Optimistic Parallel Discrete Relaxation*

Kinson Ho and Paul N. Hilfinger
Computer Science Division
University of California at Berkeley, U. S. A.
Hans W. Guesgen
Computer Science Department
University of Auckland, New Zealand
{ho,hilfingr}@cs.berkeley.edu, hans@cs.auckland.ac.nz

Abstract

Discrete relaxation is frequently used to compute the fixed point of a discrete system $X = f(X)$, where / is monotonic with respect to some partial order $\leq$, Given an appropriate initial value for X, discrete relaxation repeats the assignment $X \leftarrow f(X)$ until a fixed point for / is found. Monotonicity of / with respect to $\leq$ is a sufficient (but in general not necessary) condition for iterative, hill-climbing techniques such as discrete relaxation to find the fixed point of /.

In this paper we introduce *monotonic asynchronous iteration* as a novel way of implementing parallel discrete relaxation in problem domains for which monotonicity is a necessary condition. This is an optimistic technique that maintains monotonicity without limiting concurrency, resulting in good parallel performance. We illustrate this technique with the parallel implementation of a constraint satisfaction system that computes globally consistent solutions, and present performance numbers for experiments on a shared-memory implementation. The performance numbers show that it is indeed possible to obtain a reasonable speedup when parallelizing global constraint satisfaction. We believe that monotonic asynchronous iteration is applicable to parallel discrete relaxation in general.

1 Introduction

Discrete relaxation is frequently used to compute the fixed point of a discrete system $X = f(X)$, where / is monotonic with respect to some partial order <. Given an appropriate initial value for *X,* discrete relaxation repeats the assignment $X \leftarrow f(X)$ until a fixed point for / is found. Monotonicity of / with respect to < is a sufficient (but in general not necessary) condition for iterative, hill-climbing techniques such as discrete relaxation to find the fixed point of / [Parker, 1987].

Discrete relaxation is widely used in the solution of constraint satisfaction problems (CSPs), and many parallel implementations of discrete relaxation for CSPs have been reported [Kasif and Rosenfeld, 1983; Rosenfeld *et a*/., 1976]. These attempts have all focused on CSP solvers that compute locally consistent (arc consistent) solutions, which are relatively straightforward to parallelize as the computations are inherently monotonic. On the other hand, discrete relaxation algorithms used in CSP solvers that compute globally consistent solutions are very difficult to parallelize because for this class of problems, monotonicity is a necessary correctness condition that is not automatically satisfied. The need to maintain monotonicity (for correctness) often limits the amount of concurrency available in a parallel implementation, degrading the performance significantly.

In this paper we introduce *monotonic asynchronous iteration* as a novel way of implementing parallel discrete relaxation in problem domains for which monotonicity is a necessary condition. This is an optimistic technique that maintains monotonicity without limiting concurrency, resulting in good parallel performance. We illustrate this technique with the parallel implementation of CONSAT [Guesgen, 1989], a constraint satisfaction system that computes globally consistent solutions, and describes an experiment on a shared-memory implementation. The performance numbers show that it is indeed possible to obtain a reasonable speedup when parallelizing global constraint satisfaction, and thus proving that Kasif's conjecture is correct [1990]. We believe that monotonic asynchronous iteration is applicable to parallel discrete relaxation in general.

2 Discrete Relaxation

Consider the problem of finding the fixed point of a discrete system $X = f(X)$. For our purposes, the system has the following properties:

*Ho and Hilfinger are supported by NSF Grant CCR-84-51213. Guesgen performed part of this work while at the German National Research Center for Computer Science (GMD). in St. Augustin, Germany, and the International Computer Science Institute in Berkekey, California. At the GMD he was supported by the German Federal Ministry for Research and Technology (BMFT) within the joint projects TEX-B (grant ITW8506D) and TASSO (grant ITW8900A7).

- f: $D \times D$, for some domain D, is a relation. We shall notate it as a non-deterministic function, writing, for example, $Y \leftarrow f(X)$ to mean "set Y to some value standing in relation f with X." An equation $Y = f(X)$ means that Y is a value satisfying the relation.
- Values in D are structured, so that for any $X \in D$, $X = \langle X_1, \ldots, X_n \rangle$, where $X_i \in D_i$ and $D = D_1 \times \cdots \times D_n$.
- $I = \{1, \ldots, n\}$ is the *index set*.
- There exist functions f_c ($c \subseteq I$) such that a solution of $X = f(X)$ is a solution of a system of equations $X = f_c(X)$ for all $c \subseteq I$, where each f_c: $D \rightarrow D$ is deterministic. Each relaxation step $X \leftarrow f(X)$ is equivalent to $X \leftarrow f_c(X)$ for some $c \subseteq I$. For $i \in I$ and $c \subseteq I$, $f_c(X) = \langle Y_1, \ldots, Y_n \rangle$, where

$$Y_i = \begin{cases} X_i & \text{if } i \notin c \\ {}_if_c(X) & \text{if } i \in c \end{cases}$$

  The new value of any such component i ($i \in c$) is given by ${}_if_c(X)$ (${}_if_c : D \rightarrow D_i$). $f_c(X)$ only differs from X at the indices in c.
- f is monotonic with respect to some partial order $\leq$, i.e., $X \leq f(X)$ for any $X \in D$. Consequently, $X \leq f_c(X)$ for any $c \subseteq I$, and $X_i \leq_i {}_if_c(X)$ for $i \in I$. For $X, Y \in D$, where $X = \langle X_1, \ldots, X_n \rangle$ and $Y = \langle Y_1, \ldots, Y_n \rangle$, $X \leq Y$ iff $X_i \leq_i Y_i$ for all $i \in I$.

Assumption We assume that domain-specific knowledge has been used to ensure that f has a fixed point that can be computed in a finite number of steps using discrete relaxation. The precise conditions for this may be found in the lattice theory literature [Stoy, 1977; Schmidt, 1986].

Notation Let X^j denote the value of X computed in the jth iteration of a discrete relaxation, and let X^0 be the initial value of X.

Sequential Iterations Given an appropriate X^0, discrete relaxation repeats the assignment $X \leftarrow f(X)$ until a fixed point for f is found. In a sequential implementation of discrete relaxation, $X \leftarrow f(X)$ becomes $X^j = f(X^{j-1})$, or $X^j = f_c(X^{j-1})$ for some $c \subseteq I$.

For $i \in I$ and $j = 1, 2, \ldots$, a *sequential iteration* has the following form:

$$X_i^j = \begin{cases} X_i^{j-1} & \text{if } i \notin c \\ {}_if_c(X^{j-1}) & \text{if } i \in c \end{cases}$$

where $c \subseteq I$ is not fixed for successive values of j. X^j only depends on X^{j-1}, and not on other (older) values of X.

Asynchronous Iterations We would like to compute the fixed point of f in parallel by computing f_c for different values of c ($c \subseteq I$) concurrently. For example, f_c and f_d ($c, d \subseteq I$) should be allowed to proceed in parallel if c and d do not intersect. This is not possible under sequential iterations, because X^j is computed using the value of X from the most recent iteration (X^{j-1}) only, serializing the computations for f_c and f_d. Intuitively, f_c and f_d may be computed in parallel if the restriction that X^j is computed using the value from the most recent iteration (X^{j-1}) is relaxed so that values from some *relatively recent* iterations may be used instead.

Following Baudet [1978], we define *asynchronous iterations* by removing the restriction imposed by sequential iterations. An asynchronous iteration $X^j = f'_c(X^{src(j,1\ldots n)}, X^{j-1})$ is defined as follows:

$$X_i^j = \begin{cases} X_i^{j-1} & \text{if } i \notin c \\ {}_if_c(X^{src(j,1\ldots n)}) & \text{if } i \in c \end{cases}$$

where $X^{src(j,1\ldots n)} = \langle X_1^{src(j,1)}, \ldots, X_n^{src(j,n)} \rangle$, and $src(j,k) \in \{0, \ldots, j-1\}$. The function $src(j,k)$ determines what past value of X_k is used in the computation. For any $k \in I$, $X_k^{src(j,k)}$ is the value of X_k used by ${}_if_c$ to compute X_i^j. We make two observations:

- $src(j,k)$ is not a function of i. For any value of $k \in I$, ${}_if_c$ ($i \in c$) uses the value of X_k computed in iteration $src(j,k)$, $X_k^{src(j,k)}$.
- For a given value of j, $src(j,k)$ is a function of k. Consequently, the value of X_k ($k \in I$) used to compute X_i^j may have been computed in different iterations. This freedom reduces the amount of synchronization required in a parallel implementation of asynchronous iterations significantly.

For example,

$$\begin{aligned} X_2^7 &= {}_2f_{\{2,3\}}(\langle X_1^{src(7,1)}, X_2^{src(7,2)}, X_3^{src(7,3)} \rangle) \\ &= {}_2f_{\{2,3\}}(\langle X_1^5, X_2^3, X_3^4 \rangle), \text{and} \\ X_3^7 &= {}_3f_{\{2,3\}}(\langle X_1^{src(7,1)}, X_2^{src(7,2)}, X_3^{src(7,3)} \rangle) \\ &= {}_3f_{\{2,3\}}(\langle X_1^5, X_2^3, X_3^4 \rangle) \end{aligned}$$

For sequential iterations $src(j,k) = j - 1$.

The value of X^j computed by an asynchronous iteration

$$X^j = f'_c(X^{src(j,1\ldots n)}, X^{j-1})$$

depends on multiple previous states. The components of X not modified by f'_c come from the most recent state, X^{j-1}, while the components of X modified by f'_c come from $X^{src(j,1\ldots n)}$. As X^j is not computed using a single (consistent) state of X, f'_c is *not* monotonic in general, and so $X^{j-1} \not\leq f'_c(X^{src(j,1\ldots n)}, X^{j-1})$. Consequently, asynchronous iterations may lead to incorrect results for parallel discrete relaxations.

Monotonic Asynchronous Iterations Given an arbitrary discrete, monotonic and non-deterministic function f with functions f_c ($c \subseteq I$), it is relatively difficult to derive a condition such that the corresponding f'_c functions defined by an asynchronous iteration are monotonic. To use asynchronous iteration for parallel discrete relaxation, we apply an application-specific test for monotonicity to the result computed by f'_c such that if $X^{j-1} \leq f'_c(X^{src(j,1\ldots n)}, X^{j-1})$, then X^j is updated using this value of f'_c. Otherwise, f'_c is re-computed (using more recent values of X). We call this optimistic scheme *monotonic asynchronous iteration*:

$$X_i^j = \begin{cases} {}_if_c(X^{src(j,1\ldots n)}) & \text{if } i \in c \text{ and } \forall a \in c, \\ & X_a^{j-1} \leq {}_af_c(X^{src(j,1\ldots n)}) \\ X_i^{j-1} & \text{otherwise} \end{cases}$$

If f'_c and f'_d $(c, d \subseteq I)$ are computed concurrently, a *race* occurs if X is modified by f'_d after the values of $X^{src(j,1\ldots n)}$ used to compute ${}_if_c(X^{src(j,1\ldots n)})$ have been read, but before X is modified by f'_c. Monotonic asynchronous iteration *only* detects (and rejects) races that would violate monotonicity. Races that maintain monotonicity are allowed, so X^{j-1} and $X^{src(j,1\ldots n)}$ do not have to be equal. In fact, these *legal* races provide a source of concurrency not found in other parallel relaxation schemes, and lead to improved performance. Intuitively, f_c and f_d may be computed concurrently if they modify disjoint components of X, or if they modify common components of X in such a way that monotonicity between successive values of X is maintained.

We do not have a general way of deriving the application-specific test for monotonicity for a given system. Instead, we expect the programmer to provide this test by using high-level knowledge about the problem domain. In the context of CSP solvers that compute globally consistent solutions, this test reduces to a subset test between successive sets of value combinations that may be associated with the variables of a constraint network (see Section 3).

Parallel Implementation of Monotonic Asynchronous Iterations A step of a monotonic asynchronous iteration $X^j = f'_c(X^{src(j,1\ldots n)}, X^{j-1})$ with partial order $\leq$ may be implemented in parallel in the following way:

1. Read current value of X,
   $X^{src(j,1\ldots n)} = (X_1^{src(j,1)}, \ldots, X_n^{src(j,n)})$
   (X_is may come from different iterations.)
2. Compute ${}_if_c(X^{src(j,1\ldots n)})$ for all $i \in c$
3. Lock X
4. if $\forall\, i \in c$: $X_i^{j-1} \leq {}_if_c(X^{src(j,1\ldots n)})$
   then for each $i \in c$,
   update X:$X_i^j = {}_if_c(X^{src(j,1\ldots n)})$
   else schedule f'_c for re-execution
5. Unlock X

X is not locked while the ${}_if_c(X^{src(j,1\ldots n)})$ values are computed in step 2. If these computations are time-consuming, this implementation can lead to much better parallel performance than a naive alternative that locks X for the entire step of the monotonic asynchronous iteration. Nevertheless, the maximum speedup is bounded by $1 + (T_1 + T_2)/T_4$, where T_i is the time for step i. For example, if T_4 is 10% of $(T_1 + T_2)$, the speedup limit is 11.

Practical Considerations Monotonic asynchronous iteration does not specify how a function f'_c $(c \subseteq I)$ is chosen for execution in any iteration j, or whether f'_c and f'_d $(c, d \subseteq I)$ should be executed concurrently in a parallel implementation for optimal performance. There is considerable freedom in applying application-specific scheduling strategies for good performance.

3 Example: CONSAT

CONSAT is a system for the definition and satisfaction of constraints in arbitrary finite discrete domains. A constraint consists of a set of variables and a relation among the variables, and a constraint network is a set of constraints connected by common variables. A globally consistent solution of a constraint network is a tuple of values, one per variable of the network, that satisfies all the constraints of the network simultaneously. CONSAT is a global CSP solver whose constraint satisfaction technique is based on filtering [Waltz, 1972], i.e., on the successive deletion of inconsistent values from the set of potential values for the variables. Unlike traditional filtering algorithms, CONSAT uses values that are associated with some additional information (called tags) for maintaining interrelationships among values (of different variables). In the following we give an informal explanation of how CONSAT modifies local constraint propagation, a technique commonly used to compute locally consistent solutions, with tagging to compute the globally consistent solutions of a constraint network. As global constraint satisfaction problems over finite domains are generally NP-complete, the algorithm used by CONSAT is exponential in the worst case. A more formal treatment of CONSAT is given elsewhere [Guesgen, 1989].

Constraint Network Example Figure 1 defines constraint network A, which will be used in the examples throughout this paper. There are three variables (Va, Vb, Vc) and six constraints (C1-C6). For example, constraint C1 restricts the possible values of Va to R or Y, and constraint C4 restricts the values of (Va,Vb) to one of the combinations (R,G) or (R,B). Va and Vb are called the adjacent variables of constraint C4. (A constraint may be adjacent to more than two variables, i.e., we do not restrict ourselves to binary constraints.) The steps of CONSAT as it computes the global solutions of constraint network A are shown in Figure 2. Details of the algorithm used may be found elsewhere [Guesgen, 1989].

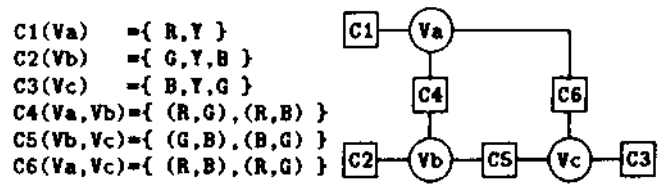

Figure 1: Constraint Network A: Variables are represented by circles and constraints by rectangles. An edge between a circle and a rectangle means that the corresponding variable belongs to the constraint represented by the rectangle.

Tagged Values For any variable of a constraint network, the potential values are associated with additional information called tags, and tagged values (of different variables) with the same full tag form a global solution of the constraint network. For a constraint network of

```
Initial:                                                    Step
    Va = Vb = Vc = UnConstrained                             0

C1: Va = { R(1,-,-,-,-,-), Y(2,-,-,-,-,-) }                  1

C2: Vb = { G(-,1,-,-,-,-), Y(-,2,-,-,-,-),                   2
           B(-,3,-,-,-,-) }

C3: Vc = { B(-,-,1,-,-,-), Y(-,-,2,-,-,-),                   3
           G(-,-,3,-,-,-) }

C4: Va = { R(1,1,-,1,-,-), R(1,3,-,2,-,-) }                  4
    Vb = { G(1,1,-,1,-,-), B(1,3,-,2,-,-) }

C5: Vb = { G(1,1,1,1,1,-), B(1,3,3,2,2,-) }                  5
    Vc = { B(1,1,1,1,1,-), G(1,3,3,2,2,-) }

C6: Va = { R(1,1,1,1,1,1), R(1,3,3,2,2,2) }                  6
    Vc = { B(1,1,1,1,1,1), G(1,3,3,2,2,2) }

C4: Va = { R(1,1,1,1,1,1), R(1,3,3,2,2,2) } *                7
    Vb = { G(1,1,1,1,1,1), B(1,3,3,2,2,2) }

C5: Vb = { G(1,1,1,1,1,1), B(1,3,3,2,2,2) } *                8
    Vc = { B(1,1,1,1,1,1), G(1,3,3,2,2,2) } *

C1: Va = { R(1,1,1,1,1,1), R(1,3,3,2,2,2) } *                9

C2: Vb = { G(1,1,1,1,1,1), B(1,3,3,2,2,2) } *               10

C3: Vc = { B(1,1,1,1,1,1), G(1,3,3,2,2,2) } *               11

Solution(Va, Vb, Vc):
 (R(1,1,1,1,1,1), G(1,1,1,1,1,1), B(1,1,1,1,1,1))
 (R(1,3,3,2,2,2), B(1,3,3,2,2,2), G(1,3,3,2,2,2))
```

Figure 2: Activations of constraint network A: Each constraint being activated (leftmost column) is shown with the feasible sets of its adjacent variables *after* its activation. An "*" at the end of a variable means that it has *not* been changed in the current activation. The feasible set of any variable non-adjacent to the current constraint can be found by searching backwards from the current activation. (Componentwise unification of full tags is the key idea behind the algorithm.)

```
  (Va,Vb,Vc) = (R,G,B) with full tag (1,1,1,1,1,1)
C1(Va)    = { R,Y }           1  Va      = R
C2(Vb)    = { G,Y,B }         1  Vb      = G
C3(Vc)    = { B,Y,G }         1  Vc      = B
C4(Va,Vb) = { (R,G), (R,B) }  1  (Va,Vb) = (R,G)
C5(Vb,Vc) = { (G,B), (B,G) }  1  (Vb,Vc) = (G,B)
C6(Va,Vc) = { (R,B), (R,G) }  1  (Va,Vc) = (R,B)

  (Va,Vb,Vc) = (R,B,G) with full tag (1,3,3,2,2,2)
C1(Va)    = { R,Y }           1  Va      = R
C2(Vb)    = { G,Y,B }         3  Vb      = B
C3(Vc)    = { B,Y,G }         3  Vc      = G
C4(Va,Vb) = { (R,G), (R,B) }  2  (Va,Vb) = (R,B)
C5(Vb,Vc) = { (G,B), (B,G) }  2  (Vb,Vc) = (B,G)
C6(Va,Vc) = { (R,B), (R,G) }  2  (Va,Vc) = (R,G)
```

Figure 3: Globally Consistent Solutions of Constraint Network A

m constraints, each full tag is an m-tuple, where the rth component (the rth *subtag)* indicates the tuple in the rth constraint that is part of the global solution. (For example, in constraint C4 of Figure 1 the tuple (R,B) corresponds to the subtag 2.) The special symbol "-" is the wildcard character for subtags and indicates that a tuple for the corresponding constraint has not been chosen. A subtag with a tuple number is said to be *determined,* while one with the special symbol "-" is said to be *undetermined.* As an example, consider the global solutions of constraint network A shown in Figure 2. The two solutions (Va,Vb,Vc) are (R,G,B) with full tag (1,1,1,1,1,1) and (R,B,G) with full tag (1,3,3,2,2,2). An intuitive explanation of the use of tags for constraint network A is given in Figure 3.

Filtering of Tagged Values The *feasible set* of a variable contains its current set of potential (tagged) values, and is initialized to the special value Unconstrained, meaning the variable may take any value from the domain under consideration. The filtering function / of the constraint network uses local propagation of tagged values to eliminate inconsistent values from the feasible sets, and replaces undetermined subtags with determined subtags. Guesgen showed in his thesis that filtering of tagged values is guaranteed to terminate with the globally consistent solutions of a constraint network in a finite number of steps provided that it is *fair* [1989]. This means each constraint is evaluated (or *activated)* at least once, and if a constraint changes the feasible set of any adjacent variable during filtering, other constraints adjacent to the modified variable have to be reactivated. (This is not the usual definition of fairness in the parallel programming literature.) Upon termination of local propagation, each tuple of values—one per variable—with identical full tags form a globally consistent solution of the constraint network (see Figure 2). If the feasible set of any variable becomes empty, there is no solution for the constraint network. Local propagation computes the fixed point of the filtering function / of the constraint network.

Filtering is Monotonic with respect to Partial Order $\leq$ of Tagged Values For two tagged values v_a and v_b with identical domain values, $v_a \leq v_b$ iff every determined rth subtag in v_a has an identical rth subtag in v_b. For example, R(1,-,-,-,-,-) $\leq$ R(1,1,-,1,-,-) $\leq$ R(1,1,1,1,1,1). For two sets of tagged values V and W, we extend the definition such that $V \leq W$ iff for every $w \in W$, there exists $v \in V$ such that $v \leq w$. In addition, $\leq$ is defined such that UnConstrained $\leq V$ for any set of tagged values V. For the activation of constraint network A in Figure 2, Va.Step1 = { R(1,-,-,-,-,-), Y(2,-,-,-,-,-) }, Va.Step4 = { R(1,1,-,1,-,-), R(1,3,-,2,-,-) }, so Va.Step1 $\leq$ Va.Step4.

Intuitively, filtering either eliminates a tagged value if it is inconsistent, or replaces one or more undetermined subtags of a tagged value by determined subtags. A determined subtag is *never* replaced by an undetermined subtag or another determined subtag. A more formal explanation of why filtering of tagged values is monotonic

with respect to < will be given elsewhere [Ho, 1993].

CONSAT as Discrete Relaxation In this section we show that CONSAT is a special case of discrete relaxation as defined in Section 2.

- CONSAT computes the fixed point of the discrete system $X = f(X)$ for the non-deterministic / that corresponds to the filtering function of the entire constraint network. This fixed point corresponds to the set of all the globally consistent solutions of the constraint network.
- / = {1, . . . , *n*) is the set of all the variables of the constraint network.
- / may be decomposed into component functions f_c for constraints $c \subseteq I$. $f_c: D \rightarrow D$ is the filtering function of constraint c, and eliminates value combinations inconsistent with c from the feasible sets of its adjacent variables. Without loss of generality, we assume that a constraint may be uniquely identified by its set of adjacent variables. For example, /{2,3,5} is the filtering function of the constraint that is adjacent to variables 2, 3 and 5. f_c only depends on the values of variables $i \in c$, and $f_c(X)$ only differs from *X* for the same set of variables. The new value of any such variable i is given by ${}_i f_c(X)$.
- For $X \in D$, $X = (X_1, \ldots, X_n)$. Each $X_i \in D_i$ is the feasible set of variable i.
- A"^{0} is the initial state of the feasible sets of all the variables. All its components have the special value Unconstrained.
- The partial order < on *D* has been defined on page 4.

We have shown that CONSAT is a special case of discrete relaxation, and have defined an application-specific monotonicity test between successive states of the relaxation. Consequently, an efficient parallel implementation of CONSAT may be obtained by using monotonic asynchronous iteration.

Performance We implemented parallel CONSAT using CLiP [Franz Inc., 1990], a commercial implementation of the multiprocessing features of SPUR Lisp [Zorn *et* al., 1989] that currently runs on the shared memory Sequent Symmetry multiprocessor. To estimate the performance of parallel CONSAT in the absence of garbage collection—an upper bound on parallel performance—we define speedup as the ratio of *real* time *excluding* garbage collection of the sequential version relative to the real time (excluding garbage collection) of the parallel version on *n* processors. This definition is chosen because the current CLiP implementation uses a sequential garbage collector that stops all but one processor each time a collection occurs. We feel our definition better models the performance of parallel CONSAT in a parallel Lisp system with a more realistic garbage collector, such as the concurrent collector in TOP-1 Common Lisp [Tanaka and Uzuhara, 1990].

We measured the parallel performance of CONSAT for a problem in machine vision, which assigns three-dimensional edge labelings (convex, concave, or occluding) to line drawings in a polyhedral world of trihedral vertices [Horn, 1986]. The constraints restrict the labeling of edges meeting at a vertex to be the few combinations physically possible. In addition, each edge is constrained to have the same label at both ends (where it meets other edges). The particular scene (constraint network) chosen, Stair5, has 33 variables and 56 constraints (36 2-variable and 20 3-variable constraints).

P	no gc	gc	T	Speedup	Steps	Abort
Seq	91	6	97	1.0	447	0
1	95	7	102	0.95	447	0
2	50	7	57	1.8	455	7
3	36	9	45	2.5	454	10
4	28	10	38	3.2	465	13
5	25	12	37	3.6	464	17
6	20	12	32	4.6	463	16

Table 1: Performance of CONSAT for Stair5: P is the number of processors, and Seq is the sequential implementation. All times are real time in seconds, no gc is time without garbage collection, gc is time for garbage collection, and T is the total time. Speedup is based on times *excluding* garbage collection. Both sequential and parallel versions are compiled with the highest optimization setting for speed. Steps is the number of discrete relaxation steps, *including* non-monotonic ones. Abort is the number of non-monotonic steps (re-executed).

The results for Stair5 are summarized in Table 1. Speedup ranges between 0.9 on one processor to 4.6 on six processors. The one processor time is within 5% of the sequential time, showing that the parallel implementation is reasonably efficient. The speedup is less than linear because a small number of constraint activations (relaxation steps) have to be re-executed because they violate monotonicity. Serial bottlenecks in the allocation routines of the current CLiP implementation also prevent parallel CONSAT from achieving better speedup.

An earlier parallel implementation of CONSAT that performs the update to the feasible sets *(X)* in a single critical section for each constraint activation has virtually no speedup because contention for *X* serializes all the concurrent constraint activations. The current implementation based on monotonic asynchronous iteration performs significantly better.

4 Related Work in Parallel Relaxation

The formulation of asynchronous iterations in Section 2 is similar to *generalized iterations* defined by Pohlmann in the context of parallel discrete event simulation [1991]. Generalized iterations operate in the domain of infinite streams of elements, each of which corresponds to an event over time in the system being simulated.

Parallel relaxation is also used by various schemes for solving systems of equations $X \leftarrow f(X)$ in the domain of real numbers in parallel, including *chaotic relaxations* defined by Chazan and Miranker [1969], *asynchronous iterations* defined by Baudet [1978] and *chaotic iterations*

with delay defined by Miellou [1975]. In these systems / is made up of component functions f_c, and the state X of the fixed-point computation may be decomposed into a set of (potentially disjoint) components that are computed by different f_c functions concurrently. Each f_c may be computed using multiple previous states, with a different state for each component of X, to minimize the amount of synchronization required. These schemes are optimized for the domain of real numbers by using the properties of real numbers, and their convergence criteria are somewhat analogous to the test for monotonicity used in monotonic asynchronous iterations. Our formulation of asynchronous iterations for parallel discrete relaxation is a domain-independent generalization of these parallel iteration schemes. In addition, the use of an optimistic test for monotonicity for improved parallel performance is unique to our approach.

5 Conclusion

In this paper we proposed monotonic asynchronous iteration as a correct and efficient way of implementing parallel discrete relaxation for systems for which monotonicity is a necessary correctness condition. Monotonic asynchronous iteration uses an optimistic scheme to compute a possible next state of the system. This optimistic scheme is highly efficient but is not necessarily monotonic (i.e., correct). An application-specific test for monotonicity is then applied to the computed state. If the test succeeds, the state transition is made (atomically). Otherwise, the computation is repeated using the current state. We have applied our technique to the parallel implementation of a constraint satisfaction system that computes globally consistent solutions, for which monotonicity is a necessary correctness condition that is not automatically satisfied. We believe monotonic asynchronous iteration is applicable to parallel discrete relaxation in general. It will be interesting to see if this application-specific monotonicity test is easy to derive for other discrete relaxation problems.

6 Acknowledgements

We thank Suresh Krishna for translating Miellou's paper on chaotic iterations from French into English, and Ed Wang for explaining the subtleties of lattice theory. We also thank Chu-Cheow Lim, Ed Wang, Luigi Semenzato and Kathy Yelick for their comments on various drafts of this paper.

References

[Baudet, 1978] Gerard M. Baudet. Asynchronous iterative methods for multiprocessors. Journal of the ACM, 25(2):226-244, April 1978.

[Chazan and Miranker, 1969] D. Chazan and W. Miranker. Chaotic relaxation. Linear Algebra and its Applications, 2:199-222, 1969.

[Franz Inc., 1990] Franz Inc. Allegro CLiP Manual, release 3.0.3 edition, March 1990.

[Guesgen, 1989] Hans Werner Guesgen. CONS AT: A System for Constraint Satisfaction. Research Notes in Artificial Intelligence. Morgan Kaufmann, San Mateo, California, 1989.

[Ho, 1993] Kinson Ho. High-level abstractions for symbolic parallel programming. PhD thesis, Computer Science Division (EECS), University of California, Berkeley, California, 1993. To appear.

[Horn, 1986] Berthold Klaus Paul Horn. Robot Vision. MIT Press, Cambridge, Massachusetts, 1986.

[Kasif and Rosenfeld, 1983] Simon Kasif and Azriel Rosenfeld. The fixed points of images and scenes. In Proceedings CVPR '83: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pages 454-456, Washington, DC, June 1983.

[Kasif, 1990] Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artificial Intelligence, 45(3):275-286, October 1990.

[Miellou, 1975] Jean-Claude Miellou. Iterations chaotiques a retards; etudes de la convergence dans le cas d'espaes partiellement ordonnes (Chaotic iterations with delay; studies of convergence for the case of partially ordered spaces). Comptes Rendus Hebdom ad aires des Seances De L 'Academic des Sciences, 280, Series A(4):233~236, January 1975. In French.

[Parker, 1987] D. Stott Parker. Partial order programming. Technical Report CSD-870067, Computer Science Department, University of California, Los Angeles, California, December 1987.

[Pohlmann, 1991] Werner Pohlmann. A fixed point approach to parallel discrete event simulation. Acta Informatica, 28(7):611-629, October 1991.

[Rosenfeld et ai, 1976] Azriel Rosenfeld, Robert A. Hummel, and Steven W. Zucker. Scene labeling by relaxation operations. IEEE Transactions on Systems, Man and Cybernetics, 6(6):420-433, June 1976.

[Schmidt, 1986] David A. Schmidt. Denotational Semantics: A Methodology for Language Development. Allyn and Bacon, Boston, Massachusetts, 1986.

[Stoy, 1977] Joseph E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, Massachusetts, 1977.

[Tanaka and Uzuhara, 1990] Tomoyuki Tanaka and Shigeru Uzuhara. Multiprocessor Common Lisp on TOP-1. In Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, December 1990.

[Waltz, 1972] D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI-TR-271, MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1972.

[Zorn et al., 1989] Benjamin Zorn, Kinson Ho, James Larus, Luigi Semenzato, and Paul Hilfinger. Multiprocessing extensions in Spur Lisp. IEEE Software, 6(4):41-49, July 1989.