Parallel Arc-Consistency for Functional Constraints

A. Ruiz-Andino †, L. Araujo †, F. Sáenz ‡, J. Ruz ‡
† Department of Computer Science
‡ Department of Computer Architecture
University Complutense of Madrid
arai@eucmax.sim.ucm.es, lurdes@eucmos.sim.ucm.es
fernan@eucmax.sim.ucm.es, jjruz@eucmax.sim.ucm.es
http://mozart.sip.ucm.es/ alvaro

Abstract

We present in this paper a parallel execution model of arc-consistency for Constraint Satisfaction Problems (CSP), implemented on a scaleable MIMD distributed memory machine. We have adopted the *indexical scheme*, an adequate approach to arc-consistency for functional constraints. The CSP is partitioned into N partitions, which are executed in parallel on N processors. Each processor applies sequential arc-consistency to its subset of constraints, updating remote domains of variables shared by any other processor. The parallel algorithm we propose is embedded in the constraint solver PCSOS (Parallel Constraint Satisfaction and Optimisation System). PCSOS invokes the parallel arc-consistency algorithm when performing labelling to obtain the solutions of a CSP. PCSOS is written in C, and it has been developed and tested on a CRAY T3E distributed memory multiprocessor with up to twenty-six processors. Results on speedup and behaviour of the system are reported for the search of the first solution of different CSPs.

1 Introduction

Constraint Programming over finite domains (CP(FD)) has been used for specifying and solving complex constraint satisfaction and optimisation problems, as resource allocation, scheduling and hardware design [6, 17]. Thanks to their combination of constraint solving and declarative style, Constraint Programming languages are able to find solutions to combinatorial problems comparable in efficiency to procedural languages, and yet requiring a much shorter development time. Finite domain Constraint Satisfaction Problems (CSP) usually describe NP-complete search problems. Much effort has been done to develop techniques in order to find solutions in polynomial time. 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 are called arc-consistency algorithms. Waltz [18] proposed the first arc-consistency algorithm, and several improved versions are described in the literature [11, 16, 15]. Consistency-based constraint solvers eliminate inconsistent values from the solution space, reducing the size of the search space both before and while searching. Most CP(FD) languages are based on a constraint solver which integrates a search procedure with an arc-consistency technique.

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 $c \equiv (V_c, R_c) \in \mathcal{C}$, $V_c \equiv \{v_1, \cdots, v_n\}$, is arc-consistent with respect to domains $\{d_1, \cdots, d_n\}$ iff for all $v_i \in V_c$, for all $a \in d_i$, there exists a tuple $(b_1, \cdots, b_{i-1}, a, b_{i+1}, \cdots, b_n) \in R_c$, 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 $\mathcal{D}$.

In this paper, we will focus on the exploitation of parallelism available on arc-consistency propagation. The parallel algorithm we propose is the core of the system PCSOS [13] (Parallel Constraint Satisfaction and Optimisation System). Besides arc-consistency parallelism, PCSOS may also exploit parallelism in search and optimisation procedures.

Several parallel processing methods for solving CSPs have been proposed. In [7, 8] authors describe logic gate level arc-consistency parallel algorithms, extending similar approaches of [14] and [4]. In [5] two hardware-level parallel arc-consistency algorithms are proposed. The first one is expressed as a massively parallel digital circuit requiring a large number of very simple processing elements. The second one is more suited to be implemented on a highly parallel SIMD machine. 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 [9]. In [10], it is concluded that parallel complexity of constraint networks is critically dependent on subtle properties of the network that do not influence its sequential complexity. They propose massively parallel processing of arc-consistency with also very simple processing elements.

Nguyen, Deville and Baudot proposed in [2, 12] a similar approach to ours, but for binary CSPs, discussing distributed versions of AC-3, AC-4, and AC-6. Our work is focused on n-ary functional constraints (AC-5), and we report empirical data obtained running the parallel arc-consistency algorithm on a CRAY T3E distributed memory multiprocessor, performing a search for the first solution to CSPs. The parallel execution mode is based on the partition of the CSP into N sub-CSPs, which are executed in parallel on N processors. Each processor applies sequential arc-consistency to its subset of constraints, updating remote domains of variables shared by any other processor. We also discuss the main issues affecting the performance of the model, like the criteria to distribute constraints are among processors, and the frequency of updating shared variables.

The rest of the paper is organised as follows. Next section describes the sequential constraint arc-consistency algorithm, based on the *indexical* scheme. Section 3 presents the model developed to parallelise constraint arc-consistency, the parallel algorithm, and a number of issues affecting performance. Section 4 presents and discusses the experimental results. Finally, conclusions are drawn in section 5.

2 Constraint Propagation

Our starting point is a sequential constraint propagation algorithm which implements arc-consistency using the *indexical scheme* [3]. In this scheme, a constraint is translated into a set of reactive functional expressions, called *indexicals*, which maintain arc-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 $V_c \equiv \{v_1, \cdots, v_n\}$, is translated into a set of n indexicals $\{I_i \equiv v_i \text{ in } E_i(V_c - \{v_i\})\}_{1 \leq i \leq n}$. Each indexical I_i writes variable v_i and depends on the remaining $n-1$ 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.

The set of finite domains which 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 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 $v \cap Eval(E(V))$, where $Eval(E(V))$ denotes the evaluation of $E(V)$ with the current domains of the set of variables V in the store.

For example, the arithmetic constraint $v_1 = v_2 + 4$ over finite integer domains is translated into two indexicals:

$I_1 \equiv v_1 \text{ in } min(v_2) + 4 \text{ to } max(v_2) + 4;$

$I_2 \equiv v_2 \text{ in } min(v_1) - 4 \text{ to } max(v_1) - 4.$

Whenever the lower or upper bound of v_2 ($min(v_2), max(v_2)$, resp.) changes, indexical I_1 removes inconsistent values from v_1's domain. Indexical I_2 behaves similarly. For instance, let $d_1 = d_2 = \{1, \cdots, 10\}$ be the initial domains for v_1 and v_2. I_1 prunes v_1's domain to $\{5, \cdots, 10\}$ and I_2 prunes v_2's domain to $\{1, \cdots, 6\}$. If some other indexical writing v_2 prunes d_2 to $\{4, 5\}$, then I_1 will prune d_1 to $\{8, 9\}$. Moreover, this modification will be *propagated* to other domains because of the execution of those indexicals which depend on v_1.

```
function Arc-Consistent-CSP(⟨VarSet, DomSet, ConstrSet⟩):Store
begin
  Store_Reset(Store, VarSet, DomSet);
  Queue_Reset(PropagationQueue);
  for each indexical I_i ∈ ConstrSet do
    if NOT Arc_Consistent(I_i,Store,PropagationQueue) then
      return FAILURE;
    end-if;
  end-for;
  while NOT Empty(PropagationQueue) do
    Queue_Pop(PropagationQueue, DomVar);
    for each indexical I_i which depends on DomVar do
      if NOT Arc_Consistent(I_i,Store,PropagationQueue) then
        return FAILURE;
      end-if;
    end-for;
  end-while;
  return Store;
end;
```

Figure 1: Arc consistency algorithm.

Figures 1 and 2 show the sequential arc-consistency propagation 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).

The 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 (lines 3 to 7). The main loop (lines 8 to 15) iterates until either the propagation queue is empty, or inconsistency is detected. In each iteration, a variable is unqueued and those indexicals which depend on it are executed. Termination, correctness, complexity, and properties of the algorithm have been studied extensively in the literature [16]. Correctness is independent of the order of reexecution of indexicals, which constitutes the basis for the correctness of the parallel version of the algorithm.

```
function Arc_Consistent( 'DomVar in E()',
                    Var Store, Var PropagationQueue ) :  BOOLEAN
begin
  NewDomain := Eval(E(), Store);
  return (Update(NewDomain,DomVar,Store,PropagationQueue)<>EMPTY);
end;

function Update( NewDomain, DomVar,
                    Var Store, Var PropagationQueue):  RESULT
begin
  NewDomain := NewDomain ∩ Store[DomVar];
  if Empty(NewDomain) then return EMPTY; end-if;
  if (NewDomain ⊂ Store[DomVar]) then
    Store[DomVar] := NewDomain;
    Queue_Push(DomVar, PropagationQueue);
    return PRUNED;
  end-if;
  return NOT_PRUNED;
end;
```

Figure 2: Store and propagation queue updating.

3 Parallel Constraint Propagation

Arc-consistency algorithms in general, and the indexical scheme in particular, have an 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 I_i may be executed only as the consequence of a previous execution of another indexical I_j that writes a variable which I_i depends on. An indexical is *ready* if any of its arguments have 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 the sequential version of the algorithm, the ready set is formed by those indexicals which depend on the variables in the propagation queue. The basis of our approach is to execute in parallel those indexicals in the ready set. Therefore, the parallel computational unit is the execution of an indexical.

The first decision to be made in order to define the parallel execution model is whether sharing or distributing the store and the set of indexicals. A straightforward implementation of the indexical execution model on a massively parallel processor would map a single indexical onto a single processor, sharing the store. Unfortunately, processor utilisation rate would be very low and communication volume would be very high. Since the current design is devoted to run on a distributed memory multiprocessor, our model is based on distributing the store and the set of indexicals, partitioning the CSP. Moreover, in order to reduce communication overhead, our model is based on a static partition of the CSP, instead of a dynamic one.

The set of constraints $\mathcal{C}$ is partitioned into n disjoint subsets of constraints, $\mathcal{C} = C_1 \cup \cdots \cup C_n$. 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 $I_j \in C_i$, the variable written by I_j, and those variables on which I_j depends on, constitute V_i ($\forall I_j \in C_i, I_j \equiv v_{I_j}$ in $E(V_{I_j}), V_i = \{v_{I_j}\} \cup V_{I_j}$.) 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 some variables in V_i. Since V_i are non-disjoint, some variables will be located at several processing elements. Therefore, each processing element P_i must broadcast any pruning on 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 detecting inconsistency.

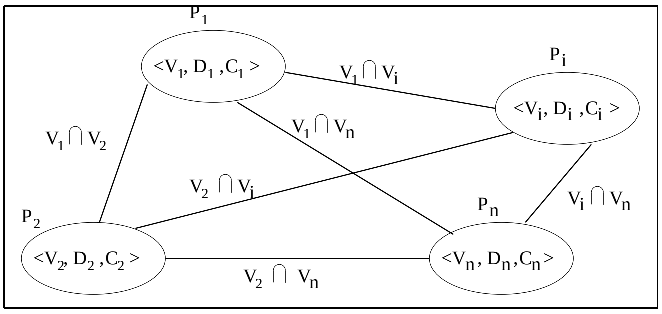

Figure 3: Partitioning the CSP. Sub-CSP $\langle V_i, D_i, C_i \rangle$ is assigned to processing element (PE) P_i. An edge between two PEs is labelled with the set of variables located at both PEs ($V_i \cap V_j$). Communication is needed to maintain the same domain for some of the variables in $V_i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

3.1 Parallel Algorithm

Figures 4 and 5 show the parallel algorithm. Like the sequential algorithm, initially every indexical assigned to the processor is executed (lines `5` to `10`). The main loop (lines `11` to `33`) is executed until either global fixed point (`GlobalFixedPoint`) or inconsistency (`Failure`) is detected. The latter can be caused by one of the following:

- an empty domain resulting from the execution of a local indexical (`Local_Arc_Consistent()`).

- an empty domain resulting from the intersection of the local domain of a variable with the domain received from another processor (`Remote_Arc_Consistent()`).
- inconsistency, detected at (and broadcasted from) another processor (`RemoteFailure`).

Each processor maintains a private propagation queue (`LocalPropQueue`). The inner loop (lines 12 to 20) 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 to a *distinguished* processor of this status (`Notify_Local_Fixed_Point()`), and it waits (lines 23 to 28) until one of the following occurs:

- 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 to the distinguished one, (`Notify_Active()`), and continues performing propagation.

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 2), the processor broadcasts a message (`Broadcast_Update()`) 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 broadcast the failure to the rest of processors (`Broadcast_Failure()`).

The algorithm terminates when every processor reaches a local fixed point and there are no messages pending on the communication network. The distinguished processor is the only 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 of reaching a local fixed point –along with the number of messages they have sent and received– (`Notify_Local_Fixed_Point()`), and of leaving it due to an incoming message (`Notify_Active()`). The distinguished processor keeps record of how many 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`).

The parallel arc-consistency algorithm will usually be repeatedly invoked from a sequential labelling procedure, or any other search procedure which needs to achieve arc-consistency. Therefore, synchronisation among all processors is needed both at the beginning and at the end of the algorithm. For this purpose, after the initialisation of the communication status variables (`ParState_Reset()`, line 1, Figure 4) and before starting propagation, a `barrier()` primitive is used. Another barrier (line 35) is needed if the algorithm finishes with failure; otherwise, the global fixed point detection implies a synchronisation among processors.

3.2 Tuning the parallel solver

There are a number of issues to be specified in the parallel execution model affecting performance. We have tried different settings, analysing how they affect work load, communications volume, and overall relevance. Mainly, there is a trade-off between two main factors affecting performance:

- Run-time size and balanced distribution of the ready set.

```
function Parallel-Arc-Consistent-CSP(
              ⟨VarSubSet, DomSubSet, ConstrSubSet⟩ ) :  Store
begin
  ParState_Reset();
  barrier();
  Store_Reset(Store, VarSubSet, DomSubSet);
  Queue_Reset(LocalPropQueue);
  for each indexical I_i ∈ ConstrSubSet do
    Failure := RemoteFailure OR
          NOT Local_Arc_Consistent(I_i,Store,LocalPropQueue) OR
          NOT Remote_Arc_Consistent(Store,LocalPropQueue);
    if Failure then break; end-if;
  end-for;
  while NOT Failure AND NOT GlobalFixedPoint do
    while NOT Failure AND NOT Empty(LocalPropQueue)) do
      Queue_Pop (LocalPropQueue, DomVar);
      for each indexical I_i which depends on DomVar do
        Failure := RemoteFailure OR
              NOT Local_Arc_Consistent(I_i,Store,LocalPropQueue) OR
              NOT Remote_Arc_Consistent(Store,LocalPropQueue);
        if Failure then break; end-if;
      end-for;
    end-while;
    if NOT Failure then
      Notify_Local_Fixed_Point(···);
      repeat
        Failure := RemoteFailure OR
              NOT Arc_Consistency_Msg(Store,LocalPropQueue);
        GlobalFixedPoint := Check_Global_Fixed_Point();
      until Failure OR GlobalFixedPoint OR
                NOT Empty(LocalPropQueue);
      if NOT Empty(LocalPropQueue) then
        Notify_Active();
      end-if;
    end-if;
  end-while;
  if Failure then
    barrier();
    return FAILURE;
  end-if;
  return Store;
end;
```

Figure 4: Parallel arc-consistency algorithm.

```
function Local_Arc_Consistent( 'DomVar in E()',
                          Var Store, Var LocalPropQueue ):  Boolean
begin
  NewDomain := Eval(E(), Store);
  switch (Update (NewDomain, DomVar, Store, LocalPropQueue))
    case EMPTY :
      Broadcast_Failure(RemoteFailure);
      return FALSE;
    case PRUNED:
      Broadcast_Update(DomVar, Store[DomVar]);
  end-switch;
  return TRUE;
end;

function Remote_Arc_Consistent( Var Store,
                          Var LocalPropQueue ) :  Boolean
begin
  while NOT Empty(MsgQueue) do
    Pop_Message(MsgQueue, DomVar, NewDomain);
    if (Update(NewDomain,DomVar,Store, LocalPropQueue) = EMPTY) then
      Broadcast_Failure(RemoteFailure);
      return FALSE;
    end-if;
  end-while;
  return TRUE;
end;
```

Figure 5: Parallel arc-consistency functions.

- Volume of communications.

A higher availability of updated domains results in a larger ready set. In addition, the more spread among processing elements indexicals are, the better balanced the ready set is. However, in order to achieve higher updated domains availability, more frequent communications are required. Similarly, spreading indexicals requires broadcasting updated information to more processing elements. Therefore, the advantage of working with a large and well balanced ready set is weakened by the effect of a higher communications volume.

The introduced parallel execution model offers the possibility to control the trade-off between communications and ready set quality through the definition of, at least, two issues: partitioning of the CSP, and frequency of broadcasting/processing domain updates.

3.2.1 Partitioning the CSP.

A key factor for the efficiency of the parallel algorithm has shown to be the way the set of constraints is partitioned. 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 between partitions.
- Static estimation of run-time ready set distribution.

Strength of connection between partitions.

The graph connectivity may be considered in order to partition the graph in either *strongly connected subgraphs*, or *highly disconnected subgraphs*.

In the former case, communications are minimised, but the ready set will be, in general, badly balanced. 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 to 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 constraint partitioning which balances the run-time ready set is expected to improve the performance, providing that communications do not increase. Since our approach is based on a static partitioning of the CSP among processors, balancing run-time ready set requires some kind of compile-time estimation.

Our approach to this estimation is to partition the constraint set in such a way that updating any variable amounts 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 work load for each processor and variable. Experimental run-time work load measures 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 that assigns constraints 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.

3.2.2 Frequency of broadcasting/processing domain updates

There is a trade-off between quick transmission of updated information, which increases the size of the ready set, and communications overhead, which decreases performance. We have tested two settings for the frequency of broadcasting domain updates:

- Immediate broadcast: domain updates are broadcasted as soon as they are locally produced.
- Fixed point broadcast: domain updates are locally queued and broadcasted whenever the local fix point is reached.

Similarly, the frequency of checking and processing incoming domains affects updated information availability. Checking for received updated domains each time an indexical is executed has shown to produce best results. In this way, processors are working with the best available updated information, while overhead due to checking for messages is negligible versus the execution time of an indexical.

4 Experimental Results

PCSOS is written in C, and it has been developed and tested on a CRAY T3E multiprocessor with up to twenty-six 400-MHz DEC Alpha processors, under UNICOS (UNIX) operating system. CRAY T3E is a scaleable parallel system which reduces latency providing sustained bandwidth. Processing elements in the T3E are connected by a bi-directional 3-D torus network achieving communication rates of 480 Mbytes/s. Parallel programming capabilities are extended through the Cray Shared Memory Library, which allows fast direct asynchronous access to remote memory.

Notification of local fixed point, activity, messages sent and received, and inconsistency detection has been implemented using the fast remote memory write feature of the CRAY T3E architecture. Queues of messages are used for receiving domain updates. Messages are broadcasted to queues also using the fast remote memory write feature. Variables used for global fixed point detection, located at the distinguished processor, must be accessed in mutual exclusion.

Reported results correspond to the search for the first solution, performing a first fail sequential labelling. Therefore, reported speedup is lower than speedup achieved in a single propagation cycle, since it comprises both phases, variable/value selection, executed sequentially, and constraint propagation, executed in parallel. This way to measure is more significant than measuring a single propagation cycle, since constraint propagation is usually embedded in some kind of a search procedure.

We have tested PCSOS on a set of benchmarks. Results obtained for two of them, *Equations* and *Suudoku* are reported. *Equations* is a synthetic benchmark. It is formed by sixteen blocks $\{B_1, \cdots, B_{16}\}$ of arithmetic equations. Each block contains fifteen equations 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 total number of indexicals is 1,626. Sequential search for the first solution performs 15,969 calls to parallel Arc-Consistent-CSP, 12,438 of them detecting inconsistency. Execution time with just one processor on the CRAY T3E is 16.2 sec. Sequential execution time on a SPARC workstation is 30.40 sec., versus 26.48 sec. obtained with Sicstus Prolog 3.6. *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. There are a total number of 28,193 indexicals. First solution is found after 72,196 calls to Arc-Consistent-CSP, 36,092 of them detecting inconsistency. Execution time with just one processor on the CRAY T3E is 152.1 sec. Sequential execution time on a SPARC workstation is 236.85 sec., versus 207.05 sec. obtained with Sicstus Prolog 3.6.

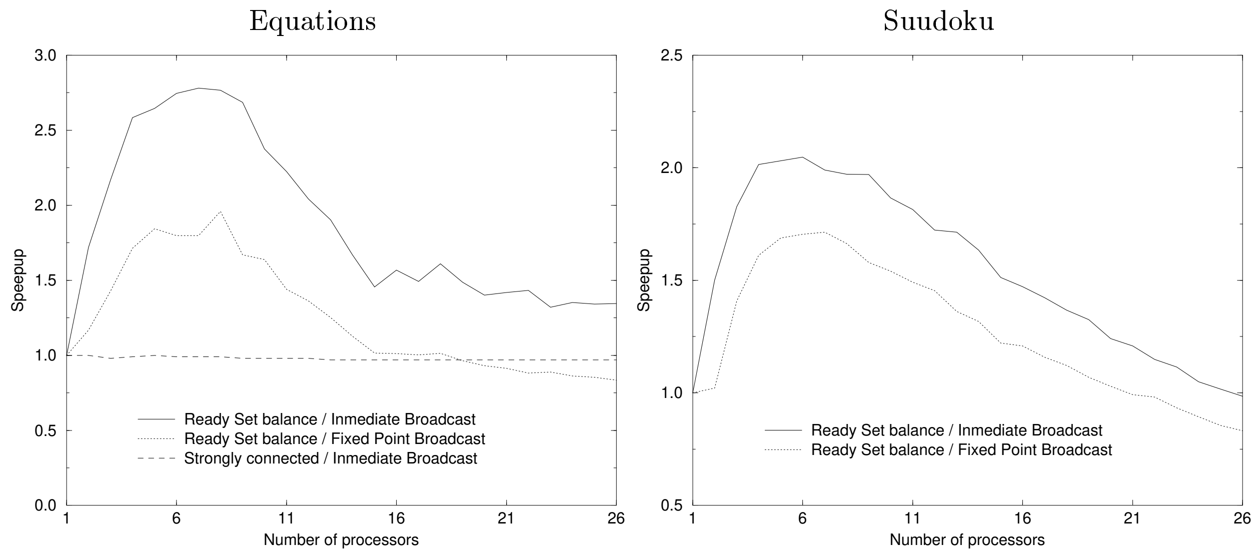

Figure 6: Speedup curves for selected benchmarks.

Charts in Figure 6 show, for each benchmark, the speedup vs. the number of processors. The ready set balance estimation was used, comparing broadcast frequency: immediate (solid line) vs. fixed point (dotted line). Chart for the *Equations* benchmark also shows the speedup obtained with a strongly connected graph partitioning (dashed line). This criteria 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.

It must be noticed that, for both benchmarks, speedup decreases as more processors are added beyond a problem dependent optimum number of processors. This behaviour has also been observed on the paralelisation of event-driven logic simulation [1] and other iterative algorithms. Figures 7, 8, 9 and 10 show the behaviour of two factors that give some hints for the decrease of the speedup.

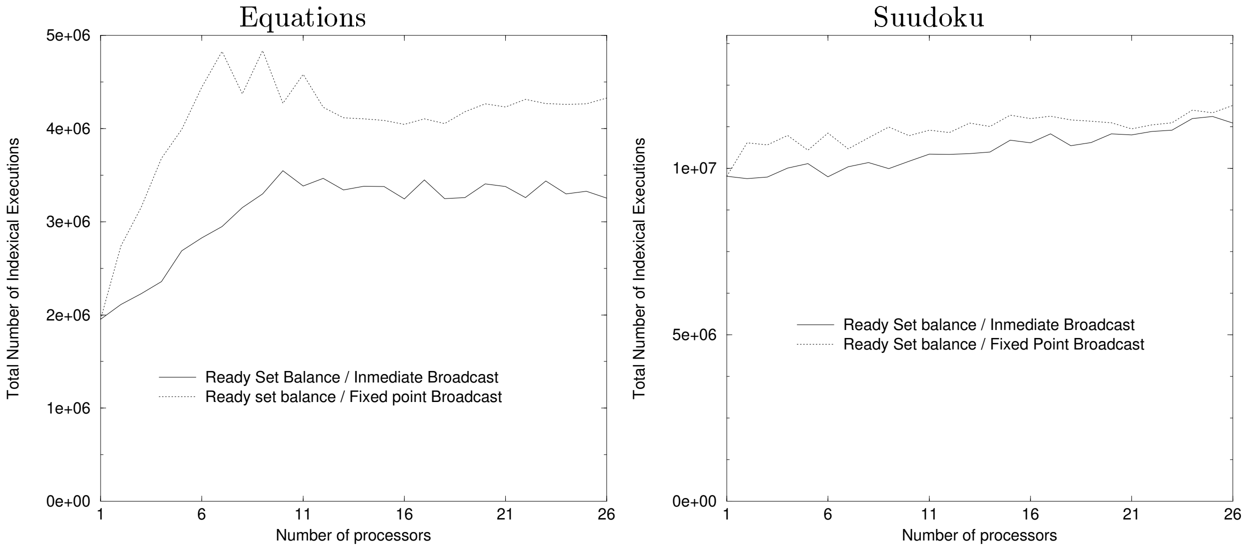

Figure 7: Total number of indexical executions

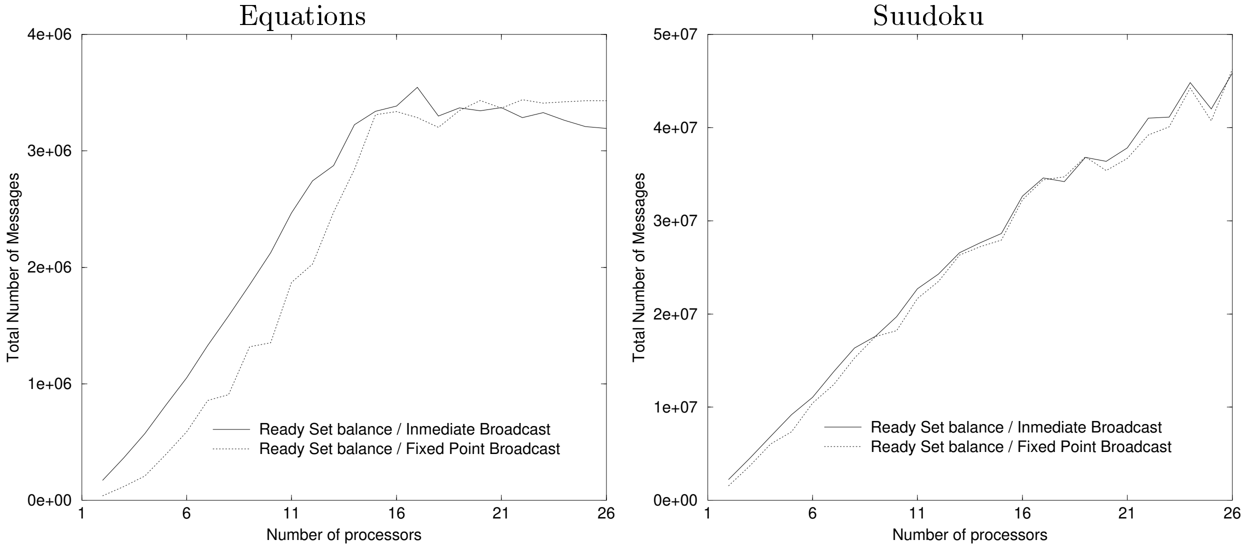

Figure 8: Total number of messages sent

Figure 7 shows the sum over all processors, over all propagation cycles, of the number of executed indexicals. It can be observed the increase of the total number of indexicals executions when running the problem in parallel, since some indexical executions work with less updated data. For both problems, broadcasting at the fixed point implies a higher number of indexical executions, but just a slightly lower number of messages (figure 8), thus explaining the lower speedup obtained.

Figure 8 shows the total number of sent messages. Results for the equations problem present a somewhat surprising behaviour: the total number of messages becomes stable or even decreases beyond a certain number of processors. This behaviour may be because variable domain changes are only sent to those processors which need them, and in this particular problem, as more processors are added, it is more likely that a change does not need to be sent to every processor.

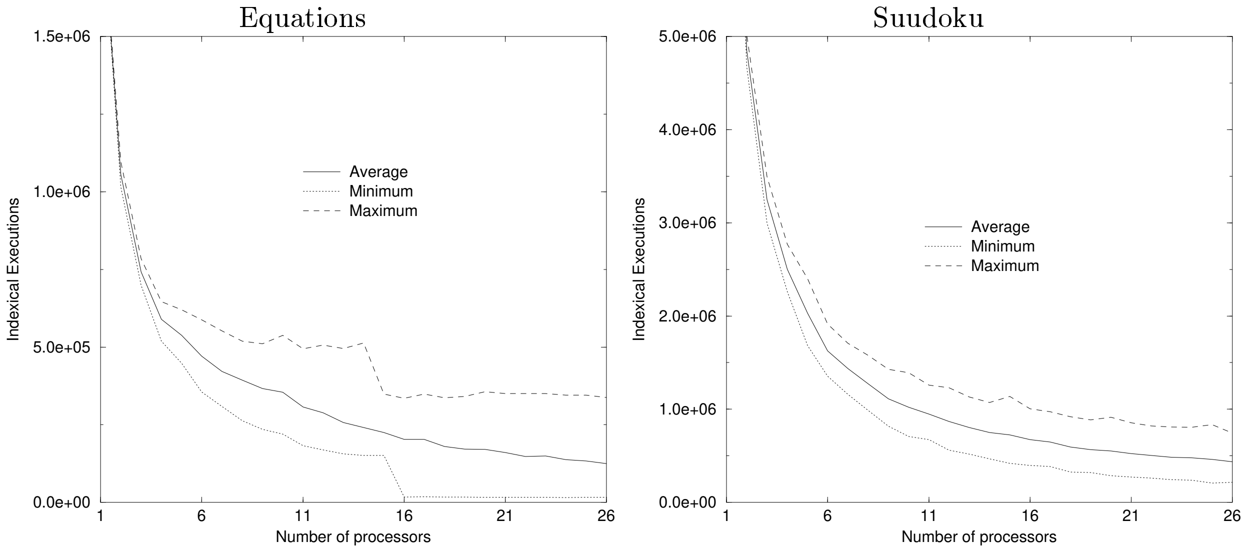

Figure 9: Indexical executions per processor (Ready Set balance / Immediate Broadcast)

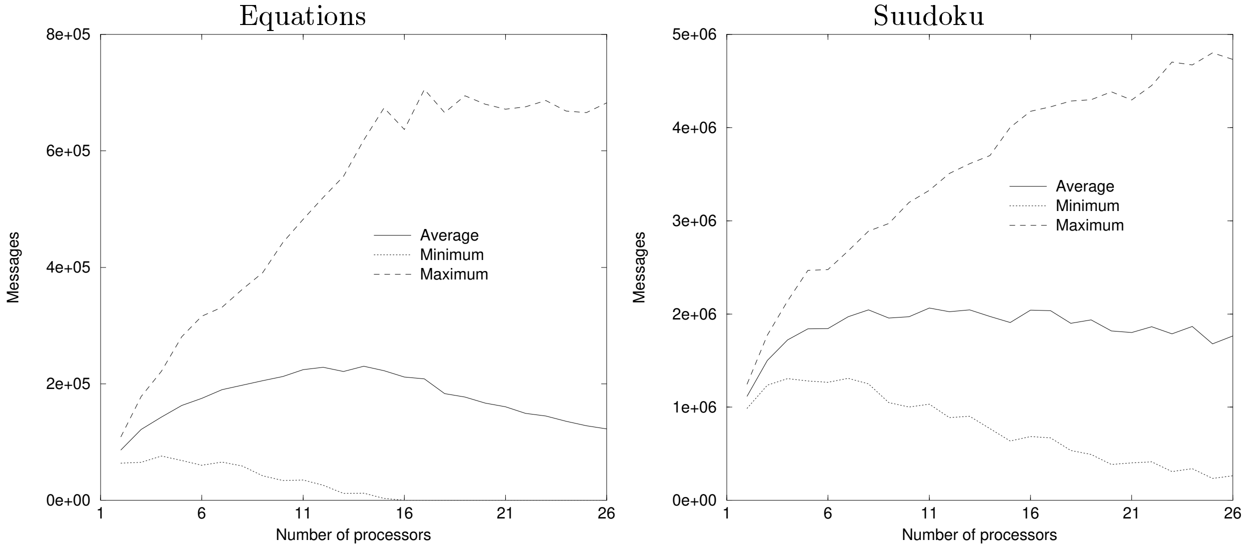

Figure 10: Messages per processor (Ready Set balance / Immediate Broadcast)

Figure 9 shows the average number of indexicals executed per processor (solid), and the sum, over all propagation cycles, of the minimum (long dashed) and of the maximum (dotted) number of indexicals executed per processor. Figure 10 compares the number of messages sent per processor. Both charts exhibit a large difference between minimum and maximum values, which increases as more processors are added. The difference between minimum and maximum indicates load work balance quality. The larger number of processors, the worse load work balance is. This behaviour limits the performance, since the execution time corresponds to the slower processor.

5 Conclusions

We have developed a parallel constraint solving system for functional constraints, based on the indexical scheme. This system has been implemented on a CRAY T3E, a distributed memory MIMD multiprocessor, and empirical data are reported for a set of benchmarks.

A number of issues affecting performance have been investigated in order to tune the 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, in general, worse than a distribution which balances the ready set. Tests on broadcast frequency revealed the convenience of an immediate update. Benchmarks considered so far exhibit a speedup between 2.0 and 3.0, though better results may be expected for larger problems.

Acknowledgements

This work has been funded by PRONTIC TIC95-0433.

References

[1] Agrawal, V.D.: Performance analysis of synchronized iterative algorithms on multiprocessors systems. IEEE Transactions on Parallel and Distributed Systems, vol. 3, No. 6 (1992)

[2] Baudot, B., Deville, Y.: Analysis of Distributed Arc-Consistency Algorithms. Tech. Rep. 97-07. Uni. of Louvain, Belgium (1997).

[3] Codognet, P., Diaz, D.: A Simple and Efficient Boolean Constraint Solver for Constraint Logic Programming. Journal of Automated Reasonig. 17,1 (1996) 97-128.

[4] Collin, Z., Dechter, R., Katz, S.: On the feasibility of distributed constraint satisfaction. Proceedings of the International Joint Conference on AI (IJCAI) (1991) 318-324.

[5] Cooper, P.R., Swain, M.J.: Arc consistency: parallelism and domain dependence. Artificial Intelligence 58 (1992) 207-235.

[6] Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving Large Combinatorial Problems in Logic Programming. Journal of Logic Programming 8 (1990).

[7] Guesguen, H.W.: Connectionist networks for constraint satisfaction. AAAI Spring Symposium on Constraint-based Reasoning (1991) 182-190.

[8] Guesguen, H.W., Hertzberg, J.: A perspective of constraint-based reasoning. Lecture Notes on Artificial Intelligence, Springer-Verlag (1992).

[9] Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artificial Intelligence 45 (1990) 275-286.

[10] Kasif, S., Delcher, A.L.: Local Consistency in Parallel Constraint-Satisfaction Networks. Artificial Intelligence 69 (1994) 307-327.

[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] Swain, M.J., Cooper, P.R.: Parallel hardware for constraint satisfaction. Proceedings of the National Conference on Artificial Intelligence (AAAI) (1988) 682-686.

[15] Tsang, E.: Foundations of constraint satisfaction. Academic Press (1993).

[16] Van Hentenryck P., Deville, Y., Teng C.M.: A generic Arc-consistency Algorithm and its Specialisations. Artificial Intelligence 57 (1992) 291-321.

[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 Algorithms for Constraint Networks. Technical Report 91-6, Dept. of Computer Science, University of British Columbia, Vancouver, Canada (1991).