Data-Parallel Design of an Interval-Based Constraint Language: Analysis of Speed-Ups [1]

Jacques Cohen
Computer Science Dept.
Brandeis University
Waltham, MA, USA 02254
Phone: (617) 736-2702
Fax: (617) 736-2741
jc@cs.brandeis.edu

Suresh Kalathur
Computer Science Dept.
Brandeis University
Waltham, MA, USA 02254
Phone: (617) 736-2745
Fax: (617) 736-2741
suresh@cs.brandeis.edu

Abstract

This paper presents what is believed to be the very first design and implementation of a deterministic data-parallel constraint language based on interval variables (integer or floating point intervals). The approach used consists of having one constraint and one interval variable per processor. Each processor narrows and intersects the constraint it stores within its memory using interval variables which are stored in different processors. Whenever a solution is found, it is printed and the processor is re-used in implementing the splitting operation. Similarly, a failed narrowing or intersection frees the corresponding processor to be re-used in a subsequent split operation.

The algorithm starts with an initial set of processors corresponding to the constraints to be solved and their variables. Narrowing and intersections are applied until convergence. Splitting is done by selecting the variable with the smallest interval and introducing a new set of processors whose data is copied from the previous set except for the variable chosen for splitting.

The paper also presents a Markovian analysis of the complexity of the proposed algorithms. Actual results were obtained using a Maspar MP2 with 4,096 processors.

Keywords: Parallel algorithms: SIMD; Parallel applications: Numerical computations, Interval arithmetic; Software: Compilers, Languages.

1 Introduction

This paper describes data-parallel algorithms designed to process interval constraints using SIMD computers. The approach consists in specifying a set of linear or non-linear

[1] Supported by NSF Grant CCR-94 0342762394.

constraints involving variables which are defined by integer (or floating-point) intervals consisting of lower and upper bounds [6]. The constraints are processed simultaneously using well known narrowing operations which are guaranteed to converge (or to fail) [4, 7]. If the resulting intervals for a given variable are different, the intersection operator is used to find the appropriate common value, if it exists. If the intersection fails then there is no solution for the variable in the given initial intervals.

A solution to the set of constraints is found when the intersected values of every variable are such that the lower bound equals the upper bound. If this result is unachievable after the narrowing and intersection, then one has to resort to the splitting operation. That consists of replacing an interval by two interval components, and reapplying the narrowing and intersection operations until a solution is found.

The novel aspects of this paper are to perform the needed operations using data-parallel algorithms. These are described in a NESL-like language [1] which operates on vectors of constraints and interval variables. The algorithms make use of typical data-parallel operations (like prefix-sum, prefix-min) which can be performed in logarithmic time in SIMD machines.

A set of processors deals with a set of constraints and the intervals of their variables. An important component of the algorithm is the collection and compaction of unused processors namely those which were utilized in narrowing and intersections that fail, or those which have found output solutions. New sets of processors and intervals are created by the split operation. This consists of doubling the set of processors by considering one of the variables for splitting, the remaining ones being copied. The variable considered for splitting is that containing the smallest interval, thus increasing the chances of earlier success or failure. The memory and processor requirements are doubled by each splitting and are decreased whenever narrowing fails and solutions are found.

The paper also presents a Markovian analysis of the complexity of the proposed algorithms. Actual results were obtained using a Maspar MP2 with 4,096 processors [5].

2 Proposed Algorithm

Given a set of m constraints involving n variables, let $I = \{(l_1, u_1), (l_2, u_2), \ldots, (l_n, u_n)\}$, be the set of couples denoting the lower and upper bounds of the n variables, $V = \{V_1, V_2, \ldots, V_n\}$. Let C be the set of constraints involving these variables, $C = \{C_1, C_2, \ldots, C_m\}$. Each C_i is of the form (op_3, V_j, V_k, V_l) (e.g., $(+, V_3, V_{10}, V_8)$), or (op_2, V_j, V_k) (e.g., $(\neq, V_1, V_5)$), or (op_1, V_j) (e.g., $(integer, V_2)$), $1 \leq j, k, l \leq n$.

A set of processors P operates on m constraints and n variables. A flowchart of the basic operations of the data-parallel algorithm is shown in Figure 1. Each processor $p_i \in P$ contains only the constraint C_i and the (l_i, u_i) interval pair of the variable V_i. Narrowing is done in parallel by set(s) of processors P. Each $p_i \in P$ operates on its corresponding C_i and computes the new interval pairs for the variables V_j, V_k, or V_l. Since different processors use different constraints, the narrowed I_j (the interval pair of the variable V_j) computed by one processor is not necessarily equal to the narrowed I_j computed by a different processor. The intersection operation reduces, in parallel, all of these to a new pair I_j, which is stored back in place of the previous I_j in processor j. Once

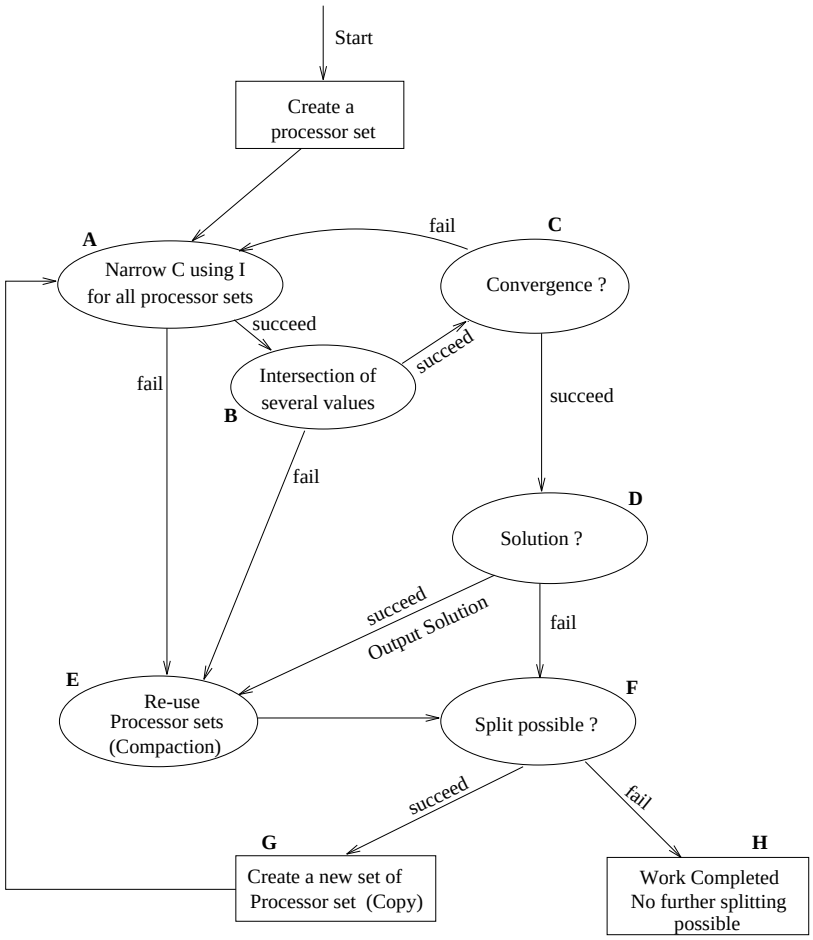


Figure 1: Flowchart of the Data-Parallel Solver

the new set I is computed, and it differs from the previous I, the narrowing operation is repeated using the new I in lieu of the previous one. The process of narrowing and intersection is repeated until I converges. (See loop $(ABC)^+$ in Figure 1.) Whenever the narrow operation on any of the C_i's fails, or if the intersection of any of the I_j's fails, the current interval pair set I is abandoned by the set of processors P and can be re-used to store other I's.

2.1 Creating New Sets of Processors

Whenever convergence is achieved, the new computed I may not be a solution. If the new I is a solution, the set of processors P can be reused after the solution output. Otherwise, a new set of processors is created by the split operation. The constraint set C and the corresponding interval sets are copied over to the new set of processors. The old set of processors contain a split interval for V (the variable one wishes to split) and the new set of processors acquire the complement of the split interval. Now we have a set of set of processors SP operating in parallel on different I's[2]. The SP grows until all the solutions are found. So, at any instant, let us assume that there are r set of processors, i.e., $SP = \{P^{(1)}, P^{(2)}, \ldots, P^{(r)}\}$, operating on r different I's, i.e., $SI = \{I^{(1)}, I^{(2)}, \ldots, I^{(r)}\}$). The $P^{(i)}$'s in parallel do the nar-

[2]Different in the sense that *one* of the variables has been split.

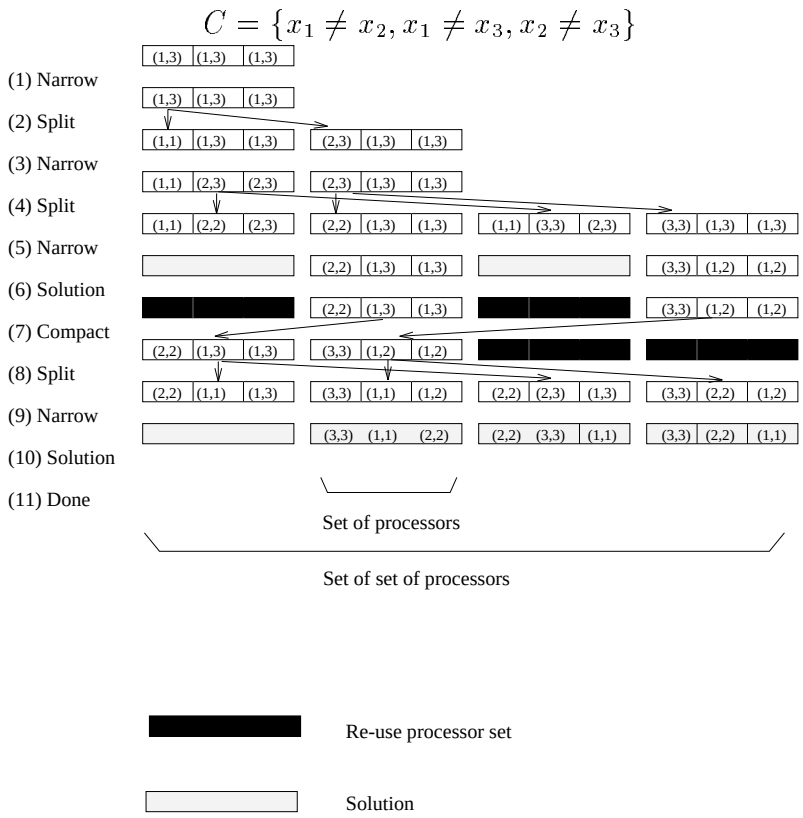


Figure 2: Example Illustrating the Operations in the Data-Parallel Solver

rowing and perform intersections until the $I^{(i)}$'s remain unchanged. If none of the intersections have failed and if none of $I^{(i)}$'s is a solution, the split operation creates r new set of processors, i.e., $SP = \{P^{(1)}, \ldots, P^{(r)}, P^{(r+1)}, \ldots, P^{(2r)}\}$, and r new I's, i.e, $SI = \{I^{(1)}, \ldots, I^{(r)}, I^{(r+1)}, \ldots, I^{(2r)}\}$. However, if some of the narrowed intersections fail, or if some of them are solutions, the corresponding set of processors are free and can be re-used by the split operation. (These correspond to the paths AE, ABE and $ABCDE$ in Figure 1) Let r' be the cardinality of the set of set of processors which have to participate in the split operation ($r' < r$). Let this set of set of processors be : $SP' = \{P^{(k_1)}, P^{(k_2)}, \ldots, P^{(k_{r'})}\}$ which operate on the corresponding interval sets: $I' = \{I^{(k_1)}, I^{(k_2)}, \ldots, I^{(k_{r'})}\}$. To maximize the use of the set of processors, the $I^{(k_i)}$'s are mapped onto the processors set's $P^{(i)}$'s: $I^{(i)} = I^{(k_i)}, \quad 1 \leq i \leq r'$.

We call this phase *compaction*. Now the r' interval sets reside within the set of processors $\{P^{(1)}, \ldots, P^{(r')}\}$. In this case, the *split* operation creates r' new set of processors: $SP = \{P^{(1)}, \ldots, P^{(r')}, P^{(r'+1)}, \ldots, P^{(2r')}\}$, and r' new I's: $SI = \{I^{(1)}, \ldots, I^{(r')}, I^{(r'+1)}, \ldots, I^{(2r')}\}$. The *split* operation terminates when r' becomes 0. An example illustrating the above operations is shown in Figure 2.

2.2 The Split Operation

The algorithm illustrating the data-parallel solver is shown in Figure 3. Consider the kth iteration of the *while* loop in Figure 3; let r_k be the number of *active* interval sets, $I_k^{(1)}, I_k^{(2)}, \ldots, I_k^{(r_k)}$. (We call an interval pair set *active* if it participates in the narrow and convergence operation.) Initially we have $I_1^{(1)} = I$. The result of applying the *narrow* operation in parallel until convergence to the r_k active interval sets yields the

following output:

- β^i, i in $1 \ldots r_k$, where β^i is a boolean variable representing the outcome of narrowing of the constraint set C using $I_k^{(i)}$. If narrowing fails, $\beta^i = 0$, otherwise, $\beta^i = 1$.
- $I_{k+1}^{(i)}$, i in $1 \ldots r_k$, where $I_{k+1}^{(i)}$ is the new interval set resulting from the narrowing of C using $I_k^{(i)}$.
- γ^i, i in $1 \ldots r_k$, where γ^i is a boolean variable indicating whether $I_{k+1}^{(i)}$ is a solution ($\gamma^i = 1$) or not ($\gamma^i = 0$). If $\beta^i = 0$, γ^i is irrelevant.

{The break statement returns control to the exit of the repeat-until loop.}
{ r_k is the number of active interval vectors in the kth iteration}
$k \leftarrow 1$; $r_k \leftarrow 1$; *work_completed* $\leftarrow$ false

while not *work_completed* (i.e., $r_k \neq 0$) do
  compaction_possible $\leftarrow$ false
  for all *active_interval_vectors* in parallel do
    repeat
      narrow
      if *narrow_fails* then *compaction_possible* $\leftarrow$ true; break end if
      intersection
      if *intersection_fails* then *compaction_possible* $\leftarrow$ true; break end if
    until *convergence* end repeat
    if *narrow_succeeds* and *intersection_succeeds* then
      if *solution* then *output_solution*; *compaction_possible* $\leftarrow$ true end if
    end if
    if *compaction_possible* then $r'_k \leftarrow$ *compact*(r_k) end if
    if $r'_k = 0$ (i.e., *spliting is not possible*)
     then *work_completed* $\leftarrow$ true; $r_{k+1} \leftarrow 0$ else $r_{k+1} \leftarrow$*split_and_copy*(r'_k) end if
  end for
  $k \leftarrow k + 1$
end while

Figure 3: Algorithm for the Data-Parallel Solver

For space and processor efficiency reasons, the data-parallel algorithm has to compact the $I_{k+1}^{(i)}$ towards the left-most processor sets by eliminating the $I_{k+1}^{(i)}$'s whose $\beta^i = 0$. This is done by a simple parallel prefix sum operation. Once the solutions (i.e., $I_{k+1}^{(i)}$'s such that $\gamma^i = 1$ and $\beta^i = 1$) are processed, the respective processor sets are made available by making the corresponding β^i's equal to 0. Let β denote the vector $(\beta^1, \beta^2, \ldots, \beta^{r_k})$. The compaction is achieved by the following steps:

---

if $\beta^i = 1$ then $\delta \leftarrow$ *prefix-sum*(β) (i.e., $\delta^i \leftarrow \sum_{j=1}^{i} \beta^j$);
$I^{(\delta^i)} = I^{(i)}$
$r'_k \leftarrow$ *reduce_add*(β) (i.e., $r'_k \leftarrow \sum_{j=1}^{r_k} \beta^j$);

Note that the above steps compute the value r'_k to be the number of the active interval sets that were successful. The split and copy phase essentially doubles the active interval sets. Let $v^{(i)}$, i in $1 \ldots r'_k$, be the variable to split for the r'_k interval sets. One strategy is to choose the $v^{(i)}$ such that the difference between $u_{v^{(i)}}$ and $l_{v^{(i)}}$ is minimum. This is determined by the parallel *prefix-min* operation. Let v denote the vector $(v^{(1)}, v^{(2)}, \ldots, v^{(r'_k)})$. The steps in the algorithm *split_and_copy* are shown below. That algorithm returns the new value of r for the $k+1$'st iteration of the while loop in Figure 3.

---

for all i in $1 \ldots r'_k$ in parallel do
  $\rho^i \leftarrow I^i(u_j - l_j), j \in 1 \ldots n$
  $v^i \leftarrow$ *prefix-min*(ρ^i) i.e., $v^i \leftarrow j | \rho^i_j$ is minimum and is $\neq 0$, $j \in 1 \ldots n$
  $Temp^i \leftarrow I^i$; $I^i \leftarrow$ split_strategy_first(v^i,I^i); $I^{r'_k+i} \leftarrow$ split_strategy_second(v^i,$Temp^i$)
end for
return $2 * r'_k$

---

3 Markovian Analysis

Given a transition probability matrix T of the nodes in the flowgraph, and the vector S denoting the time-variables of the nodes in the flowgraph [3, 9], the time-formula for the average execution time is given by:

$$(first_row((I - T)^{-1})) \; . \; S$$

where I is the identity matrix.

A Maple [3] program computing the time-formula is presented in Figure 4. The probabilities considered are as follows:

- p_n, the probability that *narrow* succeeds,
- p_i, the probability that *intersection* succeeds,
- p_c, the probability that *convergence* succeeds,
- p_r, the probability that *solutions (results)* are found,
- p_s, the probability that *split* is possible.

The computed time formula is:

$$t := \frac{narrow}{1-p_n\ p_i+p_i\ p_n\ p_c+p_n\ p_i\ p_s-p_n\ p_i\ p_s\ p_c-p_s} + \frac{p_n\ intersection}{1-p_n\ p_i+p_i\ p_n\ p_c+p_n\ p_i\ p_s-p_n\ p_i\ p_s\ p_c-p_s} +$$
$$\frac{p_n\ p_i\ convergence}{1-p_n\ p_i+p_i\ p_n\ p_c+p_n\ p_i\ p_s-p_n\ p_i\ p_s\ p_c-p_s} + \frac{p_i\ p_n\ p_c\ solution}{1-p_n\ p_i+p_i\ p_n\ p_c+p_n\ p_i\ p_s-p_n\ p_i\ p_s\ p_c-p_s} +$$
$$\frac{(p_n\ p_i\ p_c\ p_r-p_n\ p_i+1)compact}{1-p_n\ p_i+p_i\ p_n\ p_c+p_n\ p_i\ p_s-p_n\ p_i\ p_s\ p_c-p_s} + \frac{split}{1-p_s} + \frac{p_s\ copy}{1-p_s}$$

[3] A symbolic computing sysem developed in the Department of Computer Science at the University of Waterloo, Waterloo, Ontario, Canada.

```
% identify nodes
A := 1; B := 2; C := 3; D := 4; E := 5; F := 6; G := 7; H := 8;
% input of a sparse markovian matrix
T := array(1..8,1..8,
          [(A,B) = p_n, (A,E) = 1 - p_n,
           (B,C) = p_i, (B,E) = 1 - p_i,
           (C,D) = p_c, (C,A) = 1 - p_c,
           (D,E) = p_r, (D,F) = 1 - p_r,
           (E,F) = 1,
           (F,G) = p_s, (F,H) = 1 - p_s,
           (G,A) = 1 ],
      sparse);
% vector specifying identifiers for time variables
S := array(1..8,
[narrow,intersection,convergence,solution,compact,split, copy, 0] );
% the identity matrix
I := array(1..8, 1..8, identity);
% evalm --- evaluate the matrix
m := evalm(I - T);
% linalg --- linear algebra package; inverse --- inverse of matrix
r := linalg[inverse](m);
% subvector --- appropriate subvector of matrix
v := linalg[subvector](r,1,1..8);
% dotprod --- dot product of two vectors
time := linalg[dotprod](v,S);
t := collect(time,
{narrow,intersection,convergence,solution,compact,split, copy});
```

Figure 4: Maple Program to Compute the Time-Formula

For example, with the probability substitutions

$$p_n = 1,\ p_i = \tfrac{2}{3},\ p_c = \tfrac{1}{2},\ p_r = 0, \text{ and } p_s = \tfrac{1}{2},$$

corresponding to the path $ABCABCDFGABEFH$ in Figure 4,

$$t := 3\,narrow + 3\,intersection + 2\,convergence + solution + compact + 2\,split + copy$$

4 Analysis of Processor Sets

The number of processor sets in the $(k+1)^{th}$ iteration of the algorithm in Figure 3 is given by the following finite-difference equation:

$$r_{k+1} = 2 * (r_k - Q(k) * r_k) = 2 * (1 - Q(k)) * r_k$$

$$r_1 = 1$$

where $Q(k) = Q_{fails}(k) + Q_{solns}(k)$, Q_{fails} being the fraction of processor sets whose interval sets fail, and Q_{solns} being the fraction of processor sets whose interval sets are

solutions. These are precisely the processor sets reclaimed during compaction. The split and copy phase doubles the processor sets whose inputs are not yet solutions.

Solving the above finite difference equation, we have:

$$r_k = 2^{k-1} * (1 - Q(k))^{k-1}, \quad \text{where} \quad 0 \leq Q(k) \leq 1$$

The maximum number of processor sets R that can operate in parallel is bound by the physical number of processors available. If the underlying machine has N processors [4], then

$$R = N \div max(m, n)$$

where the problem has m constraints and n variables. As long as $r_k \leq R$, all the r_k processor sets operate in parallel. However, if $r_k > R$, the number of passes [5] required (NP) to process the r_k processors sets is given by:

$$NP = \begin{cases} r_k \div R & \text{if } r_k \ mod \ R = 0 \\ 1 + (r_k \div R) & \text{otherwise} \end{cases}$$

In this case, each processor set holds more than one interval set. The ith interval set is stored within the processor set $i \ mod \ R$, $1 \leq i \leq r_k$). So each processor set has NP or $NP - 1$ interval sets.

Assume Q to have a uniform distribution between 0 and 1. The number of processor sets r_k in the kth iteration is shown in Figure 5a. The number of passes required to process the r_k processor sets is shown in Figure 5b. For a fixed R, the number of passes required as the number of constraints increases is shown in Figure 5c. For a fixed problem size, the number of passes required as R increases is shown in Figure 5d.

5 Results

The sequential time is obtained using the sequential implementation of the algorithm on the SGI Onyx (with one processor). The parallel time corresponds to the data-parallel implementation on the MASPAR MP2 with 4096 processors. It should be noticed that a single processor on the MASPAR is much slower compared to the SGI Onyx.

Problem	*m*	*n*	*Solutions*	*Seq. Time*	*Parallel Time*	*Speed-up*
8-Queens	140	72	92	6.1s	1.4s	4.4
9-Queens	180	90	352	29.2s	2.7s	10.8
Perm-6	15	6	720	1.0s	0.25s	4.0
Perm-7	21	7	5040	7.0s	0.43s	16.2
6x6 Job-Shop	793	617	1	0.85s	0.47s	1.8

[4] In our case $R = 4096$

[5] A pass consists of processing 4096 constraints (or variable intervals)

6 Final Remarks

In this paper we presented the design and implementation of a deterministic constraint language based on intervals. This is believed to be a pioneer effort in the area. Usage of data-parallelism is ideal for processing a large number of constraints. In addition, the split operation can also be carried out in parallel.

There are several possible extensions of this work. Among them we should mention the following:

1. Design and implement an optimizer (preprocessor) that analyzes the variables and the constraints and place them in processors so as to globally reduce communication costs.

2. Introduce non-determinism by including further OR-parallelism to process disjunctive constraints.

3. Study the performance of the proposed approach by processing the most time consuming problems such as the job-shop scheduling [2].

References

[1] G. E. Blelloch, *Programming Parallel Algorithms*, available through WWW (http://web.scandal.cs.cmu.edu/www/nesl.html)

[2] J. Carlier and E. Pinson, *An Algorithm for Solving the Job-Shop Problem*, Management Science, vol. 35, no. 2, February, 1989, pp. 164-176.

[3] J. Cohen and A. Weitzman, *Software Tools for Micro-analysis of Programs*, Software-Practice and Experience, vol. 22, no. 9, September, 1992, pp. 777-808.

[4] T. Hickey, *CLP(F) and Constrained ODEs*, Workshop on Constraint Languages and their use in Problem Modelling, International Logic Programming Symposium, 1994.

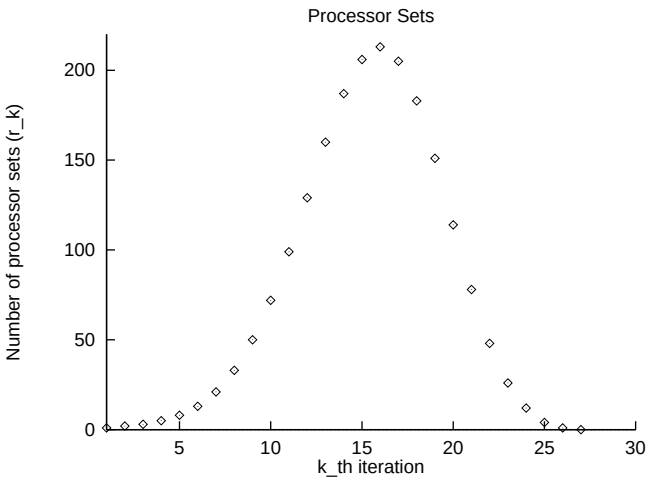
[5] *MasPar Parallel Application Language (MPL) User Guide & Reference Manual*, MasPar Computer Corporation, 1993.

[6] R. E. Moore, *Interval Analysis*, Prentice Hall, 1966.

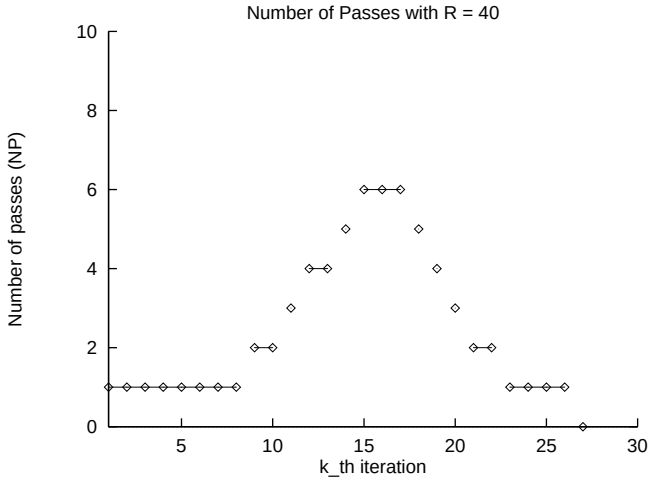
[7] W. J. Older, and A. Vellino, *Constraint Arithmetic on Real Intervals*, *Constraint Logic Programming: Selected Research*, edited by F. Benhamou and A. Colmerauer, MIT Press, 1993.

[8] B.M. Tong, and H.F. Leung, *Implementation of a Data-Parallel Concurrent Constraint Programming System*, First International Symposium on Parallel Symbolic Computation (PASCO'94), 1994, pp. 382-393.

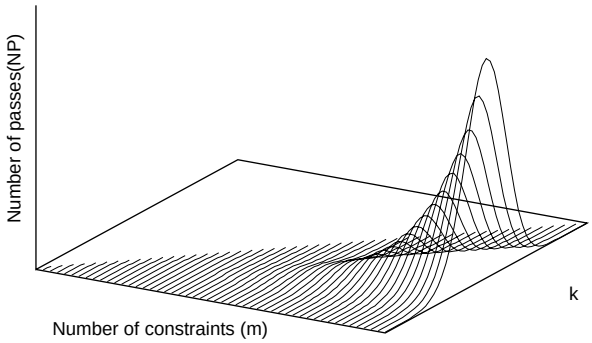
[9] K. S. Trivedi, *Probability & Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, 1982.



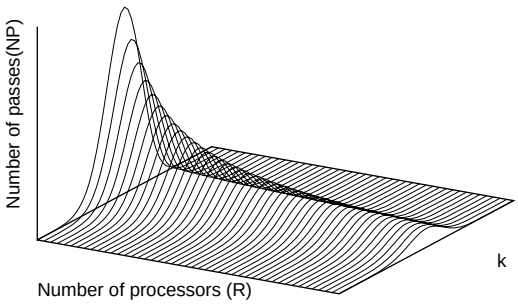
(a)



(b)



(c)



(d)

Figure 5: (a) Number of Processor Sets
(b) Number of Passes
(c) Number of Passes vs. Number of Constraints
(d) Number of Passes vs. Number of Processors