Control in Parallel Constraint Logic Programming*

Naoki Kobayashi
koba@is.s.u-tokyo.ac.jp

Satoshi Matsuoka
matsu@is.s.u-tokyo.ac.jp

Akinori Yonezawa
yonezawa@is.s.u-tokyo.ac.jp
Department of Information Science, the University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo, JAPAN, 113

Abstract

Although constraint satisfaction problems can be specified declaratively, the cost of actual solving depends heavily on the order of computation. The ideal combination of declarative problem description and automatic satisfaction is difficult; because determination of the optimal order of computation in advance is impossible in general. Our new proposal, Parallel Constraint Logic Programming(PCLP), aims to solve this dilemma. In PCLP, computation is described in terms of parallel transformation of constraints. This approach can explore natural parallelism, and is suitable for dynamic control of the computation. Our prototype system PARCS, based on this framework, can handle constraints on Herbrand universe and finite domains by using SLD-resolution and consistency techniques such as forward checking. To control the computation dynamically, priority is associated with each process according to run-time information. In addition, learning mechanisms adjust the parameters of the scheduling policy so that scheduling becomes near optimal.

1 Introduction

It is often difficult and costly to employ imperative programming languages to program problems involving complex arithmetic or symbolic constraints, because the flow of information becomes too complex for explicit user specification. We, therefore, aim to develop a programming language which can specify constraints declaratively, and solve them automatically and efficiently. The requirements for such a programming language are as follows:

1. *Declarative specification of constraints* for clean, lucid description of problems,
2. *Dynamic and automatic control of computation* to free the user from the concern for synchronization, scheduling, and distribution.
3. *Natural exploitation of parallelism* to achieve considerable speed-up on fine-grained multicomputers.

The second requirement is especially important: in contrast to imperative programming where directions of inputs and outputs are fixed, constraint solving involves multi-directional flow of information. Thus, the system must appropriately decide and alter the order of computation in this regard. Computational frameworks for constraint programming such as Jaffar et al's CLP scheme[8] serve as frameworks for languages that can handle constraints on practical domains such as real numbers and boolean values. However, previous constraint languages such as CLP(R)[7] (which is a direct instantiation of the CLP scheme) are sometimes not sufficient to handle complex constraints for some classes of problems because the order of computation is fixed. In this paper, we propose *Parallel Constraint Logic Programming (PCLP)* as a framework for languages which satisfy our three requirements, and describe *PARCS*, a prototype language based on PCLP. PCLP can exploit a very high degree of parallelism by viewing constraints as processes. PARCS controls the computation dynamically by *prioritizing constraints*, and further optimizes their control by the *learning mechanism*. With PARCS, we can solve constraints efficiently in parallel while avoiding abusive use of parallelism which resulting in explosion in the number of processes.

In previous constraint logic programming languages, such as Prolog-III[3] and CHIP[5], literals are distinguished from other constraints, and the selection order of clauses and literals is the same as in Prolog. In our approach, literals and other constraints are handled uniformly, thereby achieving the same-level of And/Or-parallelism for constraint solving as well as resolution. Although several studies on parallelization of constraint logic programming language are already reported, such as parallel CHIP[6] by Hentenryck et al., the significance of PCLP and PARCS is that we aim not merely to distribute constraint solving to multiple processors, but emphasize on using the information of *viability of constraint solving* for fine-grained control of parallelism. Sarasswat's Concurrent Constraint Programming gives the framework for concurrent systems based on constraint solving, where concurrent agents *tell* and *ask* constraints to global store. In Concurrent Constraint Programming, the *implementation details* of the global store itself is unspecified. In-

*To appear in *Proceedings of the Logic Programming Conference, Lecture Notes in Artificial Intelligence*

stead, we concentrate on how constraints are scheduled and solved; this would, in the context of Concurrent Constraint Programming, correspond to specifying the behavior of the global store in detail with respect to constraint solving.

This paper is organized as follows. Section 2 introduces PCLP, and its comparison to the CLP scheme. Section 3 describes our prototype language PARCS, concentrating on the control of parallel computation and the learning mechanism. Section 4 proposes *lazy branching*, an execution mechanism for distributed constraint solving. Section 5 evaluates PARCS using some examples. Section 6 concludes this paper.

2 Parallel Constraint Logic Programming (PCLP)

In PCLP, atoms in traditional logic programming are also viewed as constraints. This enables the programmers to define complex numerical constraints and symbolical constraints with clauses. The uniform view of atoms and constraints contributes towards exploring natural parallelism in processing constraints as we will show.

2.1 Goal, Clause, and Program in PCLP

A goal in PCLP is a set of constraints. We denote a goal $\{C_1, \ldots, C_n\}$ as follows:

$$\leftarrow C_1, \ldots, C_n.$$

Note that a goal is a *set* of constraints, not a sequence. A clause in PCLP is a pair that consists of an atom and a goal. We denote a clause as

$$A \leftarrow G.$$

where A is an atom and G is a goal. The clause $A \leftarrow G.$ can be interpreted in two ways; either (1) operationally i.e., to solve the constraint A, solve the constraints in G, or (2) declaratively i.e., constraint A is defined as the set of constraints G. A program is a set of clauses. It can be regarded as a definition of an entire system of constraints.

2.2 Operational Semantics

We denote a program and domain theory as P and $\mathcal{T}$ respectively. The derivations in PCLP are as follows:

Derivation in PCLP When we have the goal

$$\leftarrow C_1, \ldots, C_n$$

and the condition

$$P, \mathcal{T} \models \forall(C_i, \ldots, C_{k-1} \leftarrow C'_1, \ldots, C'_l)$$

holds, then the new goal

$$\leftarrow C_1, \ldots, C_{i-1}, C'_1, \ldots, C'_l, C_k, \ldots, C_n$$

can be derived(transformed).

Successful Derivation A derivation sequence is successful iff its last goal is a normal form other than *False*, which is a unique normal form for unsatisfiable constraints. An answer is the last goal of a successful derivation.

Failure of Derivation A derivation sequence fails when its last goal is *False*. In practice, the following situations lead to the failure of derivations:

- Constraint in the goal cannot be transformed — for instance, an atom with no matching clause.
- The result of transformation is clearly unsatisfiable — for instance, $X = 1 \wedge X = 2$.

As an example of derivation in PCLP, given the clause:

$$p \leftarrow q, r.$$

in the program and the goal,

$$\leftarrow p, C_2, C_3, \ldots$$

we can derive the new goal

$$\leftarrow q, r, C_2, C_3, \ldots.$$

As another example, let $\mathcal{T}$ be the domain theory for real numbers, then the condition

$$\mathcal{T} \models \forall(X + Y = 3, X + 2 * Y = 5 \leftarrow X = 1, Y = 2)$$

holds, so from the goal,

$$\leftarrow X + Y = 3, X + 2 * Y = 5, C_3, \ldots$$

the new goal

$$\leftarrow X = 1, Y = 2, C_3, \ldots$$

can be derived.

As described above, each computation step in PCLP is a transformation of partial constraints in the goal towards a solution. In this regard, each step in SLD-resolution and constraint solving algorithms can be treated within PCLP as a transformation of partial constraints. Furthermore, PCLP can explore two kinds of natural parallelism in the transformations: One is OR-parallelism, where different transformations are applied in parallel to the same set of constraints. Another is AND-parallelism, where disjoint sets of constraints in the goal are transformed in parallel. These parallelisms are not just the ordinary AND/OR-parallelism in the usual sense (parallelism in resolution), but also capture the parallelisms in constraint solving algorithms.

2.3 Discussion - Comparison with CLP Scheme

In the CLP Scheme[8], proposed by Jaffar et al., a goal is composed of the *literal part* and the *constraint part*, and

the latter is checked for satisfiability at each derivation. In PCLP, by contrast, the satisfiability of the entire set of constraints in the goal is not checked at each derivation. Rather, we require only partial satisfiability, and the total satisfiability of the set of constraints is guaranteed only when an answer is found. This is reasonable in PCLP, because literals are regarded as constraints, and as a result the satisfiability of the entire set of constraints would have no meaning during the computation. The motivation for requiring only partial satisfiability is due to the following reasons relevant to parallel distributed computation:

1. Global information is required for checking the satisfiability of the entire set of constraints. This becomes a bottleneck for parallel computation, especially due to distribution.

2. The costs for checking satisfiability of constraints are not generally predictable, and the costs for complex constraints are usually very high. The scheme where the satisfiability of constraints is checked at each derivation, therefore, in practice could only handle special domains for which effective constraint solving algorithms are already found.

3. It is difficult to efficiently determine which transformation is optimal at each derivation. Instead, we have the system automatically attempt a set of certain transformations and judge the interim *viability* of each. PCLP can perform such automatic self-adaptation effectively due to smaller units of derivation compared to the CLP scheme.

In most cases, the satisfiability of constraints can be checked by transforming them into normal forms. For example, the satisfiability of simultaneous equations can be checked via normalization into the Gröbner base[2]. In the same manner, the entire derivation sequence in PCLP is a checking process of satisfiability for a set of constraints, and each derivation is a step towards the total satisfaction of the entire set of constraints. In this regard, the CLP scheme can be viewed as a restricted version of PCLP where constraints other than literals are always processed prior to literals.

3 PARCS: A Prototype PCLP System

A programming language based on PCLP would allow exploitation of a high degree of parallelism in solving constraints specified declaratively, because the system is given more freedom in determining the order of computation that is efficient. Simplistic exploitation of parallelism, however, would be inefficient due to excessive parallelism and redundant computation in actual execution. Control of parallelism is therefore essential. For this purpose, the *viability information of each process with respect to eventual satisfiability of the constraint set* is employed to prioritize their execution. In this section, we introduce a prototype system PARCS (PARallel Constraint Solving), based on PCLP, and describe its mechanism for the control of parallel computation. A pseudo-parallel version of PARCS written in C is running on UNIX[1].

3.1 Overview of PARCS

PARCS can currently handle constraints on two kinds of computational domains: the Herbrand universe and *Finite domains* in the style of CHIP[5]. In PARCS, finite domains are a finite subset of integers or a finite set of symbols.

Figure 1 shows a sample program in PARCS for the cryptoarithmetic puzzle 'SEND + MORE = MONEY'. The problem is to assign each S,..,Y a different digit from 0 to 9 so that the equation SEND+MORE=MONEY is satisfied. Figure 2 is the result of the execution.

As illustrated above, in PARCS, constraints can be described declaratively and lucidly, compared to the test-generation style program written in Prolog. Moreover, since PARCS automatically controls the order of computation, it is not necessary for the programmer to employ special predicates such as *freeze* for delayed evaluation. As a result, readability of program is preserved, and the dual roles of arguments as both input and output are also retained.

The following describes the derivation rules of PARCS:

- *Resolution*
  From the goal

  $$\leftarrow \ldots, p(X_1, \ldots, X_n), \ldots$$

  and the rule,

  $$p(Y_1, \ldots, Y_n) \leftarrow q_1(Z_{11}, \ldots, Z_{1n_1}), \ldots, q_k(Z_{k1}, \ldots, Z_{kn_k}).$$

  the new goal

  $$\leftarrow \ldots, q_1(Z_{11}, \ldots, Z_{1n_1}), \ldots, q_k(Z_{k1}, \ldots, Z_{kn_k}), X_1 = Y_1, \ldots, X$$

  can be derived

- *Forward Checking*

  From the goal

  $$\leftarrow \ldots, C(X), X \in D, \ldots$$

  the new goal

  $$\leftarrow \ldots, X \in E, \ldots$$
  where $E = \{e \in D \mid C(e)\text{is true.}\}$

  can be derived, where X denotes a variable, D and E some finite domain, and C denotes an arithmetic constraint comprised of $=, \neq, \leq$,etc. or a symbolic constraint comprised of $=, \neq$.

- *(Partial) Looking Ahead*

[1] Unix is a registered trademark of AT&T

```
defdomain from0to9 int{0..9}.
sendmory([S,E,N,D,M,O,R,Y]) :-
        from0to9(S,E,N,D,M,O,R,Y),
                  /* S,E,...,Y range from 0 to 9 */
        1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E
                          = 10000*M+1000*O+100*N+10*E+Y,
                               /* SEND + MORE = MONEY */
        S!=0,M!=0,      /* most significant digit is not 0*/
        all_different([S,E,N,D,M,O,R,Y]).
                            /* S,E,...,Y are all different*/

all_different([]).
all_different([A|X]) :- different(A,X),all_different(X).

different(A,[]).
different(A,[B|X]) :- A!=B ,different(A,X).
```

Figure 1: Programming Example in PARCS

From the goal

$$\leftarrow \ldots, C(X_1, \ldots, X_n), X_1 \in D_1, \ldots, X_n \in D_n, \ldots$$

the new goal

$$\leftarrow \ldots, C(X_1, \ldots, X_n), X_1 \in E_1, \ldots, X_n \in E_n, \ldots$$

where $D_i \supseteq E_i$
$\supseteq \{e_i \in D_i \mid \exists e_1 \in E_1, \ldots, \exists e_{i-1} \in E_{i-1},$
$\exists e_{i+1} \in E_{i+1}, \ldots, \exists e_n \in E_n$
$(C(e_1, \ldots, e_n)$ is true.)$\}$

can be derived, where X_i, D_i and E_i, C denote a variable, a finite domain, arithmetic constraint respectively.

- *Solving an Equation*

  From the goal,

  $$\leftarrow \ldots, f(X) = 0, \ldots$$

  the new goal

  $$\leftarrow \ldots, X \in D, \ldots$$

  where $D = \{d \mid f(d) = 0\}$

  can be derived, where $f(X)$ is a linear or a quadratic equation.

- *Variable Instantiation*

  From the goal,

  $$\leftarrow \ldots, X \in D, \ldots$$

  the new goal

  $$\leftarrow \ldots, X = e, \ldots$$

  where $e \in D$

  can be derived.

```
<PARCS>#read("sendmory").

<PARCS>:-sendmory(X).

yes
# of failures = 6
max # of or-nodes = 4
X = [9,5,6,7,1,0,8,2];

no.
# of failures = 6
max # of or-nodes = 4
```

Figure 2: Execution Example

3.2 Scheduling

3.2.1 The Basic Control Strategy

The optimal ordering of constraint solving is difficult to determine statically in general. For example, consider the following clause:

grand_parent(X,Y) :- parent(X,Z), parent(Z,Y).

Whether the literal $parent(X, Z)$ or $parent(Z, Y)$ should be given more priority over the other depends on the order of instantiation of X and Y at run-time. In order to capture this run-time information, one approach is to let the user manually specify such a dynamic decision with extralogical predicates such as var(X). This approach, however, suffers from (1) the combinatorial increase of programming burden with the increase of the number of variables, and (2) also makes the program error-prone. Instead, the approach we take in PARCS is to dynamically and automatically compute the priority for each constraint solving processes according to the run-time information with no or little user intervention. As we had indicated earlier, priority is computed based on the 'viability' information of each process, i.e., how much we can count on the process to lead to the fastest solution to the entire set of constraints, based on the state of the process.

A rough sketch of the basic strategies for process prioritization is as follows:

1. a constraint with higher level of information is given higher priority.
2. a constraint with a lower solving cost is given higher priority.

For example, "p(X) should have higher priority compared to p(a)" corresponds to the former case, and "linear equations should have higher priority compared to quadratic equation" corresponds to the latter case. The rule of computation in Warren's Andorra model[12], "deterministic goals are processed prior to non-deterministic goals" is also an example of the latter case, because branching incurs high computational cost. We could say that PARCS prioritizes constraints more sophisticatedly compared to Andorra and its variants by using the information of multiple constraint domains.

In PARCS, priority is expressed with the priority function:

$$p(S_1, \ldots, S_m, C_1, \ldots, C_n)$$

where $S_1, \ldots, S_m$ are values which depend on the state of process, and $C_1, \ldots, C_n$ are the *scheduling parameters* which are used to dynamically alter the scheduling policy. The parameters are optimized using the *learning mechanism* described in section 3.2.5.

3.2.2 Process Organization

Figure 3 illustrates And/Or organization of the processes in PARCS. Each AND-node corresponds to a process. The scheduler selects the process in two stages. It first selects a set of OR-nodes and then selects a set of their child AND-nodes. Scheduling of distinct OR-nodes in parallel results in OR-parallelism, and scheduling AND-nodes in the same OR-node results in AND-parallelism. The system controls the parallelism by associating a priority to each OR-node and AND-node.

3.2.3 Control of OR-parallelism

In order to control the OR-parallelism, the following runtime information of each process is employed:

1. Number of variables(Nv), and the range of each variable.
2. Number of AND-nodes(Ng).
3. Number of derivation steps from the initial goal(Nd).

With the above information, the priority of each OR-node is computed by the following function:

$$P_{OR} = Cv * Nv + Cg * Ng + Cd * Nd + Cgd * Ng/Nd$$

Parameters $Cv, Cg, \ldots$ are constants during a single execution. They can be set manually by the programmer or their optimal values can be automatically determined by the learning mechanism.

In our current pseudo-parallel implementation, OR-nodes are scheduled round-robin. The time quantum for priority po is:

$$Q_{OR} = Cq * f(Cpitch * (po - \overline{po})/\sigma)$$

where $\overline{po}$ is the mean of po, and σ is the standard deviation of po (In practice, the values of $\overline{po}$ and σ are approximated). Function f is the sigmoid function,

$$f(X) = 1/(1 + exp(-X)).$$

Cq is the parameter for changing the depth of the depth-first search, and $Cpitch$ adjusts the degree of the effect of the priority value on the time quantum. Both of these parameters adjust the degree of parallelism to avoid the explosion of the number of processes. Figure 4 shows the effect of $Cpitch$ on the sigmoid function. The abscissa in the figure is the normalized value of priority ($(po - \overline{po})/\sigma$) and the ordinate is the time quantum with $Cq = 1$. Processes with a time quantum shorter than a certain value are not scheduled, thereby allowing the control of the ratio of active processes. For combinatorial problems, this feature is especially important, because the number of process is likely to be very large.

The increase of Cq causes higher locality in memory reference when a contiguous memory block is allocated for each OR-node. This is evident in the current prototype and we expect it to hold for future distributed implementations.

Contrasting to other OR parameters, $Cpitch$ and Cq are changed by the system during execution according to the number of process, and are also adjusted by the learning mechanism. For optimization in our current prototype, the scheduling policy is changed to the depth-first search when the number of OR-node exceeds a certain value.

3.2.4 Control of AND-parallelism

To control the AND-parallelism, PARCS structures the multi-level scheduling queue for AND-nodes as follows, in the decreasing order of priority:

- Level 1: constraints which can be processed deterministically.
- Level 2: Non-deterministic constraints (e.g., a literal which can be unified with several clauses, a variable instantiation)
- Level 3: suspended constraints, i.e., constraints which cannot yet be processed (e.g. $X^2 + Y^3 = 1$)

In our current prototype, the scheduling algorithm differs for each level: Constraints in Level 1 queue are scheduled round-robin. Constraints in Level 2 queue are selected only when Level 1 queue is empty. The resulting scheduling strategy is thus, "deterministic goals are executed prior to non-deterministic goals", postponing the costly division of goals. This is similar to the strategies in the systems based on the Andorra model[1][12]; however, PARCS is different from Andorra in that PARCS

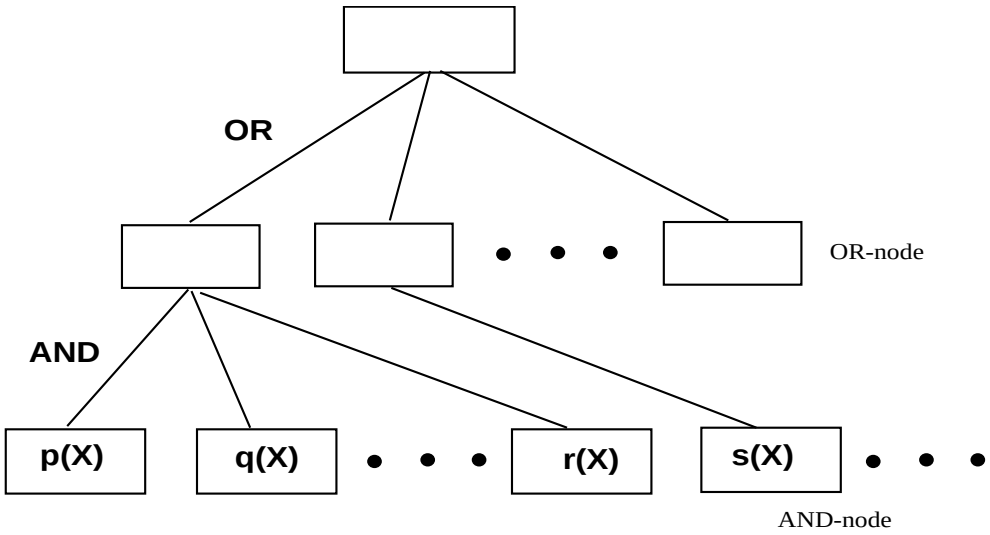

Figure 3: Process Organization

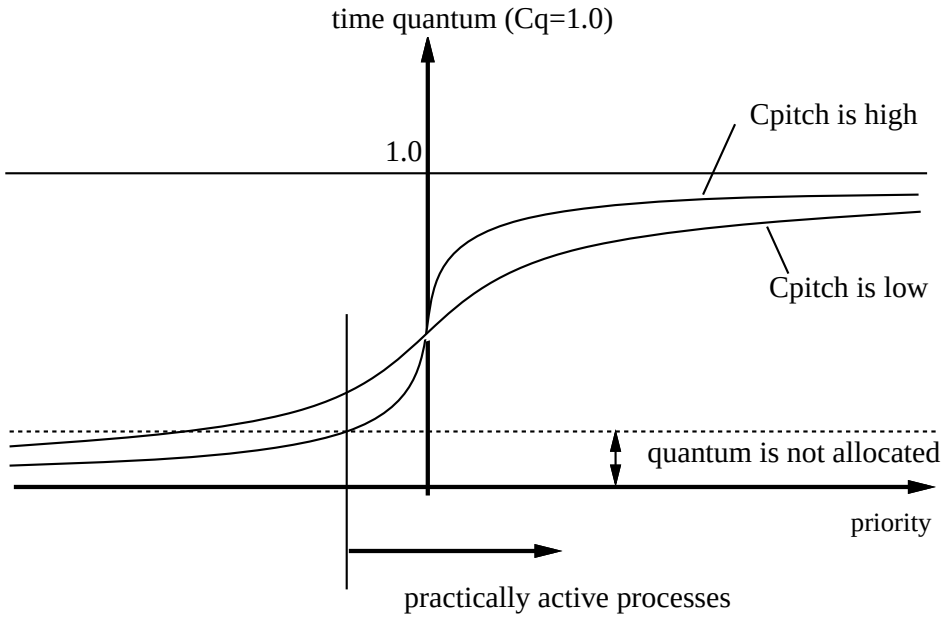

Figure 4: The effect of Cpitch

```
#priority append(0.3, 0, 0.5) 0
append([ ],X,X).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

Figure 5: Declaration for priority parameters

automatically selects the optimal goal from Level 2 queue according to priority information, whereas Andorra always selects the left-most goal. This difference is very important because the order of division of goals extremely affects the overall cost of computation.

Level 2 queue contains both atoms(for resolution) and variables(for instantiation). In order to make them compete for computational resources, PARCS prioritizes them uniformly in the following manner:

- The priority P_{AND} for the atom $p(X_1, \ldots, X_n)$ is given by the following expression:

  $$P_{AND} = C_0 + C_1 * A_1 + \ldots + C_n * A_n$$

  where A_i is either 1 or 0 according to whether the argument X_i is instantiated or not, and C_j is a predicate-dependent parameter which ranges from 0.0 to 1.0. An atom is suspended when its priority is 0. For example, suppose the predicate *append/3* is defined as shown in Figure 5; then, $append(X, [a], Y)$ is suspended until X or Y is instantiated.

- The priority for variable instantiation is given by the following expression:

  $$P_{AND} = C_{RANGE} * N_{RANGE} + C_{REF} * N_{REF}$$

  where N_{RANGE} is the size of the variable range, and N_{REF} is the number of references from other constraints. By prioritizing the processes according to these information, PARCS automatically decides when to instantiate variables at run-time.

3.2.5 The Learning Mechanism

Some logic programming languages with coroutining mechanisms, such as NU-Prolog[9], analyze programs *statically* and generate declarations for delayed evaluation. For some cases, however, it is difficult to derive such control specifications purely by static analysis. Instead, PARCS takes the *dynamic* analysis approach, i.e., provides the *learning mechanism* for priority parameters, where a program is repeatedly executed in search for an optimal value for each parameter.

As described in Section 3.2.3 and Section 3.2.4, the priority functions and time quantum functions contain several parameters to control the parallelism to optimize system performance. The system performance PF can be given as a function of all parameters involved, i.e.:

$$PF = f_{PF}(C_1, \ldots, C_N)$$

where $C_1, \ldots, C_N$ are all the parameters contained in the priority functions and the time quantum functions. The goal of learning is to "find values for $C_1, \ldots, C_N$ so that PF is minimum"(the smaller the PF, the better the performance). Since the function f_{PF} is itself unknown, the hill-up climbing algorithm is used in the current implementation to minimize PF. Parameters are moved one by one in the direction PF decreases. To avoid PF from converging to the local minimum, parameters are moved by a large scale at first and by a smaller scale afterwards.

The value of PF is computed for each execution by

$$PF = W_f * N_f + W_o * N_o$$

where N_f and N_o are the number of failures, and the maximum number of OR-nodes until the first answer is found, respectively. In a sequential context, N_f indicates the number of backtrackings and N_o is related to the amount of memory usage. The weights W_f and W_o are specified by the user.

Figure 6 shows the basic algorithm for minimizing PF. The function evaluate() in the figure executes a given sample goal, and returns the value of PF for that execution. The system executes the entire program each time a parameter is moved; thus, the number of executions in learning is kN, where N is the number of parameters and k is some constant. The number of executions can be decreased by changing the sample goals — first a simple sample and gradually increasing the complexity.

Figure 7 shows the example learning session. The input line beginning with `#learn-and` in the figure instructs the system to perform learning (`#learn-and` is a command to learn parameters for AND-parallelism.). A single learning command causes the system to perform a certain number of trials to optimize parameters.

As described above, the learning mechanism is fairly simple. Despite its simplicity, however, it is nevertheless effective in obtaining drastic improvement in performance as will be shown in section 5.

4 Lazy Branching - Distribution of Constraint Solving

In PARCS, when an OR branch occurs on processing a constraint in a goal, the effect of branching is not localized to that constraint; rather the entire constraints in the goal must be copied and divided. This division is very expensive not only because of the copying cost, but also because it imposes the necessity for synchronization of the entire goal, inhibiting AND-parallelism, and increasing redundant computation. To avoid this expensive division, we propose *Lazy Branching*, which can be stated informally as: "When a branch happens for a certain constraint in the goal, other constraints in the goal are not immediately divided. Rather, the division is suspended until the non-divided constraints reference the binding of a variable of the divided constraint; then, the non-divided constraints are divided when the reference results in a difference in their behaviors." Lazy branching has the

```
optimize()
{
        for(i=1;i<=N;i++){  /* initialize parameter C[i] */
                C[i] = 0.5;
                HIGH[i] = 1.0;
                LOW[i] = 0.0;
        }
        optimized_value = evaluate();
                   /* execute goal and evaluate performance */

        while(...){   /* iterate for several times */
                for(i=1; i<=N; i++){  /* for each parameter */
                        current = C[i];

                        C[i] = (HIGH[i]+current)/2.0;
                                    /* try increasing value */
                        test_value = evaluate();
                        if(test_value<optimized_value){
                          /* if performance is improved */
                                optimized_value = test_value;
                                LOW[i] = current;
                                continue;
                        }

                        C[i] = (LOW[i]+current)/2.0;
                                    /* try decreasing value */
                        test_value = evaluate();
                        if(test_value<optimized_value){
                                optimized_value = test_value;
                                HIGH[i] = current;
                                continue;
                        }
                        C[i] = current;
                        HIGH[i] = (HIGH[i]+C[i])/2.0;
                        LOW[i] = (LOW[i]+C[i])/2.0;
                }
        }
}
```

Figure 6: Learning Algorithm

```
<PARCS>#read("parser-dcg").

<PARCS>:-parse([this,is,a,black,pen],X).

yes
# of failures = 29
max # of or-nodes = 22
X = s(np(this),vp(v(is),np(d(a),np(a(black),np(n(pen))))))

<PARCS>:-parse([a,man,with,a,pen,is,ken],X).

yes
# of failures = 369
max # of or-nodes = 294
X = s(np(np(d(a),np(n(man))),pp(p(with),np(d(a),np(n(pen))))),
                                                vp(v(is),np(ken)))

<PARCS>#learn-and :-parse([a,man,with,a,pen,is,ken],X).

<PARCS>:-parse([this,is,a,black,pen],X).

yes
# of failures = 21
max # of or-nodes = 7
X = s(np(this),vp(v(is),np(d(a),np(a(black),np(n(pen))))))

<PARCS>:-parse([a,man,with,a,pen,is,ken],X).

yes
# of failures = 49
max # of or-nodes = 32
X = s(np(np(d(a),np(n(man))),pp(p(with),np(d(a),np(n(pen))))),
                                                 vp(v(is),np(ken)))
```

Figure 7: Example Learning Session

effect of distributing the computation, because it localizes branch operations and allows AND-sibling goals to be processed independently without strict synchronizations. It also avoids redundancy in the computation when constraints in the goal are independent, such as:

$$\leftarrow p(X), q(Y)$$

Although *Restricted AND-parallelism*[4] also has this delayed division effect, lazy branching can retain the effect even when constraints are not completely independent. Moreover, Restricted AND-parallelism requires the programs to be analyzed statically, whereas such analysis is not required for lazy branching; this is because independence of constraints is automatically exploited.

To illustrate lazy-branching, consider the program in Figure 8: Given the goal:

$$\leftarrow p(X), q(X, Y)$$

it is transformed in the following order. First by unfolding p(X) and q(X,Y) independently, the goal becomes

$$\leftarrow (X = a \text{ or } X = b), ((X = a, Y = b) \text{ or } (r(X), s(Y)))$$

where 'or' denotes a disjunction whose division is delayed. Notice that the effect of branching on p(X) and q(X,Y) is not yet propagated to each other. Then, subgoal $(X = a, Y = b)$ looks up the binding of X in its AND-sibling goal $(X = a \text{ or } X = b)$ and the goal becomes $(X = a, Y = b)$

```
p(a).
p(b).
q(a,b).
q(X,Y) :- r(X), s(Y).
r(X) :- .../* cost for r(X), s(X) is high */
s(X) :- ...
```

Figure 8: Example of Lazy Branching

for this OR-branch; thus, the top-level disjunction of the goal is now divided:

$$\leftarrow (X = a, Y = b)$$

$$\leftarrow ((X = a \text{ or } X = b), r(X), s(Y))$$

Notice that $r(X)$ is not divided into $r(a)$ and $r(b)$ yet at this point in the second goal — the division is delayed until the binding of X is required for some future transformations. Since $r(X)$ and $s(Y)$ are now independent constraints, $s(Y)$ is processed only once.

Although this lazy branching mechanism is not incorporated into our prototype PARCS, we expect it to be well-suited for the implementation of PARCS in fine-grained object-oriented concurrent programming languages; because of the locality of computation, a transformation of constraints can be viewed as a state change of objects.

```
defdomain from1to6 int {1..6}.
defdomain from1to8 int {1..8}.

sixqueens([X1,X2,X3,X4,X5,X6])
             :- from1to6(X1,X2,X3,X4,X5,X6),
                queens([X1,X2,X3,X4,X5,X6]).
eightqueens([X1,X2,X3,X4,X5,X6,X7,X8])
        :- from1to8(X1,X2,X3,X4,X5,X6,X7,X8),
           queens([X1,X2,X3,X4,X5,X6,X7,X8]).

queens([ ]).
queens([X|Y])       :- noattack(X,Y), queens(Y).

noattack(X,Y)       :- noattack(X,Y,1).
noattack(X,[],N).
noattack(X,[Y|Z],N) :- N2=N+1;
                       X!=Y, X!=Y+N, X!=Y-N,
                       noattack(X,Z,N2).
```

Figure 9: N-queens Problem

5 Evaluation of PARCS and its Learning Mechanism

5.1 The N-queens Problem - the Control of OR-Parallelism

The N-queens problem (given in Figure 9) is an example of OR-parallelism control. Table 1 shows the results: The learning was performed for N=6 for all the examples, where the number of executions in learning ranged from 40 to 80 tries. The results show that, before learning, the number of processes exploded at N=10. By learning, we were able to obtain solutions comparable to depth-first search even for this (single processor) pseudo-parallel implementation. In a true parallel implementation, we would enjoy a drastic increase in speed while concurrency is effectively controlled. Had we exploited full OR-parallelism, the number of processes naturally would have increased uncontrolled in the exponential order of N.

5.2 A Simple Parser - the Control of AND-Parallelism

Let us demonstrate the AND-parallelism control with a DCG-like simple parser in Figure 10. Here, given the clause:

$$s(X,Y,s(NP,VP)) : -np(X,Z,NP), vp(Z,Y,VP).$$

vp and *np* should be processed not in parallel, but in the sequential order of *vp*, *np* for optimal performance (The similar situations occur also for several other clauses). The results in Table 2 show that the performance after learning is comparable to the performance where the programmer explicitly specifies control. It apparently indicates that the system is able to learn the control heuristics "*vp* should be processed prior to *np*" for optimal performance. A closer examination of the control parameters confirms this conjecture: the parameters described in Section 3.2.4 for $vp/3$ moved from the initial values $C_0 = 0.1, C_1 = C_2 = C_3 = 0.3$ to $C_0 = 0.05, C_1 = 0.65, C_2 = 0.15, C_3 = 0.3$: this implies that the importance in the instantiation of first argument of *vp*, which receives the remaining unparsed sentence from *np*, became greater than the second argument. As a result, *vp* will not be processed until the first argument is instantiated by *np*, in effect serializing the execution of *np* and *vp*.

To further emphasize the ability for the system to automatically learn the order of processing constraints, consider a more elaborate example of natural language processing: Suppose we want to execute the following goal:

$$:\text{-}input(X), morpheme(X,Y), syntactic(Y,Z), semantic(Z,W)$$

When an incomplete input is given, it should be complemented by inferring from the syntactic and semantic constraints. To achieve this, the system must process these constraints in parallel while supporting *don't know* non-determinism. Although stream programming based on *don't care* non-determinism using committed-choice concurrent logic programming language such as Concurrent Prolog[10] and GHC[11] can also execute these goals in parallel, the difficulty is that streams can't directly deal with bidirectional flow of information. As a result, the program would become much more complicated compared to PARCS. This is especially evident when it is impossible for the programmer to specify the processing order in advance, e.g., determine "Which of p(X,Y) produced from syntactic constraints and q(X,Y) from semantic constraints should be processed first?". By contrast, PARCS directly and efficiently supports the don't know non-determinism in parallel constraint solving.

We also make a note that the effect of learning is observed for the sentence other than the one used for learning. This indicates that learning using small samples is not only effective for OR-parallelism, but for AND-parallelism as well; we currently believe the reason for this is that there is some consistency for optimal parameters values among different samples of the same problem domain.

	P	P/L	P/L/DF	P/D
# of failures (A)	560.5	45.6	124.4	35.8
max # of OR-nodes (A)	399.5	12.7	9.5	9.6
# of failures (B)	40.4	14.4	4.0	15.0
max # of OR-nodes (B)	27.6	4.3	6.0	4.5

P, L, DF, and D denote AND/OR-parallelism, after learning, depth-first search, declaration of priority by programmer, respectively. Sentence A is 'A man with a pen is ken' and sentence B is 'This is a black pen.' Learning was performed for the sentence 'A man with pen is ken.' Values in the table are average for 10 trials. The number of execution in learning is from 100 to 200.

Table 2: Results for Parser Program

N		before learning	after learning	depth-first search
6	# of failures	22.4	4.7	8.0
	max # of OR-nodes	33.0	9.5	4.0
8	# of failures	69.2	8.4	23.0
	max # of OR-nodes	286.8	24.4	6.0
10	# of failures	-	15.2	9.0
	max # of OR-nodes	-	69.3	7.0

'#' denotes 'number'. Learning was performed for N=6.
The average of 10 trials gave this result.

Table 1: Results for N-queens Problem

```
parse(Sentence,Ptree) :- s(Sentence,[],Ptree).

s(X,Y,s(NP,VP))       :- np(X,Z,NP),
                         vp(Z,Y,VP).

np(X,Y,np(N))         :- n(X,Y,N).
np(X,Y,np(D,NP))      :- d(X,Z,D),
                         np(Z,Y,NP).
np(X,Y,np(A,NP))      :- a(X,Z,A),
                         np(Z,Y,NP).

.....

n([pen|X],X,n(pen)).
n([man|X],X,n(man)).

.....
```

Figure 10: DCG-like Parser

5.3 Discussion - Relationship between the Effect of Learning and Complexity of Problems

Examples in Section 5.1 and Section 5.2 are fairly simple. In this section, let us consider the effect of learning for more complex, real-life problems. The learning yields the following effect:

- Restrain unnecessary expansion of nodes in the search tree.
- Prune branches of the search tree as early as possible.

By learning, in the best case, performance is expected to improve by the factor of c^N, where N is the number of derivation steps, and c is some constant. In other words, the more complex the problem, the more we expect the effect of learning to be greater. We plan to apply PARCS to a larger, real-life problems to observe the validity of this conjecture.

6 Conclusion and Future Work

We proposed PCLP, a framework of programming languages to describe and solve constraints with automatic system control of parallelism, and implemented a prototype language PARCS. Our approach, where a program is written declaratively and the system automatically exploits and controls its parallelism, seems promising for future ultra-parallel processor environment on which sole reliance on traditional imperative programming will become very difficult.

For an immediate extension to PARCS, we are considering the following: When there are several solutions, it is often important that the optimal solution be found as far as possible, rather than that the one found 'fast'. To meet this requirement, the following extensions can be incorporated into the scheduling policy:

1. Prioritize according to what clause was selected on the resolution.
2. Prioritize according to what value was selected to instantiate a variable.

Other future work includes (1) implementation of a more practical system on multi-processors, (2) investigating a formal framework of PCLP where the 'control' in PARCS can be expressed, and (3) to combine our paradigm with other paradigms such as Object-Oriented Concurrent Programming and Functional Programming. Lazy branching would be one factor in realizing (3): we are now studying a way to compile programs in PARCS into Object-Oriented Concurrent language, so that PCLP and Concurrent Objects can be combined at the implementation level.

Acknowledgment

We would like to thank Kôiti Hasida and Hirosi Tsuda for their numerous advices during the course of this work.

References

[1] Bahgat, R. and S. Gregory, "Pandora: Nondeterministic Parallel Logic Programming," in *Proceedings of the Sixth International Conference on Logic Programming*, pp. 471–486, 1989.

[2] Buchberger, B., "Gröbner bases: an algorithmic method in polynomial ideal theory," tech. rep., CAMP-LINTZ, 1983.

[3] Colmerauer, A., "Opening the Prolog-III Universe," *BYTE Magazine*, vol. 12, no. 9, 1987.

[4] DeGroot, D., "Restricted AND-parallelism," in *Proceedings of FGCS'84*, pp. 471–478, 1984.

[5] Dincbas, M., P. V. Hentenryck, et al., "The Constraint Logic Programming Language CHIP," in *Proceedings of the International Conference on FGCS*, pp. 693–702, 1988.

[6] Hentenryck, P. V., "Parallel constraint satisfaction in logic programming," in *Proceedings of the Sixth International Conference on Logic Programming and Symposium*, pp. 165–179, 1989.

[7] Jaffar, J. and S. Michaylov, "Methodology and Implementation of a CLP System," *Journal of Logic Programming*, pp. 196–218, 1987.

[8] Jaffar, J. and J.-L. Lassez, "Constraint Logic Programming," in *Proceedings of SIGACT/SIGPLAN Symposium on Principles of Programming Language*, pp. 111–119, ACM, 1987.

[9] Naish, L., "Automating Control for Logic Programs," *Journal of Logic Programming*, no. 3, pp. 167–183, 1985.

[10] Shapiro, E., "Concurrent Prolog: A Progress Report," in *Concurrent Prolog* (E. Shapiro, ed.), vol. 1, pp. 157–187, The MIT Press, 1987.

[11] Ueda, K., "Guarded Horn Clauses," in *Concurrent Prolog* (E. Shapiro, ed.), vol. 1, pp. 140–156, The MIT Press, 1987.

[12] Yang, R. and V. S. Costa, "Andorra-I: A System Integrating Dependent And-parallelism and Or-parallelism," tech. rep., Department of Computer Science, University of Bristol, 1990.