A Parallel Interval-Based Constraint Language: Implementation and Performance Analysis *

Jacques Cohen & Suresh Kalathur †
Computer Science Department
Brandeis University
Waltham, MA 02254
E-Mail: {jc, suresh}@cs.brandeis.edu

December 6, 1995

*Supported by NSF Grant CCR-94 0342762394.
†Corresponding author.
Abstract

This paper presents the design and implementation of a non-deterministic constraint programming language based on interval variables (integer or floating point intervals) on shared-memory multiprocessors. The non-deterministic constructs in the language are the choice and split statements. Each processor narrows the constraints encountered along the path of the execution tree. Whenever a solution is found, the processor is re-used in exploring other branches of the execution tree. Similarly, a failed narrowing frees the corresponding processor to be re-used with a subsequent split or choice statements. A meta-level interpreter is presented to describe the execution behavior of the language. The behavior of the meta-level interpreter is also approximated by considering strings generated by a context-free grammar and derivable from the interpreter rules. A novel approach presented here is an attempt to determine the average case speed up by generating uniform random strings using the context-free grammar. The constraint language is implemented on the SGI-ONYX which is a shared memory multiprocessor system. Speed-ups are presented for constraint problems over finite domains and floating-point intervals.

1 Introduction

This paper describes a non-deterministic constraint programming language and its parallel implementation on a shared-memory multiprocessing machine. The values of the variables are represented by intervals. Constraints are solved using the interval narrowing techniques. Non-determinism is achieved through the choice and split statements and their components are executed in parallel.

We have opted to consider an explicit choice primitive because our goal was to design a C-like language for which a syntax-directed-translator (DCG or YACC) could be used to generate code suitable for execution on a shared-memory MIMD machine. It would not be difficult to design other front ends based on CLP like languages.

CLP(BNR) extends Prolog with interval constraints [OlVe 93]. The CLP language Newton [BeMcHe 94, HeMK 95] deals with interval constraints in their entirety instead of breaking them up into primitive constraints as done in CLP(BNR). The Newton approach appears to be more efficient than the BNR method. Although the current implementation of the constraint language decomposes constraints into primitive constraints, the techniques presented in this paper could be extended to Newton.

Section 2 describes the operational semantics of a subset of the implemented language. A meta-level
interpreter, the implementation and analyses are presented in section 3.

2 Operational Semantics

The syntax of a subset of the language described in this work is captured by the following grammar:

\[ P := S; \ldots; S \]

\[ S := E = E \mid E \leq E \mid E \neq E \mid \text{BIND } v (a, b) \mid \text{CHOICE } \{S; \ldots; S\} \mid \text{SPLIT } v (E, E) (E, E) \]

\[ E := c \mid v \mid L(v) \mid U(v) \mid E + E \mid E - E \mid E \times E \mid E / E \mid \neg E \mid E \land E \mid E \lor E \]

\( P \) denotes the program, \( S \) the statements, and \( E \) the expressions. \( x, y, z, v \) denote variables and \( a, b, c \) denote constants. The domain considered here is the set of real numbers \( \mathbb{R} \).

The narrowing operation considers a constraint and attempts to reduce the interval domain of its variables. For any variable \( x \), let \( \sigma(x) \) denote the interval for \( x \), i.e., \( \sigma(x) = (L(x), U(x)) \), where \( L \) and \( U \) represent the numeric values of the lower and upper bounds respectively. Let \( \text{var}(S) \) denote the set of variables appearing in a (single) constraint \( S \). For each \( x_i \in \text{var}(S) \), let \( N(x_i, S) \) represent the narrowed interval domain after applying the narrowing operation for the constraint \( S \). Let \( \sigma \) denote the interval vector. The notation \( \sigma[i_x/x] \) represents the interval vector obtained by replacing the current interval for the variable \( x \) with the new interval \( i_x \).

**Definition:** A state of the constraint solver is defined as a pair \( \delta =< S, \sigma > \), where \( S \) is the set of constraints to be processed, and \( \sigma \) is the interval-vector comprising the interval domains of the variables appearing in the source program \( P \). Let \( \mathcal{P}x \) denote the set of constraints in \( \mathcal{P} \) which contain the variable \( x \).

**Definition:** A transformation is defined as the change of the given state by a single step of the constraint solver. The transformation of the state \( < S, \sigma > \) is represented in the following way:

\[ < S, \sigma > \rightarrow < S', (b, \sigma') > , \quad b \in \{ \text{true, false} \} . \]

If \( b \) is \text{true}, the new state of the constraint solver is \( \delta' =< S', \sigma' > \), otherwise a failure state is reached.
The constraint solver is described by an automaton whose behavior is simulated by the program given below.

Input: \( < S, \sigma > \) (Set of constraints \( S \) and interval vector \( \sigma \))

\[
b \leftarrow \text{true}
\]

while \( b \) and \( S \neq \phi \) do

narrow:

\[
< S, \sigma > \rightarrow < S', (b', \sigma') >
\]

if \( b' \) then

\[
( < S, \sigma > , b ) \leftarrow ( < S', \sigma' > , b' )
\]

else failure; exit

end while

Output: \( \sigma \)

The operational semantics of the language constructs is presented below. The BIND rule specifies that binding a variable to an interval can only be done once, by narrowing the \((-\infty, \infty)\) interval to the given bounds. The CHOICE statement specifies that one of the alternatives is selected non-deterministically. The other alternatives are explored either by parallel exploration of the choice branches or by backtracking. The SPLIT statement specifies how the values of a variable can be split into two interval components and explored in parallel.

The constraints are assumed to be of the form \( op(x, y, z) \), where \( op \) is a ternary, binary, or a unary operator, and the variables \( x, y, \) and \( z \) are the operands. (In the case of binary and unary constraints, the extra operands are filled with dummy variables.) Before performing the narrowing operation, the constraints are checked to determine if they are passive or not. A constraint is termed passive if the interval values of the operands are such that future narrowings of the values by other constraints do not affect the satisfiability of the constraint. If a constraint is determined to be passive, it is added to the passive constraint set \( \mathcal{P}_{\text{passive}} \).

The narrow operation considers a constraint and determines the narrowed intervals for the operands. If the intervals of any of the operands differ from their original value, all the non-passive constraints in which the respective operands occur are added to the new set of constraints for which narrowing has to be redone.

**Notation:** The list \( c_1 \cdot [ c_2, \ldots, c_n ] \) represents the set \( \{ c_1, c_2, \ldots, c_n \} \). \( S_1 \cup S_2 \) represents the union of the two sets.
Initialize Variables:

\[
< \text{BIND } x (a, b) \cdot S, \sigma > \rightarrow \begin{cases} 
  < S, (\text{true}, \sigma[(a, b)/x]) > & \text{if } \sigma(x) = (-\infty, \infty), \\
  < S, (\text{false}, \sigma) > & \text{otherwise}.
\end{cases}
\]

Choice:

\[
< \text{CHOICE } \{S_1; S_2; \ldots; S_n\} \cdot S, \sigma > \rightarrow \begin{cases} 
  < S_1 \cdot S, \sigma > & \text{or} \quad < \text{CHOICE } \{S_2; \ldots; S_n\} \cdot S, \sigma > 
\end{cases}
\]

Split:

\[
< \text{SPLIT } x (E_{f_1}, E_{f_2}) (E_{s_1}, E_{s_2}) \cdot S, \sigma > \rightarrow \begin{cases} 
  < S, (\text{true}, \sigma[(\sigma(E_{f_1}), \sigma(E_{f_2}))/x]) > & \text{or} \\
  < S, (\text{true}, \sigma[(\sigma(E_{s_1}), \sigma(E_{s_2}))/x]) > & \text{if } L(x) \neq U(x), \\
  < S, (\text{true}, \sigma) >, & \text{otherwise}.
\end{cases}
\]

Passive Check:

\[
< s \cdot S, \sigma > \rightarrow < S, (\text{true}, \sigma) >
\]

\[
\begin{cases} 
  s \text{ is } x \leq y \text{ and } U(x) \leq L(y) \\
  s \text{ is } x \neq y \text{ and } (L(x), U(x)) \cup (L(y), U(y)) = \bot \\
  \ldots
\end{cases}
\]

\[\text{Or represents don't know non-determinism.}\]
Narrow:

\[
\begin{align*}
< s \cdot S, \sigma > & \rightarrow \begin{cases} 
< S \cup S', (\text{true}, \sigma') > & \text{where } s = \text{op}(x, y, z) \text{ and } S' = \bigcup_{i \in \{x, y, z\}} S^i, \\
S^i = \begin{cases} 
\mathcal{P}^i \setminus \mathcal{P}^{\text{passive}} & \text{if } \sigma(i) \neq \mathcal{N}^{\text{op}}(i, s) \\
\phi & \text{otherwise}
\end{cases}, \\
\text{and } \sigma' = \sigma[\mathcal{N}^{\text{op}}(i, s)/i], i \in \{x, y, z\}, \\
< S, (\text{false}, \sigma) > & \text{if } \exists i, i \in \{x, y, z\}, \text{ such that } \mathcal{N}^{\text{op}}(i, s) = \perp
\end{cases}
\end{align*}
\]

**Example:** Consider the statement \textit{SPLIT} \(x \ (L(x), L(x)) \ (L(x) + 1, U(x))\). Let \(\sigma(x) = (1, 10)\). The variable \(x\) in the two interval vectors resulting from the \textit{SPLIT} statement will have the intervals \((1, 1)\) and \((9, 10)\) respectively. Suppose the \textit{SPLIT} statement is specified as \textit{SPLIT} \(y \ (L(y), (L(y) + U(y))/2) \ ((L(y) + U(y))/2, U(x))\). Let \(\sigma(y) = (1, 10)\). In this case, the variable \(y\) in the two interval vectors will have the intervals \((1, 5.5)\) and \((5.5, 10)\). If the variable \(y\) is constrained to be an \textit{integer} variable, then the above intervals would be narrowed to \((1, 5)\) and \((6, 10)\) respectively.

**Example:** Consider the integer constraints

\[
\begin{align*}
(1) \ x \neq y & \quad (2) \ x \neq z & \quad (3) \ y \neq z
\end{align*}
\]

We have \(\mathcal{P}^x = \{1, 2\}, \mathcal{P}^y = \{1, 3\}, \text{ and } \mathcal{P}^z = \{2, 3\}\). Let \(\sigma(x) = (1, 2), \sigma(y) = (3, 4), \text{ and } \sigma(z) = (3, 3)\). The details in narrowing the constraint set \(\{1, 2, 3\}\) are shown below:

\[
< \{1, 2, 3\}, \sigma > \rightarrow < \{2, 3\}, (\text{true}, \sigma) >,
\]

since the constraint (1) is passive. Hence, \(\mathcal{P}^{\text{passive}} = \{1\}\).

\[
< \{2, 3\}, \sigma > \rightarrow < \{3\}, (\text{true}, \sigma) >,
\]

since the constraint (2) is passive. Hence, \(\mathcal{P}^{\text{passive}} = \{1, 2\}\).

\[
< \{3\}, \sigma > \rightarrow < \{\} \cup \{3\}, (\text{true}, \sigma') >,
\]

\(\sigma'(x) = (1, 2), \sigma'(y) = (4, 4), \text{ and } \sigma'(z) = (3, 3)\). While narrowing the constraint (3), the interval for the variable \(y\) changed. Hence the set \(\mathcal{P}^y \setminus \mathcal{P}^{\text{passive}}\), i.e., the set \(\{1, 3\} \setminus \{1, 2\} = \{3\}\) needs to be added to the set of constraints yet to be processed.

\[
< \{3\}, \sigma' > \rightarrow < \{\}, (\text{true}, \sigma') >.
\]

Since the resulting constraint set is empty, the constraint solver terminates successfully with the interval vector \(\sigma'\).
3 The Meta-Level Interpreter

A meta-level interpreter for a minimal CLP(Interval) parallel language is presented in Figure 1. For description purposes, the input program is considered as a list of statements. A statement is either a choice statement, a split statement, a bind statement, or a constraint. The components of the choice statement as well as the split statement are executed in parallel. The constraints are assumed to be quadruples of the form \((op, x, y, z)\). The solve procedure has four parameters:

1. the list of statements to be processed,
2. the current set of accumulated constraints,
3. the input interval vector of the variables appearing in the source program, and
4. the new interval vector obtained by updating the input interval vector.

```prolog
solve([[choice, [Statement|Rest]]|RestStatements], ConstrList, Vec, New_Vec) :-
    parallel(
        solve([Statement|RestStatements], ConstrList, Vec, New_Vec);
        solve([Rest|RestStatements], ConstrList, Vec, New_Vec)
    ).
solve([[split, Var, First, Second]|RestStatements], ConstrList, Vec, New_Vec) :-
    split.intervals(Var, Vec,First, Second, Vec1, Vec2),
    parallel(
        solve(RestStatements, [[split, Var, First, Second]|ConstrList], Vec1, New_Vec);
        solve(RestStatements, [[split, Var, First, Second]|ConstrList], Vec2, New_Vec)
    ).
solve([[Constraint|RestStatements], ConstrList, Vec, New_Vec) :-
    passive(Constraint, Vec), !,
    solve(RestStatements, ConstrList, Temp_Vec, New_Vec).
solve([[Constraint|RestStatements], ConstrList, Vec, New_Vec) :-
    narrow([[Constraint|ConstrList], Vec, Temp_Vec],
    solve(RestStatements, [Constraint|ConstrList], Temp_Vec, New_Vec).
solve([], ConstrList, Vec, New_Vec) :- print_solution(Vec).
solve([], ConstrList, Vec, New_Vec) :- enumerate(ConstrList, Vec, New_Vec).
```

Figure 1. Meta-Level Interpreter
In the above Figure, the command \texttt{parallel}(G_1;G_2) specifies that the goals \(G_1\) and \(G_2\) are to be executed in parallel using copies of their current parameters. The procedure \texttt{split\_intervals} splits the given interval vector into two components as specified by the \texttt{split} statement. The procedure \texttt{passive} determines if the given constraint is passive (as defined by the operational semantics) with respect to the given interval vector. If the constraint is not passive, the \texttt{narrow} procedure is called with the current constraint appended with the accumulated constraints. The narrow procedure determines the new interval vector such that the narrow operation on all the constraints converges.

An interval vector is a solution if the lower and upper bounds are equal for all the variables which are constrained to be integers, or the bounds are within a desired precision for the variables not constrained as integers. The procedure \texttt{enumerate} is useful in enumerating the solutions. The \texttt{split} statement is also used as a directive in the enumeration phase. The procedure \texttt{split\_var} determines the variable to be split (for e.g., using the first-fail heuristic, selecting the variable with the smallest difference in its upper and lower bounds).

\begin{verbatim}
enumerate(Constr\_List, Vec, New\_Vec) :-
    split\_var(Constr\_List, Vec, Var, First, Second),
    split\_intervals(Var, Vec, First, Second, Vec1, Vec2),
    parallel(
        (narrow(Constr\_List, Vec1, Temp\_Vec),
            enumerate(Constr\_List, Temp\_vec, New\_Vec));
        (narrow(Constr\_List, Vec2, Temp\_Vec),
            enumerate(Constr\_List, Temp\_vec, New\_Vec))
    ).
\end{verbatim}

\textbf{Example:} An example illustrating the \textit{choice} and the \textit{split} statements is shown in Figure 2. The execution tree of the program is shown in Figure 3.
BIND y (1, 3);
CHOICE {
    {x = 1};
    {x = 2};
    {x = 3}
};
SPLIT y (L(y), L(y)) (L(y) + 1, U(y));

Results: 1, 2, 1, 3 2, 1 2, 3 3, 1 3, 2

Figure 3: Execution Tree for the program in Figure.
3.1 Implementation on a Shared Memory MIMD Machine

The parallel execution model supported on the shared memory MIMD machine (SGI-Onyx) is the fork-join model shown in Figure 4b. This model does not map directly to the parallelism exploited in the solve procedure since it uses the parallel forking on the components of the CHOICE and SPLIT constraints, assuming an unbounded number of processors. In the case of a finite number of processors, the parallel scheduling is achieved using a stack (or queue) which is shared by the various parallel processors (Figure 4a). Each available processor acquires work from the shared stack. Whenever a processor acquires work, it is said to be busy.

The arguments of the solve procedure are stored in the shared stack. Whenever a processor encounters the CHOICE or the SPLIT statement, it pushes the arguments of the second parallel candidate onto the shared stack and continues execution with the first candidate. (Recall that CHOICE and SPLIT operate on binary trees.) A processor becomes available if it encounters a successful leaf node of the execution tree or if the narrow operation being executed fails. The solve procedure terminates when the shared stack is empty and all the processors are free. Otherwise, a free processor attempts to pop the required arguments from the shared stack and continue the execution. The operations with the shared stack are done in a critical region in which only one processor can enter at a given time.

![Figure a. Parallelism in Algorithm](image1)

![Figure b. Parallelism Model on Shared Memory MIMD Machine](image2)

Figure 4: Shared Memory MIMD Machine Execution model.

The parallel procedure psolve (see Figure 5) forks the required number (say, $N$) of psolve_per_proc
(see Figure 6) processes. The main call is \( psolve(p,NIL,iv) \), where \( p \) points to the root of the program (binary) tree, and \( iv \) is the interval-vector with all the variables initialized to \( (-\infty, \infty) \). The second parameter \( cl \) (initialized to \( NIL \)) contains the list of accumulated constraints. The procedure \( fork \) initiates \( N \) executions of copies of \( psolve\_per\_proc \) procedure.

\[
\text{procedure } psolve(p,cl,iv) \\
\text{begin} \\
\quad \text{shared } num\_busy\_procs \leftarrow 0 \\
\quad \text{shared } stack \leftarrow \text{empty} \\
\quad push(p,cl,iv) \\
\quad fork( N, psolve\_per\_proc()) \\
\text{end } psolve
\]

Figure 5: The procedure \( psolve \).

The three critical regions of the program in Figure 6 correspond to attempts of a processor to access the global stack (or queue). This occurs when:

1. a \textit{choice} command is encountered and a new processor is needed; in this case the request for obtaining a new processor is pushed on the stack;

2. a \textit{split} command is encountered and its execution requires a new processor; the behavior is analogous to that described for the choice command;

3. if a failure is determined or if a processor successfully completes its task then the critical region is entered to find if there are remaining tasks to be executed in the stack.
procedure psolve_per_proc()
begin
work_completed ← false
while not work_completed do
work_acquired ← false
    critical region { if pop(p, cl, iv) then
        num_busy_procs ← num_busy_procs + 1; work_acquired ← true }
    if work_acquired then
        while p ≠ NIL do
            switch head(p)
            case CHOICE:
                critical region { push(cons(tail(head(p)), tail(p)), cl, iv) }
                p ← cons(head(head(p)), tail(p))
            case SPLIT:
                split_intervals(head(p), iv, iv1, iv2)
                critical region { push(tail(p), cons(head(p), cl), iv2) }
                p ← tail(p); iv ← iv1
            otherwise: /* a constraint */
                if narrow(cons(head(p), cl), iv, iv') then
                    p ← tail(p); cl ← cons(head(p), cl); iv ← iv'
                else break
            end switch
        end while
        critical region { num_busy_procs ← num_busy_procs − 1 }
    end if
    critical region { if stack_empty and num_busy_procs = 0 then work_completed ← true }
end while
end psolve_per_proc

Figure 6: The code executed by each processor.
3.2 Analysis

The calling sequence of the recursive procedure solve can be expressed by strings generated by a context-free grammar. Let the sequential execution of two processes $P_1$ and $P_2$ be denoted by $P_1 P_2$ and the parallel execution of $P_1$ and $P_2$ be represented by $\frac{P_1}{P_2}$. The context-free grammar derived from the program shown in Figure 1 is as follows:

$$.solve \rightarrow \textit{CHOICE} \quad \frac{solve}{solve} \quad | \quad \textit{SPLIT} \quad \frac{solve}{solve} \quad | \quad \textit{narrow} \quad \frac{solve}{solve} \quad | \quad \textit{enumerate} \quad | \quad \epsilon$$

$$\textit{enumerate} \rightarrow \textit{SPLIT} \quad \frac{narrow}{narrow} \quad \frac{enumerate}{enumerate} \quad | \quad \epsilon$$

To each program execution there is a corresponding syntax-tree which is obtained from the execution tree by disregarding the values of the variables. We can now consider an abstract model of the execution tree as the syntax-tree of the given context-free grammar. This abstract representation is convenient for attempting to provide an analytical study of speed-ups. A parse of the example shown in Figure 2 is outlined in Figure 7.

<table>
<thead>
<tr>
<th>a. narrow SPLIT</th>
<th>a1. narrow</th>
<th>a2. narrow SPLIT</th>
<th>a21. narrow</th>
<th>a22. narrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>narrow CHOICE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b1. narrow SPLIT</th>
<th>b11. narrow</th>
<th>b12. narrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>b. CHOICE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b2. narrow SPLIT</th>
<th>b21. narrow</th>
<th>b22. narrow</th>
</tr>
</thead>
</table>

Figure 7: Parse of the Execution Tree in Figure 3.

An estimate of the average execution time can be obtained by generating all strings of length $n$ using the context-free grammar $G$. Let $m$ be the number of such strings (usually a very large number compared to $n$). Let $E_i$ denote the execution time for string $i$. The average execution time is $\frac{\sum_{i=1}^{m} E_i}{m}$. When $n$ is small, all the strings of length $n$ can be generated by using DCG’s. However, for large $n$, there are techniques that do not require the generation of all the strings. These techniques developed by [HiCo 88, FLAJ 87] just generate the so called uniformly random strings. Such a uniform random generation can be thought of as one in which each proposed string is chosen according to an index of a random number between 1 and $m$. 
3.3 Processor Scheduling

For an analytical study of the number processors required the nodes of the execution-tree are assigned processor labels. Assuming an unbounded number of processors, a labelling of the execution-tree in Figure is shown in Figure 8. When a SPLIT or a CHOICE is encountered, the node of the left branch is assigned the same label as its parent. The node of the right branch is labeled with the next available processor label. Let us assume that the processor labels are stored in a stack. When a leaf node is encountered, the processor label is free and pushed onto the stack. In the case shown in Figure 8, six processors are needed to complete the execution in five time steps.

In the case of a finite number of processors, some portions of the execution-tree are suspended until the processors become available. The table in Figure 9 shows the number of time steps needed with varying the number of processors for the execution tree in Figure 3. The context-free grammar is used in generating random strings of arbitrary length [ZIMM 92]. The speed-up achieved with varying the number of processors can be analytically calculated [SHUB 95].

![Figure 8: Labelling the Execution Tree in Figure 3.](image)

The program shown in Figure 10 labels the given execution tree. Constant (unit) time assumptions are made for choice, split, narrow, and enumerate. The execution tree is input as a list of triplets of the form \((Node, Level, Id)\), where \(Node\) is the node label, \(Level\) is the level of the node in the execution tree, and \(Id\) is the identifier for the node as mapped to a complete binary tree. A sample execution tree along with its input representation is shown in Figure 11.

The program shown in Figure 10 can be extended to take into account varying times for choice, narrow, split, and enumerate. Random strings of arbitrary length can be drawn using the context-
### Table: Time Steps for the Execution Tree in Figure 3.

<table>
<thead>
<tr>
<th>#Processors</th>
<th>Time Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 9: Time Steps for the Execution Tree in Figure 3.

free grammar:

\[
S \rightarrow cSS \mid sSS \mid nS \mid e
\]

(c: choice, s: split, n: narrow, e: enumerate)

The speed-ups with varying the number of processors can be analytically calculated. The speed-up curves are shown in Figure 12. The notation used in the figure is as follows:

L.String-length.c-time.s-time.e-time.n-time

r.c-time.s-time.e-time.n-time
Figure 10: Program for Labelling the Execution Tree
Input Representation:

\[
\begin{array}{c}
\text{[ (c,1,1), (s,2,2), (c,2,3), (c,3,4), (n,3,5), (n,3,6), (c,3,7), (n,4,8), (n,4,9), (e,4,10), (e,4,12), (n,4,14), (n,4,15), (e,5,16), (e,5,18), (e,5,28), (e,5,30) ]}
\end{array}
\]

Figure 11: Sample Execution Tree
Figure 12: Speed-Up Curves
<table>
<thead>
<tr>
<th>Problem</th>
<th>Seq. Time (sec)</th>
<th>Parallel Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>i4</td>
<td>1.55</td>
<td>2.0</td>
</tr>
<tr>
<td>Combustion</td>
<td>63.4</td>
<td>2.0</td>
</tr>
<tr>
<td>9-Queens</td>
<td>10.3</td>
<td>2.0</td>
</tr>
<tr>
<td>13-Queens</td>
<td>160.4</td>
<td>2.0</td>
</tr>
<tr>
<td>Job-Shop (abz6)</td>
<td>42.5</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Figure 13: Speed-ups

4 Results

The speed-up achieved on various problems is shown in Figure 13. The constraint solver is run on an SGI-Onyx with 16 processors. The problems i4 and combustion are from [HeMK 95]. The data for the Job-Shop problems is taken from [ApCo 91].

5 Final Remarks

Our main result is that one can obtain linear speed-ups on MIMD processing of a parallel language when the execution tree contains a substantial number of nodes. The execution of the choice and split primitives have similar behavior and represent operations that can be effectively done in parallel. The narrowing, as performed by our interpreter, is essentially sequential but it may be worthwhile to study if narrowing parallelization could improve the speed-ups.

The approach of using strings generated by context-free grammar to simulate the parallel execution is definitely worthwhile since it enables a fast estimate of average speed-ups without resorting to actual experimental performance analysis using parallel computers. The present model for estimating speed-up is a simple one that could be easily refined to account for waiting times spent in critical regions and the time to spawn new processes. It would suffice to include in the tree traversal programming mechanisms that simulate locks and the forking of new processes.
References


