Parallel Execution Models for Constraint Programming over Finite Domains *

Alvaro Ruiz-Andino, Lourdes Araujo, Fernando Sáenz, and José Ruz

Department of Computer Science
University Complutense of Madrid

Abstract. Many problems from artificial intelligence can be described as constraint satisfaction problems over finite domains (CSP(FD)), that is, a solution is an assignment of a value from a finite domain to each problem variable such that a set of constraints is satisfied. Arc-consistency algorithms remove inconsistent values from the set of values that can be assigned to a variable (its domain), thus reducing the search space. We have developed two parallelisation models of arc-consistency to be run on MIMD multiprocessors. Two different policies, static and dynamic, to schedule the execution of constraints have been tested. In the static scheduling policy, the set of constraints is divided into N partitions, which are executed in parallel on N processors. We discuss an important factor affecting performance, the criterion to establish the partition in order to balance the run-time workload. In the dynamic scheduling policy, any processor can execute any constraint, improving the workload balance. However, a coordination mechanism is required to ensure a sound order in the execution of constraints. Both parallelisation models have been implemented on a CRAY T3E multiprocessor with up to thirty four processors. Empirical results on speedup and behaviour of both models are reported and discussed.

1 Introduction

Constraint Programming over finite domains (CP(FD)) [5,7] has been used for specifying and solving complex constraint satisfaction and optimisation problems, as resource allocation, scheduling and hardware design [6,17]. Finite domain Constraint Satisfaction Problems (CSP) usually describe NP-complete search problems, but it has been shown that by working locally on constraints and their related variables it is possible to dynamically prune the search space in an efficient way. Techniques following this approach, called arc-consistency algorithms, eliminate inconsistent values from the solution space. They can be used to reduce the size of the search space both before and while searching. Waltz [18] proposed the first arc-consistency algorithm, and several improved versions are described in the literature: AC-3 [10], AC-4 [11], AC-5 [15], and AC-6 [1].

AC-3, AC-4 and AC-6 deal with extensional constraints, that is, constraints are expressed as the set of tuples that satisfies it, whereas AC-5 can be specialised

* Supported by project TIC98-0445-C03-02.

for functional, anti-functional and monotonic constraints. This specialisation provides an efficient decision procedure for the basic constraints of constraint programming languages.

We have developed and tested two parallelisation models of arc-consistency for MIMD distributed shared memory multiprocessors. These models arise from two policies of *scheduling* the constraints to be processed, static and dynamic. In the static model, the set of constraints is partitioned into N partitions, which are processed in parallel on N processors. We discuss the two main issues affecting the performance of this model: the criterion to distribute constraints among processors, and the frequency of updating shared variables. In the dynamic model any processor can process any constraint, improving the workload balance. However, a coordination mechanism is required to ensure a sound processing order of constraints.

Several parallel processing methods for solving CSPs have been proposed. In [20], a parallel constraint solving technique for a special class of CSP, acyclic constraint networks, is developed. It also presents some results on parallel complexity, generalising results in [8]. In [9], it is concluded that parallel complexity of constraint networks is critically dependent on subtle properties of the network which do not influence its sequential complexity. They propose massively parallel processing of arc-consistency with also very simple processing elements.

In [2,12] Nguyen, Deville and Baudot proposed distributed versions for AC-3, AC-4, and AC-6 for binary CSPs, based on a static scheduling. Our work considers both static and dynamic scheduling policies, and it is focused on the AC-5 specialisation for functional, anti-functional and monotonic n-ary constraints. More precisely, it is a parallelisation of the *indexical* scheme [4,3,16]. We have integrated the parallel execution of arc-consistency within a labelling process that searches for solutions to the constraint satisfaction problem, embedded in a constraint logic programming language. Labelling is performed sequentially, that is, parallel arc-consistency phases are interleaved with variable-value assignment phases, synchronous and identically performed by every processing element, in contrast with other distributed constraint satisfaction techniques as [19].

The rest of the paper is organised as follows. Next section describes basic concepts of constraint programming over finite domains of integers. Section 3 discusses the parallelism presented by the arc-consistency algorithm and introduces two models to exploit it. Section 4 describes the static scheduling execution model, whereas Section 5 is devoted to the dynamic one. Section 6 reports and discusses the experimental results. Finally, conclusions are drawn in section 7.

2 Constraint Programming

A constraint satisfaction problem over finite domains may be stated as follows. Given a tuple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where

- $\mathcal{V} \equiv \{v_1, \cdots, v_n\}$, is a set of domain variables,
- $\mathcal{D} \equiv \{d_1, \cdots, d_n\}$, is the set of an initial *finite domain* (finite set of values) for each variable,

- $\mathcal{C} \equiv \{c_1, \cdots, c_m\}$, is a set of constraints among the variables in $\mathcal{V}$. A constraint $c \equiv (V_c, R_c)$ is defined by a subset of variables $V_c \subseteq \mathcal{V}$, and a subset of allowed tuples of values $R_c \subseteq \bigotimes_{i \in \{j/v_j \in V_c\}} d_i$, where $\bigotimes$ denotes Cartesian product.

The goal is to find an assignment for each variable $v_i \in \mathcal{V}$ of a value from each $d_i \in \mathcal{D}$ which satisfies every constraint $c_i \in \mathcal{C}$.

A constraint $c \equiv (V_c, R_c) \in \mathcal{C}$, $V_c \equiv \{v_1, \cdots, v_k\}$, is arc-consistent with respect to domains $\{d_1, \cdots, d_k\}$ iff for all $v_i \in V_c$, for all $a \in d_i$, there exists a tuple $(b_1, \cdots, b_{i-1}, a, b_{i+1}, \cdots, b_k) \in R_c$, where $b_j \in d_j$. A CSP is called arc-consistent iff all $c_i \in \mathcal{C}$ are arc-consistent with respect to $\mathcal{D}$.

The starting point of this work is a sequential constraint solver which implements consistency using the *indexical scheme* [4,3,16]. In this scheme, a constraint is translated into a set of reactive functional expressions, called *indexicals*, which maintain consistency. An indexical has the form "v in $E(V)$", where $v \in \mathcal{V}$, $V \subseteq \mathcal{V}$, and $E(V)$ is a monotonic functional expression which returns a finite set of values. Given an indexical $I \equiv v$ in $E(V)$, we call V its *set of arguments*, and we say that, for all $v_i \in V$, I *depends on* v_i, and I *writes* the domain variable v. A constraint $c \equiv (V_c, R_c)$ relating the set of domain variables $V_c \equiv \{v_1, \cdots, v_k\}$, is translated into a set of k indexicals $\{I_i \equiv v_i$ in $E_i(V_c - \{v_i\})\}$. Each indexical I_i writes variable v_i and depends on the remaining $k-1$ variables. Functional expressions $E_i(V_c - \{v_i\})$ are properly defined for arc-consistency to be achieved (removal of inconsistent values) with respect to constraint c. Most common high level constraints, such as arithmetic, symbolic and relational ones can be easily translated to indexicals.

The set of finite domains that keeps the current domain of each variable in $\mathcal{V}$ is called the *store*. The initial value of the store is defined by $\mathcal{D}$. The execution of an indexical v in $E(V)$, is triggered by changes in the domains of its set of arguments V in a data driven way. When an indexical is executed, the domain of v in the store is updated with $d_v \cap Eval(E(V))$, where d_v denotes the current value of the domain of v in the store, and $Eval(E(V))$ denotes the evaluation of $E(V)$ with the current domains of the set of variables V in the store.

Figures 1 and 2 show the sequential arc-consistency algorithm. Its input argument is the CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ whose arc-consistency is to be achieved. The set of constraints $\mathcal{C}$ is expressed as a set of indexicals. The algorithm returns either a store where the domain for each variable has been pruned achieving arc-consistency, or `FAILURE` if inconsistency is detected (the domain of a variable was pruned to an empty domain).

A sequential arc-consistency algorithm executes indexicals until either the fixed point is reached, or inconsistency is detected. The fixed point is reached iff the store is arc-consistent. A propagation queue is used to schedule the execution of indexicals (`PropagationQueue`, figure 1). As the result of the execution of an indexical (`Arc_Consistent()`), the domain of a variable may be pruned, and in such a case the variable is queued (`Update()`). Initially, all indexicals are executed, initialising the `PropagationQueue` (line `1`). The main loop (lines `2` to `9`) iterates until either the propagation queue is empty, or inconsistency is

```
function Arc-Consistent-CSP($\langle VarSet, DomSet, ConstrSet\rangle$): Store
begin
  Queue_Init(PropagationQueue, $\cdots$ );
  while NOT Empty(PropagationQueue) do
    Queue_Pop(PropagationQueue, $v_i$);
    for each indexical $I_j$ which depends on $v_i$ do
      if NOT Arc_Consistent($I_j$,Store,PropagationQueue) then
        return FAILURE;
      end-if;
    end-for;
  end-while;
  return Store;
end;
```

Fig. 1. Arc consistency algorithm.

detected. In each iteration, a variable is dequeued and those indexicals that depend on it are executed.

```
function Arc_Consistent( '$v_i$ in E()',
                         Var Store, Var PropQueue ) : Boolean
begin
  NewDomain := Eval(E(), Store);
  return (Update(NewDomain,$v_i$,Store,PropQueue) <> EMPTY);
end;

function Update( NewDomain, $v_i$, Var Store,
                             Var PropQueue): RESULT
begin
  NewDomain := NewDomain $\cap$ Store[$v_i$];
  if Empty(NewDomain) then return EMPTY; end-if;
  if (NewDomain $\subset$ Store[$v_i$]) then
    Store[$v_i$] := NewDomain;
    Queue_Push($v_i$, PropQueue);
    return PRUNED;
  end-if;
  return NOT_PRUNED;
end;
```

Fig. 2. Store and propagation queue updating.

Termination, correctness, complexity, and properties of the algorithm have been studied extensively in the literature [15,14,3]. Correctness is independent of the order of reexecution of indexicals, which constitutes the basis for the correctness of the parallel version of the algorithm.

3 Parallel Arc-Consistency

The arc-consistency algorithm presents inherent parallelism. Each indexical behaves as a concurrent process which updates the store, triggered by changes in the store. There is an inherent sequentiality, as well, since an indexical must be executed only as the consequence of a previous execution of another indexical. This sequentiality defines a partial order among (re)execution of indexicals. An indexical is *ready* if any of its arguments has changed after its last execution. At any time during the execution of the arc-consistency algorithm there will be a set of ready indexicals, called the *ready set.* In a sequential version of a consistency algorithm the ready set is stored in a *propagation queue* (updated whenever a variable is modified), ensuring a sound execution order of indexicals, that is, that an indexical is executed after the pruned variable has been updated. Parallel consistency algorithms simultaneously execute the indexicals in the ready set, providing mechanisms to maintain a sound order.

We have investigated the feasibility of both static and dynamic scheduling policies for execution of indexicals.

In the static scheduling model, the set of indexicals is divided into N partitions, which are executed in parallel on N processors. A static scheduling ensures a sound execution order of indexicals, since the parallel algorithm is basically the sequential one, but applied to a subset of the indexicals. The only coordination mechanism needed by this model comes from the detection of termination, which can be carried out by one of the processors, called the *distinguished* one. The mapping of indexicals to processors is generated previously to the execution of arc-consistency. An important factor for the efficiency of this model is the criterion for the distribution of indexicals among processors, therefore different criteria have been investigated.

A dynamic scheduling policy requires a coordination mechanism to guarantee a sound execution order. Section 5 discusses the dynamic scheduling model where a sound execution order is achieved by means of synchronisation points.

Parallelisation of the consistency algorithm requires every processor to have access to a common store. Since the considered parallelisation models are focused on distributed shared memory architecture, each processor has a (partial) local copy of the store. Changes in the variables' domains must be communicated to concerned processors in order to maintain coherency among local copies of the store.

4 Static Scheduling of Indexicals

The set of indexicals $\mathcal{C}$ is partitioned into n disjoint subsets, $\mathcal{C} = C_1 \cup \cdots \cup C_n$. This partitioning induces a distribution of the set of domain variables $\mathcal{V}$ in n not necessarily disjoint subsets $V_1, \cdots, V_n$ ($\mathcal{V} = V_1 \cup \cdots \cup V_n$). For all indexicals $I_j \in C_i$, the variable written by I_j, and those variables on which I_j depends on, constitute V_i ($\forall I_j \in C_i, I_j \equiv v$ in $E(V_j), V_i = \{v\} \cup V_j$.) Figure 3 sketches the partitioning process of the CSP.

Partitions $\langle V_i, D_i, C_i \rangle$ are mapped one-to-one to processing elements P_i. Each processing element P_i performs sequential arc-consistency, executing those indexicals in C_i, and consequently updating local copies of variables in V_i. Since the distribution of the set of variables $\mathcal{V}$ is non-disjoint, some variables will be located at several processing elements. Therefore, each processing element P_i must broadcast the pruning of the domain of variable v to every processing element P_j which had been assigned any of those indexicals which depend on v. Upon receiving the notification, processing elements P_j intersect their local copies of the domain with the incoming domain, probably triggering further propagation. Communication among processors is also needed in order to detect termination of the algorithm, either because of reaching the global fixed point, or because of inconsistency detection.

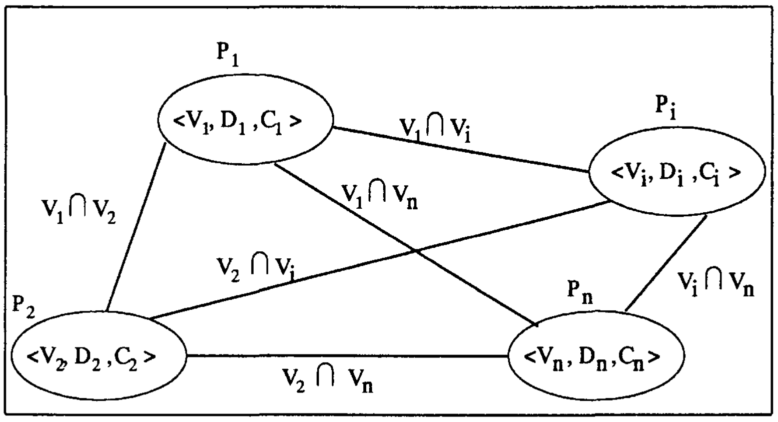

Fig. 3. Partitioning the CSP. Sub-CSP $\langle V_i, D_i, C_i \rangle$ is assigned to processing element (PE) P_i. An edge between two PE's is labelled with the set of variables located at both PE's ($V_i \cap V_j$). Communication is needed to maintain the same domain for some of the variables in $V_i \cap V_j$.

4.1 Parallel Algorithm

Figures 4 and 5 show the parallel execution algorithm. As in the sequential one, initially every indexical assigned to the processor is executed, initialising the local propagation queue (line 3). The main loop (lines 4 to 23) is executed until either global fixed point (`GlobalFixedPoint`) or inconsistency (`Failure`) is detected. The latter can be caused either by:

- an empty domain results from the execution of a local indexical (`Local_Arc_Consistent()`).
- an empty domain results from the intersection of the local domain of a variable with the domain received from another processor (`Remote_Arc_Consistent()`).
- inconsistency is detected at (and broadcasted from) another processor (`RemoteFailure`).

Each processor maintains a private propagation queue (LocalPropQueue). The inner loop (lines 6 to 14) performs local propagation until either the queue is empty or inconsistency is detected, like the main loop of the sequential algorithm. Once a local fixed point is reached, the processor notifies it to the *distinguished* processor of this status (Notify_Local_Fixed_Point()), and it waits (lines 17 to 21) until either:

- global fixed point is detected (Check_Global_Fixed_Point()).
- some other processor communicates inconsistency (RemoteFailure).
- the processor receives a message which updates its local propagation queue. In this case, the processor notifies it (Notify_Active()) to the distinguished one and continues executing indexicals.

```
    function Parallel-Consistency(
                      ⟨$VarSubSet, DomSubSet, ConstrSubSet$⟩ ) : Store
    begin
 1     Parallel_State_Reset();
 2     Synchronisation;
 3     Queue_Init(LocalPropQueue, ··· );
 4     while NOT Failure AND NOT GlobalFixedPoint do
 5        Notify_Active();
 6        while NOT Failure AND NOT Empty(LocalPropQueue)) do
 7           Queue_Pop (LocalPropQueue, $v_i$);
 8           for each indexical $I_i$ which depends on $v_i$ do
 9              Failure := RemoteFailure OR
10                 NOT Local_Arc_Consistent($I_i$,Store,LocalPropQueue) OR
11                 NOT Remote_Arc_Consistent(Store,LocalPropQueue);
12              if Failure then break; end-if;
13           end-for;
14        end-while;
15        if NOT Failure then
16           Notify_Local_Fixed_Point(···);
17           repeat
18              Failure := RemoteFailure OR
19                      NOT Consistency_Msg(Store,LocalPropQueue);
20              GlobalFixedPoint := Check_Global_Fixed_Point();
21           until Failure OR GlobalFixedPoint OR Message_Received();
22        end-if;
23     end-while;
24     if Failure then
25        Synchronisation(); return FAILURE;
26     end-if
27     return Store;
    end-function;
```

Fig. 4. Static Parallel Consistency Algorithm.

When the local execution of an indexical (Local_Arc_Consistent(), figure 5) results in the modification of the domain of a variable v (Update(), figure 5), the processor broadcasts a message (Broadcast_Update(), line 7) to the set of processors that have been assigned any of those indexicals which depends on variable v. Upon receiving the message (Remote_Arc_Consistent(), figure 5), these processors either detect inconsistency or properly update their local propagation queue and their local copy of variable v. Whenever a processor detects inconsistency, it broadcasts the failure to the rest of processors (Broadcast_Failure()).

```
  function Local_Arc_Consistent( '$v_i$ in E()',
                                 Var Store, Var PropQueue ): Boolean
  begin
1   NewDomain := Eval(E(), Store);
2   switch (Update (NewDomain, $v_i$, Store, PropQueue))
3      case EMPTY :
4         Broadcast_Failure(RemoteFailure);
5         return FALSE;
6      case PRUNED:
7         Broadcast_Update($v_i$, Store[$v_i$]);
8   end-switch;
9   return TRUE;
  end-function;

  function Remote_Arc_Consistent( Var Store,
                                  Var PropQueue ) : Boolean
  begin
1   while NOT Empty(MsgQueue) do
2      Pop_Message(MsgQueue, $v_i$, NewDomain);
3      if (Update(NewDomain,$v_i$,Store, PropQueue) = EMPTY) then
4         Broadcast_Failure(RemoteFailure);
5         return FALSE;
6      end-if;
7   end-while;
8   return TRUE;
  end-function;
```

Fig. 5. Parallel consistency functions.

The algorithm terminates when every processor reaches a local fixed point and there are no pending messages. The distinguished processor is the only one responsible for the detection of termination. However, it performs local propagation as any other processor. In order to be able to detect the global fixed point, processors must notify to the distinguished one whenever they reach a local fixed point –along with the number of messages they have sent and received– (Notify_Local_Fixed_Point()), and whenever they leave it due to an incoming message (Notify_Active()). The distinguished processor keeps record of which processors are at a local fixed point, and the number of messages sent and received by all processors. When termination is detected, the distinguished processor notifies it to the rest of processors (`GlobalFixedPoint`).

Since this parallel algorithm is part of a labelling procedure where variable-value assignment is performed synchronously, a synchronisation point among all processing elements is needed at the beginning of the algorithm, just after the initialisation of the communication status variables (`Parallel_State_Reset()`, line 2, figure 4). Another synchronisation (line 25) is needed if the algorithm finishes with failure; otherwise, the global fixed point detection implies a synchronisation among processors. Synchronisation points guarantee that every processing element waits until the last processing element has finished the current arc-consistency cycle before it starts working on the next one.

4.2 Partition of the CSP

The way the set of indexicals is partitioned has shown to be an essential factor for the efficiency of the parallel algorithm. A CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ can be represented as a hyper-graph where the set of nodes is the set of domain variables $\mathcal{V}$ and the set of hyper-edges is the set of indexicals defined by $\mathcal{C}$. Therefore, partitioning the CSP among processors means partitioning the set of hyper-edges in disjoints subsets, inducing a not necessarily disjoint partitioning of the set of nodes. We have tested two different graph partition criteria:

- Strength of connection among partitions.
- Static estimation of run-time ready set distribution.

Strength of connection among partitions The graph topology can be considered in order to partition the graph in *strongly connected subgraphs*, or *highly disconnected subgraphs*.

In the former case, communications are minimised, but the ready set will be badly balanced, in general. A strongly connected partitioning induces an almost disjoint partitioning of the set of variables $\mathcal{V}$, thus avoiding communications. However, it is very likely that most of those indexicals which depend on a variable v are assigned to the same processing element P. Whenever variable v is pruned, the ready set is enlarged with those indexicals which depend on v, but almost all of them will be sequentially executed by P, thus loosing the potential parallelism exploitation.

In the latter case, the ready set is better balanced, but it is likely that almost every variable will be located at almost every processing element, increasing communications.

Experimental results show the benefit of a better balanced ready set versus a communications reduction. Moreover, partitioning the CSP in strongly connected subgraphs is a hard problem, whereas a highly disconnected CSP partitioning is easily achieved with a shuffle distribution of indexicals.

Static estimation of run-time ready set distribution A partition of the set of indexicals that balances the run-time ready set is expected to improve the performance, providing that communications do not increase. Since this model is based on a static partitioning of the CSP among processors, balancing run-time ready set requires some kind of compile-time static estimation.

The idea is to partition the set of indexicals in such a way that updating any variable causes a similar number of indexicals to be executed by each processor [13]. We have defined an objective function to be minimised, which considers the peak workload for each processor and variable. Experimental measures of run-time workload have confirmed the accuracy of our static estimation. Since we are dealing with n-ary constraints, finding the optimal solution is a NP problem. Therefore, we recourse to an algorithm which assigns indexicals one by one, in a decreasing arity order, greedily choosing the processor which minimises the objective function. Solutions found with this greedy algorithm have shown to be quite close to the optimal one when the CSP is constituted by a large number of low-arity constraints. Taking into account that this is just an estimation of the actual run-time ready set distribution, the greedy approach is fully justified.

5 Dynamic Scheduling of Indexicals

A dynamic scheduling policy dispatches the *ready set* of indexicals every *execution cycle*, in order to balance workload. However, these models require mechanisms to ensure that the indexicals depending on a variable are executed after the change in the domain of the variable have been updated in the store of the processor executing the indexical. The alternatives to achieve a sound execution order are either to introduce synchronisation points during the execution (distributed control) or to include a master processor (centralised control) to perform the dispatching of indexicals. The latter model leads to tasks of small granularity, inappropriate for a distributed memory architecture. Therefore, we concentrate on the distributed control alternative.

The dynamic parallelisation model is based on dividing the execution in synchronised *execution cycles.* An execution cycle consists of generating of the ready set, distributed selection and execution of the ready set, and a synchronisation point. In order to distribute the queued indexicals, every processor must generate identical propagation queues of indexicals. In this way, each processor independently selects and executes, according to a fixed rule, a different subset of indexicals of those present in the propagation queue. Synchronisation points between execution cycles are introduced in order to generate identical propagation queues. Besides, the store must be replicated in every processor.

The consistency algorithm for this model initially queues every indexical. Then, execution cycles are performed until either there are no indexicals to execute or inconsistency is detected. An execution cycle comprises the following actions:

- Each indexical in the queue is executed by a particular processor, until the queue is empty. The coordination criterion ensures that every queued index-

ical is selected by only one processor, and that the workload is well balanced in each execution cycle.
- Modified variables are recorded and broadcasted, but indexicals are not queued in the current propagation queue.
- Changes in domains of variables received from remote processors are updated and queued.
- Once the propagation queue is empty, and after a synchronisation, which ensures that all processors have the same value of the domains of the variables, a new propagation queue is generated, queuing all indexicals depending on modified variables.

Two criteria to select indexicals from the queue have been investigated in order to tune the model:

- Assigning the same number of indexicals to each processor. This criterion can lead to unbalanced workload since each indexical involves a different amount of work.
- Dynamic distribution, in which each processor selects, in mutual exclusion, the next pending indexical.

First criterion has yielded better results, showing that workload balance is good enough, while second criterion increases communication overhead.

6 Experimental Results

The presented parallel algorithms have been written in C, and developed and tested on a CRAY T3E multiprocessor with thirty four 400-MHz DEC Alpha processors, 128 Mb of memory per processor, under UNICOS (UNIX) operating system. Notification of failure, global and local fixed point detection, activity status, and number of messages sent and received, have been implemented using the remote memory write feature of the CRAY T3E multiprocessor. Queues of messages are used for receiving domain updates. Messages are broadcasted to queues also using the fast remote memory write feature.

Reported results correspond to the time required to reach the first or all solutions, depending on the benchmark, performing a first fail sequential labelling. Therefore, reported speedup is lower than speedup achieved in a single call to the arc-consistency algorithm, since the search for a solution usually comprises a large number of calls to the arc-consistency algorithm, executed in parallel, interleaved with the selection and assignment of a value to a variable, executed sequentially.

6.1 Benchmarks

We have tested the parallelisation models on a set of benchmarks:

1. *Arithmetic* is a synthetic benchmark. It is formed by sixteen blocks of arithmetic relations, $\{B_1, \cdots, B_{16}\}$. Each block contains fifteen equations and inequations among six variables. Blocks B_i, B_{i+1} are connected by an additional equation between a pair of variables, one from B_i and the other one from B_{i+1}. Coefficients were randomly generated. The goal is to find an integer solution vector.
2. *Suudoku* is a crypto-arithmetic Japanese problem. Given a grid of 25x25 squares, where 317 of them are filled with a number between 1 and 25, fill the rest of squares such that each row and column is a permutation of numbers 1 to 25. Furthermore, each of the twenty-five 5x5 squares starting in columns (rows) 1, 6, 11, 16, 21 must also be a permutation of numbers 1 to 25.
3. *N-Queens* problem consists in placing N queens in an NxN chess board in such a way that no queen attacks each other. The instance presented corresponds to N = 111, size which leads to a significant execution time.
4. *Parametrizable Binary Constraint Satisfaction Problem* (PBCSP). Synthetic PBCSPs allow studying the performance of arc-consistency algorithms as some significant problem parameters vary. Instances of this problem are randomly generated given four parameters: number of variables, the size of the initial domains, density, and tightness. All constraints are binary, that is, they involve only two variables. A constraint is defined as the set of pairs of values that satisfies it. Density and tightness are defined as follows:

$$Density = \frac{nc}{nv - 1} \qquad Tightness = 1 - \frac{np}{ds^2}$$

where nv is the number of variables, nc is the number of constraints involving one variable (it is the same for all variables), np is the number of pairs that satisfies the constraint, and ds is the size of the initial domains. Figure 6 reports results obtained for an instance of this problem where $nv = 100$, $ds = 20$, $Density = 0.75$, and $Tightness = 0.85$.

Table 1. Benchmarks characteristics.

	Arithmetic	*Suudoku*	*N-Queens*	*PBCSP*
Search for	first sol.	first sol.	first sol.	all sol.
No. of Variables	126	308	111	100
No. of Constraints	254	13,942	6,105	3,713
No. of Indexicals	1,468	27,884	12,210	7,426
No. of Calls to Consistency	15,969	72,196	8,660	65
Seq. Exec. Time (s.)	15.05	132.98	12.62	5.25
No. of ind. executed	1,953,660	9,764,960	246,262	318,552
Avg. time per call (ms.)	0.9	1.8	1.5	80.8

Table 1 summarises relevant data about the four benchmarks. The three first benchmarks are executed searching for the first solution, whereas the fourth one keeps searching until all solutions (40) are found. The table shows the number of variables, number of constrains and number of indexicals for all benchmarks, as well as the number of calls to the arc-consistency algorithm. It also reports the sequential execution time, and the total number of indexicals executed in the sequential version. Finally, the table reports the average execution time per call to the arc-consistency algorithm, which indicates the granularity of the process to be parallelised.

6.2 Speedup

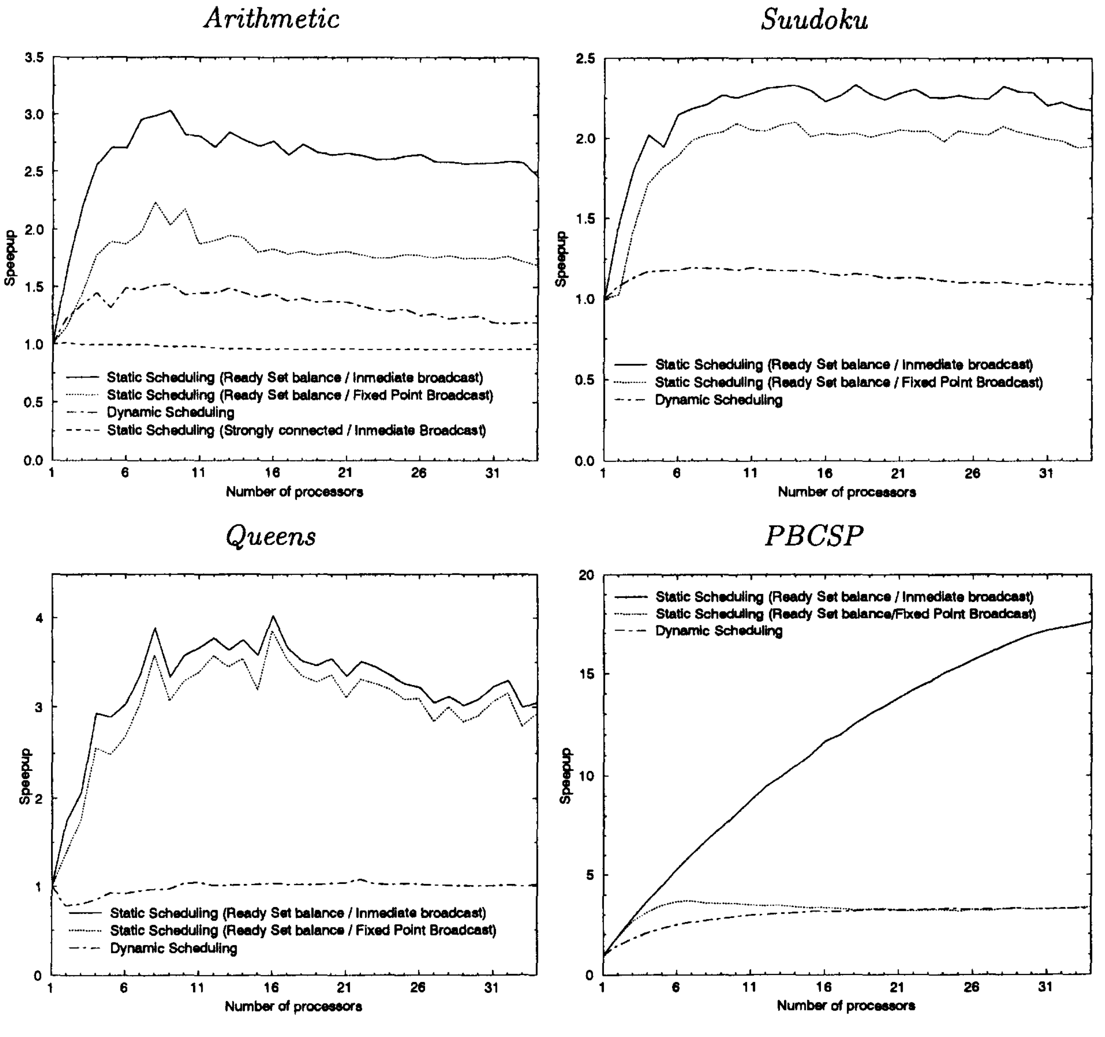

Fig. 6. Speedup curves for selected benchmarks.

Charts in figure 6 show, for each benchmark, the speedup vs. the number of processors. For the static scheduling policy the ready set balance estimation was used, comparing broadcast frequency: immediate (solid line) vs. fixed point (dotted line). Chart for the *Arithmetic* benchmark also shows the speedup obtained with a strongly connected graph partitioning (dashed line). This criterion has not been considered for the other benchmark, since it clearly provides worse results than ready set balance, and because it is too computationally expensive to apply. The speedup obtained with the dynamic scheduling policy (dot-dashed line) is worse than the best one of the static policy, mainly because the overhead due to the synchronisation points introduced in the dynamic model is too large versus the granularity of indexicals in the considered problems. However, this model could be more efficient using large granularity indexicals or propagators, as those arising from global constraints.

It can also be observed that whereas the *PBCSP* problem presents a nearly linear speedup for the best static scheduling policy, the speedup for the rest of benchmarks stops increasing from a certain number of processors. The main factor for this different behaviour is that in the *PBCSP* benchmark calls to the arc-consistency algorithm have a larger execution time, and indexicals executions have larger granularity (see Table 1). Besides, *PBCSP* has a constraint graph with a more uniform topology, leading to a better workload balance. In order to study this factor we have measured the workload distribution among the processing elements.

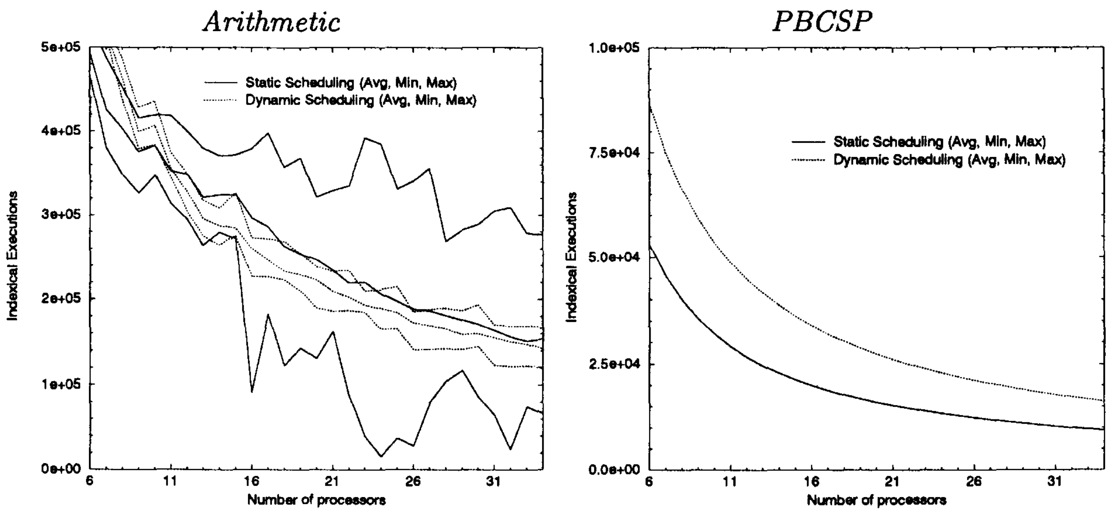

Fig. 7. Average, minimum and maximum number of executions of indexicals per processor.

Figure 7 shows the average, the minimum, and the maximum number of indexicals executed per processor, for the dynamic scheduling policy and the static scheduling policy with immediate broadcast. The difference between minimum and maximum indicates workload balance quality. It can be observed that, in the *Arithmetic* problem, the larger number of processors, the worse workload balance is. This fact limits the performance, since the execution time corresponds to the slower processor, because of serialisation between consecutive call to arc-consistency. The dynamic scheduling policy exhibits a better workload balance. Nevertheless, the speedup for this model is limited by the need of synchronisation points. For the *PBCSP* benchmark, the minimum and maximum curves do not differ in the static neither in the dynamic policy, indicating a high quality workload balance. Nevertheless, it can be expected that *PBCSP* benchmark will also reach a saturation point for a larger number of processors.

6.3 Scaleup

It is important to know how the performance of the parallel system depends on the characteristics of the problem. The *PBCSP* benchmark, as a generic parametrizable constraint satisfaction problem, offers the opportunity to study what characteristics are desirable in a problem in order to achieve a high performance when executed in parallel.

Table 2. Benchmarks characteristics of figure 8(a), scaleup vs. number of variables.

No. of variables	**25**	**50**	**100**	**200**
No. of Constraints	228	910	3,713	14,932
No. of Indexicals	456	1,820	7,426	29,864
No. of Calls to Consistency	63	61	65	66
Seq. Exec. Time (s.)	0.29	1.19	5.25	22.46
No. of ind. exec.	18,192	71,218	318,552	1,311,061
Avg. time per call (ms.)	4.6	19.5	80.8	340.3

Table 3. Benchmarks characteristics of figure 8(b), scaleup vs. density.

Density	**0.25**	**0.50**	**0.75**	**1.00**
No. of Constraints	1,216	2,464	3,713	4,950
No. of Indexicals	2,432	4,928	7,426	9,900
No. of Calls to Consistency	64	64	65	62
Seq. Exec. Time (s.)	1.78	3.40	5.25	6.59
No. of ind. exec.	105,542	205,400	318,552	400,430
Avg. time per call (ms.)	27.8	53.1	80.8	106.3

The size of a *PBCSP* mainly depends on the number of variables and the density of the constraint graph. Figure 8(a) shows the speedup versus the number of processors, for four different numbers of variables, fixing density to 0.75, tightness to 0.85, and domain size to 20. Figure 8(b) shows the speedup versus the number of processors, for different densities, fixing the number of variables to 100, tightness to 0.85, and domain size to 20. Tables 2 and 3 summarises relevant data about the problem instances used to plot the curves. All instances were run searching for all solutions (40). Both charts indicate that the larger the problem is, the higher speedup is obtained. This fact indicates the suitability of the system for large problems, provided a uniform constraint graph, which is a much desirable property in order to solve the real scale combinatorial problems which constraint programming aims to tackle.

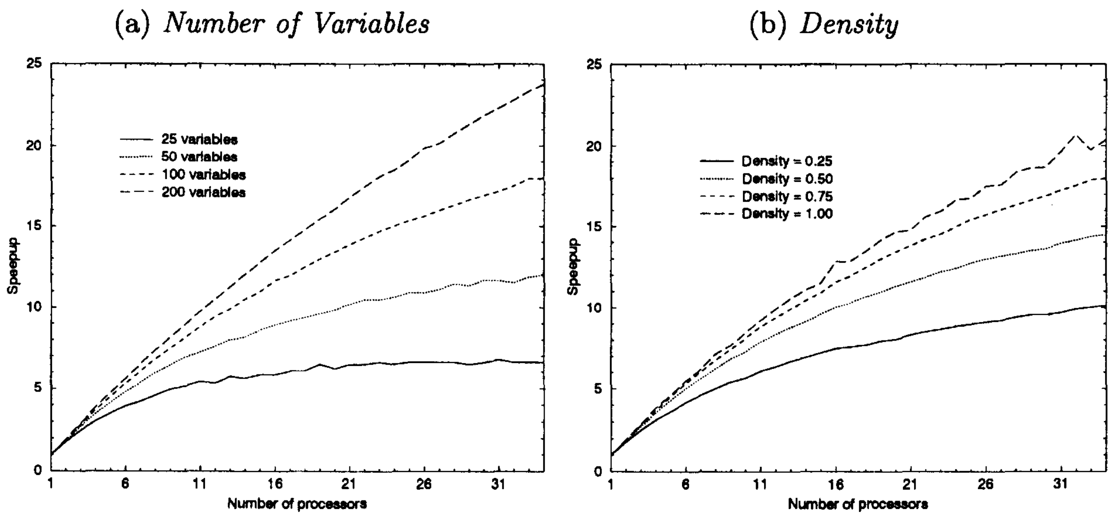

Fig. 8. Scaleup with the number of variables and the density of the problem.

7 Conclusions

We have developed and evaluated parallelisation models of an arc-consistency algorithm for constraint satisfaction problems over finite domains. These models have been implemented on a CRAY T3E, a distributed shared memory MIMD multiprocessor, and empirical data is reported for several benchmarks.

Two different techniques for scheduling the execution of constraints, dynamic and static, have been tested. The dynamic model has shown poor speedups, particularly when compared with those obtained with the static model, therefore we have focused our work on the static scheduling policy.

A number of topics affecting performance have been investigated in order to tune the static scheduling model. The way constraints are distributed among processors, and the frequency of updating shared variables, are determining factors for the performance of the model. The study of the distribution of constraints among processors has shown that a strongly connected partitioning (high number of shared variables) is worse than a partition based on an estimation of the run-time workload balance. Tests on broadcast frequency revealed the convenience of an immediate broadcast.

The speedup obtained is nearly linear for *PBCSP* benchmark, whereas for the rest of them it stops increasing from a problem dependent number of processors. This difference is mainly due to the more uniform constraint graph and larger granularity of the *PBCSP* benchmark, which leads to a better workload balance.

Anyway, the *PBCSP* benchmark would also reach a saturation point for a larger number of processors.

In order to study how the performance of the parallel system depends on the characteristics of the constraint satisfaction problem to solve, the parametrizable synthetic benchmark has been tested for different sets of parameters. Results show that the system is better suited for large scale problems with a dense constraint graph.

References

1. Bessiere, D.: Arc-consistency and arc-consistency again. Artificial Intelligence Journal 65 (1994) 179-190.
2. Baudot, B., Deville, Y.: Analysis of Distributed Arc-Consistency Algorithms. Tech. Rep. 97-07. Uni. of Louvain, Belgium (1997).
3. Carlson, B.: Compiling and Executing Finite Domain Constraints. PhD Thesis, Computer Science Department, Uppsala Univ. (1995).
4. Codognet, P., Diaz, D.: A Simple and Efficient Boolean Constraint Solver for Constraint Logic Programming. Journal of Automated Reasoning. 17,1 (1996) 97-128.
5. Dincbas, M., Van Henteryck, P., Simmons H., Aggoun, A.: The Constraint Programming Language CHIP Proceedings of the 2nd International Conference on Fifth Generation Computer Systems. (1988), 249-264.
6. Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving Large Combinatorial Problems in Logic Programming. Journal of Logic Programming 8 (1990).
7. ILOG: ILOG Solver 3.2 User's Manual (1996).
8. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artificial Intelligence 45 (1990) 275-286.
9. Kasif, S., Delcher, A.L.: Local Consistency in Parallel Constraint-Satisfaction Networks. Artificial Intelligence 69 (1994) 307-327.
10. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence 28 (1977) 225-233.
11. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence 28 (1996) 225-233.
12. Nguyen, T., Deville, Y.: A Distributed Arc-Consistency Algorithm. Science of Computer Programming, 30 (1998) 227-250.
13. Ruiz-Andino, A., Araujo, L., Ruz, J.: Parallel constraint satisfaction and optimisation. The PCSO system. Technical Report 71.98. Department of Computer Science. Universidad Complutense de Madrid (1998).
14. Saraswat, V.A.: Concurrent Constraint Programming Languages. MIT Press (1993).
15. Van Hentenryck P., Deville, Y., Teng C.M.: A generic Arc-consistency Algorithm and its Specialisations. Artificial Intelligence 57 (1992) 291-321.
16. Hentenryck, P.V., Saraswat, V.A., Deville, Y.: Constraint Logic Programming over Finite Domains: the Design, Implementation and Applications of cc(FD). Tech. Rep., Computer Science Dept., Brown University (1992).
17. Wallace, M.: Constraints in Planing, Scheduling and Placement Problems. Constraint Programming, Springer-Verlag (1994).
18. Waltz, D.: Generating semantic descriptions for drawings of scenes with shadows. Technical Report AI271, MIT, Cambridge, MA. (1972).

19. Yokoo, M.: Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. Principles and Practice of Constraint Programming (1995) 88-102.
20. Zhang, Y., Mackworth, A.K.: Parallel and Distributed Finite Constraint Satisfaction: Complexity, Algorithms and Experiments. Parallel Processing for Artificial Intelligence. Elsevier Science. (1993).

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Gopalan Nadathur
The University of Chicago, Department of Computer Science
1100 East 58th Street, Chicago, IL 60637, USA
E-mail: gopalan@cs.uchicago.edu

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Principles and practice of declarative programming : proceedings / International Conference PPDP '99, Paris, France, September 29 - October 1, 1999. Gopalan Nadathur (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1702)
ISBN 3-540-66540-4

CR Subject Classification (1998): D.3, D.1, F.3

ISSN 0302-9743
ISBN 3-540-66540-4 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN: 10704567 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper