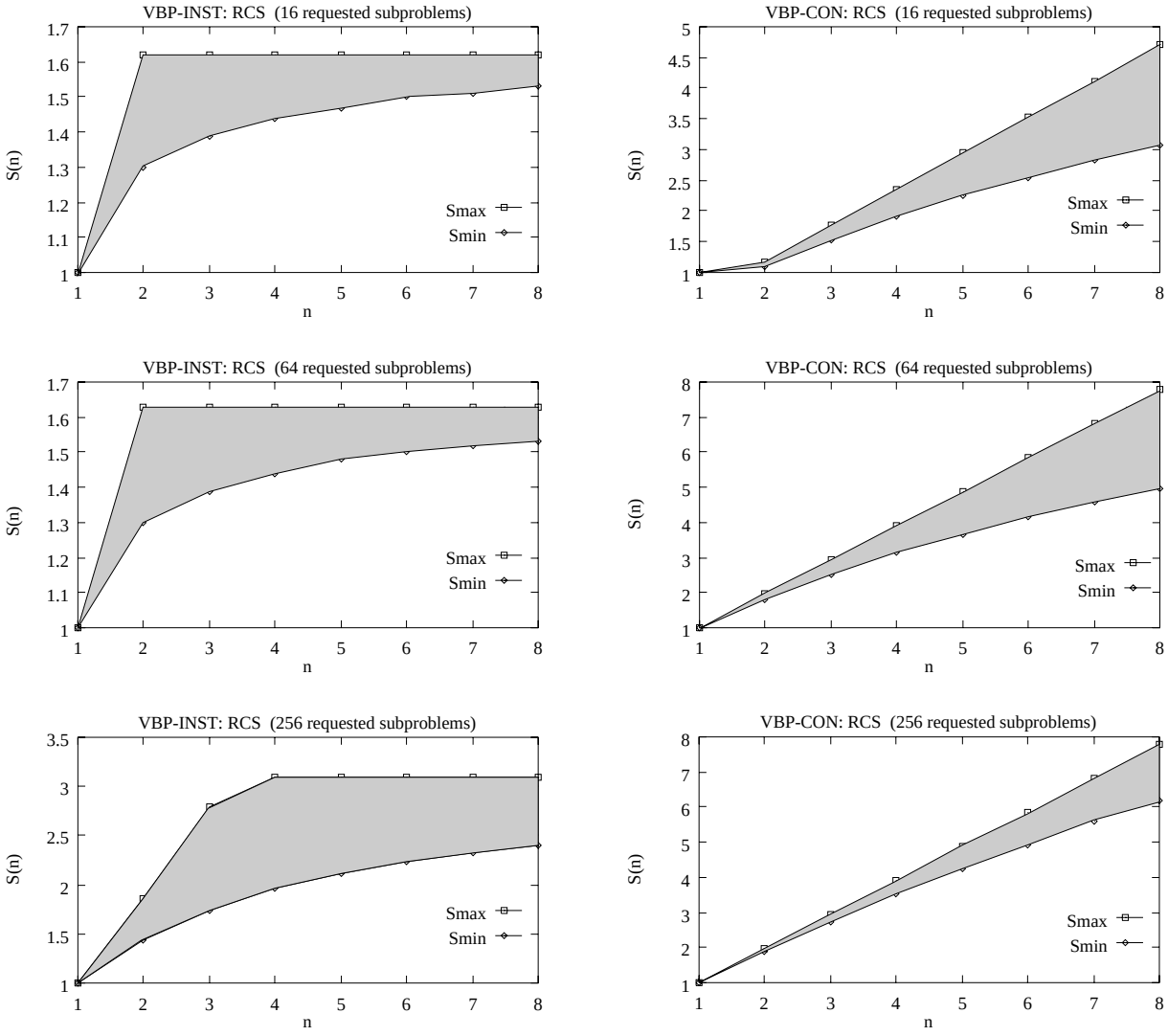


Figure 7: Speedup limits of VBP for the RCS model. Speedup limits for the partitioning heuristics VBP-INST and VBP-CON are shown in the left and right column plots.

processor) executes the first and the last step of the parallel CSP algorithm, i.e., the partitioning of the CSP and the merging of the partial results. All other processing elements (slave processors) are responsible for solving the subproblems. Hence, the subproblems are sent from and their results are received by the master processor; no communication is required between slave processors. Software is implemented in 'C' using the distributed real-time operating system Virtuoso [22]. This operating system is specially designed for multi-DSP architectures and allows a flexible and portable implementation.

In QSim, CSPs with the same constraint network are often solved successively. In these CSPs, the sets of variables V and constraints C remain the same; only the set of domains D and, thus, the set of tuples T change. To reduce the communication overhead of the parallel implementation, the subproblems are transferred to the individual processing elements in two steps. In the initial step, the sets of variables and constraints are broadcasted to all slave processors. After CSP partitioning, only the sets of tuples of each subproblem are sent to the individual slave processors. This avoids retransmission of the unchanged constraint network.

5.2 Experimental Results

The parallel implementation of constraint satisfaction in QSim is evaluated using input data from the QSim simulation models STLG, RCS and QSEA. Simulation of the QSEA model (38 variables and 37 constraints) results in CSPs with the longest sequential execution times [20]. Table 4 presents the execution times and the speedups measured for one CSP — as opposed to the summarized execution times in Table 3 — during the simulation of each model. Table 4 shows the sequential execution time t_{seq}, the time required for the partitioning algorithm VBP-CON t_{vbp} and the parallel execution time of all subproblems $t_{exe-merge}$. The time required for broadcasting the sets of variables and constraints, for merging the partial results and for transmitting the subproblems and their results is

included in the execution time $t_{exe-merge}$. The total parallel execution time t_{par} is the sum of t_{vbp} and $t_{exe-merge}$. The execution times of the parallel algorithm were measured using 1 to 7 slave processors, and 32 subproblems were requested from the partitioning algorithm VBP-CON.

Parallel execution of the STLG CSP reveals an interesting behavior. First, superlinear speedup is observed. This occurs because the partitioning algorithm discards many inconsistent subproblems. The total execution time of the remaining consistent subproblems is smaller than the execution time of the unpartitioned problem ($t_{tot} < t_{seq}$). Second, the speedup decreases with the number of slave processors. This occurs because the rejection of subproblems of VBP-CON is so effective for this CSP that only one subproblem can be generated. The overhead increases with the number of slave processors due to the broadcasting of the sets of variables and constraints. Therefore, the speedup decreases. A speedup greater than 1 using one slave processor is also observed for the RCS model. In this case, the partitioning algorithm also discards many inconsistent subproblems, but more than one subproblem is generated. Therefore, the speedup increases with the number of slave processors. Figure 8 presents graphically the speedups for these models. The speedup of the parallel backtracking implementation is shown for 1 to 7 slave processors.

6 Discussion

This paper presented a parallel constraint satisfaction algorithm for the qualitative simulator QSim. This parallel algorithm is based on a parallel-agent-based strategy which partitions the overall search space of the CSP into independent subproblems. The variable-based-partitioning (VBP) method was introduced to generate the independent subproblems. Based on an analytical speedup model, four different heuristics of VBP were evaluated and compared. The partitioning heuristic VBP-CON achieves the best results for the

model	t_{seq}	t_{vbp}	$\#slaves$	$t_{exe-merge}$	$S(n)$
STLG	4.095 ms	0.808 ms	1	1.052 ms	2.20
			2	1.170 ms	2.07
			3	1.271 ms	1.97
			4	1.416 ms	1.84
			5	1.647 ms	1.67
			6	1.769 ms	1.59
			7	1.906 ms	1.51
RCS	47.372 ms	5.565 ms	1	26.481 ms	1.48
			2	15.187 ms	2.28
			3	11.483 ms	2.78
			4	9.628 ms	3.12
			5	9.208 ms	3.21
			6	8.464 ms	3.38
			7	8.261 ms	3.43
QSEA	1298.144 ms	7.050 ms	1	1320.989 ms	0.98
			2	679.746 ms	1.89
			3	448.521 ms	2.85
			4	352.248 ms	3.61
			5	294.869 ms	4.30
			6	260.055 ms	4.86
			7	249.284 ms	5.06

Table 4: Execution times and speedups of the parallel CSP algorithm for the QSIM models STLG, RCS and QSEA using 1 to 7 slave processors. The total parallel execution time t_{par} is the sum of t_{vbp} and $t_{exe-merge}$.

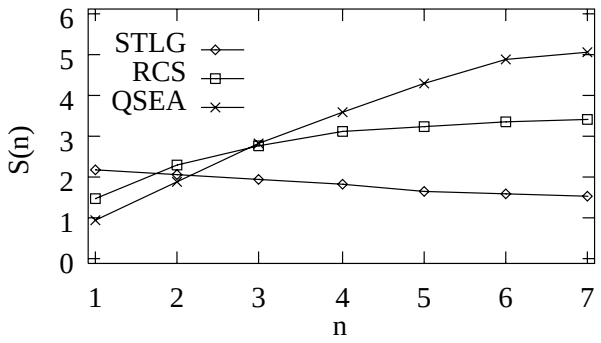


Figure 8: Speedup $S(n)$ of the parallel CSP implementation for the models STLG, RCS and QSEA using $n = 1 \ldots 7$ slave processors.

QSIM CSPs. The parallel algorithm was implemented on a MIMD processor architecture where the speedup of three CSPs from QSIM models were measured.

With a speedup of $S = 5.06$ for 7 slave processors for the CSP of the QSEA model and $S = 3.43$ for the RCS model, the experimental results proved that our partitioning method VBP and the chosen heuristic VBP-CON are efficient. The actual achieved speedup depends strongly on the input simulation model of QSIM, i.e., the CSPs derived from these models. In general, CSPs with long sequential execution times achieve higher speedups. From the viewpoint of 'real-world' applications of qualitative simulation, simulation models with many variables and constraints must be expected — resulting in CSPs with long execution times. We expect that our parallel CSP algorithm will perform even better with these 'real-world' models than with the 'experimental' models used in this evaluation.

The speedup limits derived from our analytical model match in most cases with the measured speedups of the parallel implementation. However, the speedup model is based on the assumption that the times for partitioning, merging and communicating are negligible compared to the parallel execution time. If this assumption is violated, the achieved speedup can differ from the estimations. This happens, for example, for the RCS model with 64 requested subproblems. In this case, the achieved speedup for more than 3 slave processors is smaller than the estimated worst-case. However, for the most complex CSP, the CSP of the QSEA model, and also for the CSPs of the expected 'real-world' QSIM models, the partitioning time is clearly negligible compared to the parallel execution time (compare Table 4). This validates our speedup model and, hence, confirms the chosen partitioning heuristic.

The application of the parallel CSP algorithm presented in this paper is not limited to the qualitative simulator QSIM. Our algorithm can be applied to any CSP that is described by sets of constraints, variables and domains as defined in Section 1 and where all

solutions of the CSP are required. The use of the dual constraint network representation in this algorithm is actually no restriction, since CSPs can be transformed between the different representations. The limitations on the constraint arity and the domain size in QSim CSPs influence the choice of the partitioning heuristic. For other types of CSPs, partitioning heuristics other than VBP-CON may be more effective. The partitioning strategy VBP itself does not pose any limitation on the constraint arity and the domain size. Therefore, the presented parallel CSP algorithm will probably not only significantly reduce the runtimes of 'real-world' applications in qualitative simulation but will also be applicable to a wide range of constraint satisfaction tasks.

Acknowledgment

This research project was partially supported by the Austrian Science Fund *Fonds zur Förderung der wissenschaftlichen Forschung* under grant number P10411-MAT.

References

[1] B. Burg. "Parallel Forward Checking: First part". Technical Report TR-594, Institute for New Generation Computer Technology, September 1990.

[2] B. Burg. "Parallel Forward Checking: Second part". Technical Report TR-595, Institute for New Generation Computer Technology, September 1990.

[3] P. R. Cooper and M. J. Swain. "Arc consistency: parallelism and domain dependence". *Artificial Intelligence*, 58:207–235, 1992.

[4] R. Dechter. "Constraint Networks". In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 276–285. John Wiley & Sons, Inc., 1992.

[5] S. Kasif. "On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks". *Artificial Intelligence*, 45:275–286, 1990.

[6] H. Kay. "A qualitative model of the space shuttle reaction control system". Technical Report AI92-188, Artificial Intelligence Laboratory, University of Texas, September 1992.

[7] B. Kuipers. "Qualitative Simulation". *Artificial Intelligence*, 29:289–338, 1986.

[8] B. Kuipers. *"Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge"*. Artificial Intelligence. MIT Press, 1994.

[9] W. Lin and B. Yang. "Fast Parallel Tree Search with Static Load-Balancing Forward Checking Technique". In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 33–38, Orlando, Florida U.S.A., September 1995.

[10] W. Lin and B. Yang. "Probablistic Performance Analysis for Parallel Search Techniques". *International Journal of Parallel Programming*, 1995.

[11] Q. P. Luo, P. G. Hendry, and J. T. Buchanan. Strategies for Distributed Constraint Satisfaction Problems. In *Proceedings 13th International DAI Workshop*, Seattle, WA, 1994. DAI.

[12] A. K. Mackworth. "Constraint Satisfaction". In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 285–293. John Wiley & Sons, Inc., 1992.

[13] A. K. Mackworth and E. C. Freuder. "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems". *Artificial Intelligence*, 25:65–74, 1985.

[14] R. Mohr and T. C. Henderson. "Arc and Path Consistency Revisited". *Artificial Intelligence*, 28:225–233, 1986.

[15] M. Platzner and B. Rinner. "Improving Performance of the Qualitative Simulator QSIM — Design and Implementation of a Specialized Computer Architecture". In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 494–501, Orlando, USA, September 1995.

[16] M. Platzner, B. Rinner, and R. Weiss. "Exploiting Parallelism in Constraint Satisfaction for Qualitative Simulation". *J.UCS The Journal of Universal Computer Science*, 1(12):811–820, December 1995.

[17] M. Platzner, B. Rinner, and R. Weiss. "Parallel Qualitative Simulation". *Simulation Practice and Theory — International Journal of the Federation of European Simulation Societies*, 5(7-8):623–638, 1997. Elsevier Science Publishers B.V.

[18] P. Prosser. "Hybrid Algorithms for the Constraint Satisfaction Problem". *Computational Intelligence*, 9(3):268–299, 1993.

[19] V. N. Rao and V. Kumar. "On the Efficiency of Parallel Backtracking". *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993.

[20] B. Rinner. *"Design, Implementation and Experimental Evaluation of a Scalable Multiprocessor Architecture for Qualitative Simulation"*. PhD thesis, Graz University of Technology, 1996.

[21] R. Simar, P. Koeppen, J. Leach, S. Marshall, D. Francis, G. Mekras, J. Rosenstrauch, and S. Anderson. "Floating-Point Processors Join Forces in Parallel Processing Architectures". *IEEE Micro*, pages 60–69, August 1992.

[22] E. Verhulst. "Virtuoso: A virtual single processor programming system for distributed real-time applications". *Microprocessing and Microprogramming*, 40:103–115, 1994.

[23] Y. Zhang and A. K. Mackworth. "Parallel and Distributed Finate Constraint Satisfaction: Complexity, Algorithms and Experiments". In L. Kanal, V. Kumar, H. Kitano, and C. Suttner, editors, *Parallel Processing for Artificial Intelligence*, chapter 1. Elsevier Sience Publishers B.V., 1993.