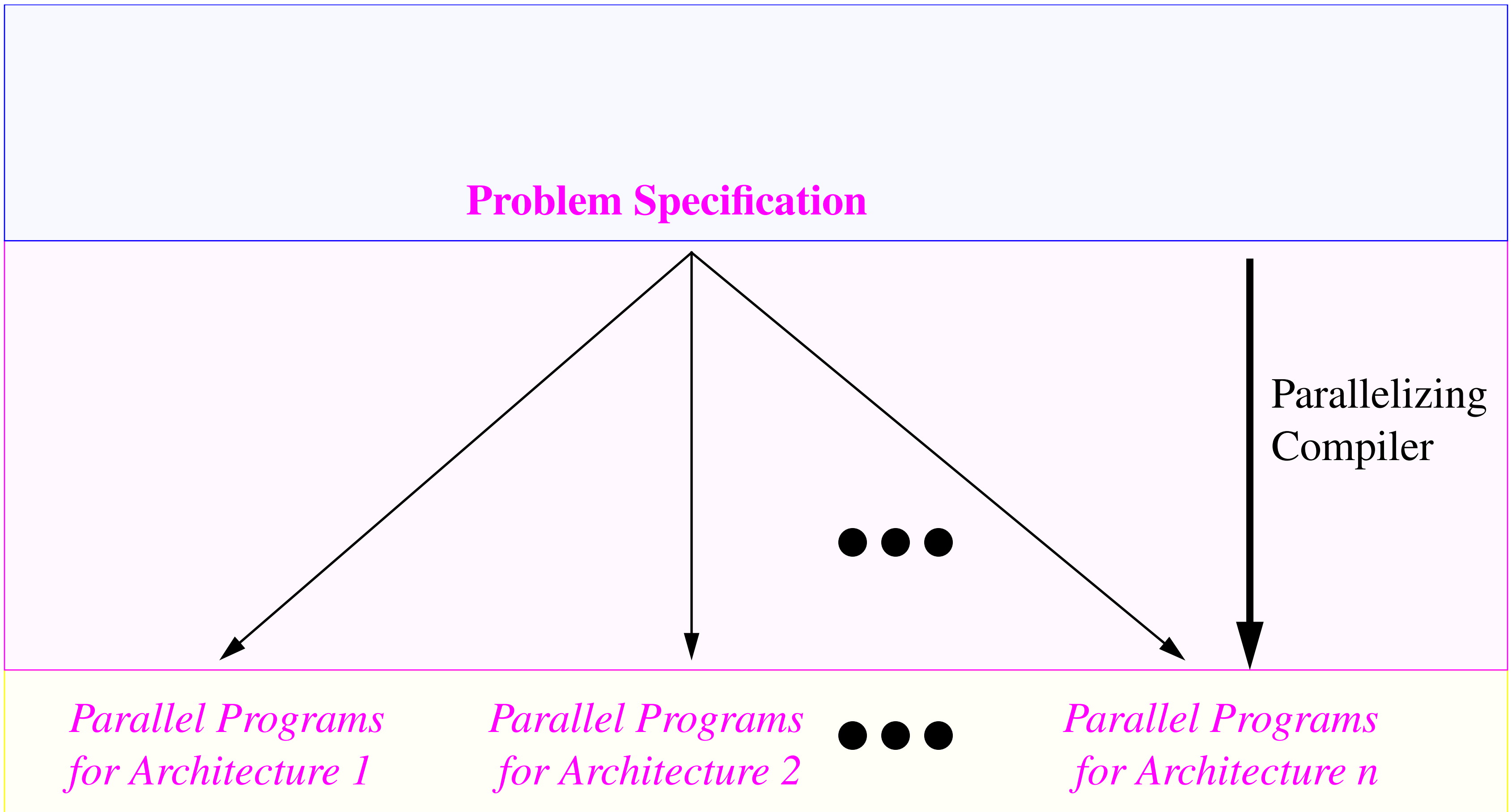
Compilation of Constraint Systems to Parallel Procedural Programs

Ajita John

ajohn@cs.utexas.edu
http://www.cs.utexas.edu/users/ajohn

Advisor: Prof. J.C. Browne

Application Oriented Declarative Specification of Parallel Programs



Goal

Declarative Specifications ---> High Performance

A representation for problem specification which

- places minimum burden on user
    - no explicit parallelism in specification
- allows extraction of all forms of parallelism
- allows reuse of components

Constraints:

attractive choice for a representation for problem specification

Why ?

Goal

Extraction of <u>efficient</u> parallel structures from constraints

Contributions

1. Programming system based on constraints for matrix computation.

    - *Hierarchical Type System*
    - *Programming Representation*
    - *Modular Structures for Constraint Systems*
    - *Basic Compilation Algorithm*
    - *Extension for Cyclic Constraints*
    - *Extraction of Task and Data Parallelism*

2. Generation of multiple programs from single specification.

3. Generation of effective/complete programs.

4. Translation to multiple parallel architectures.

5. Development of applications yielding high performance.

Overview

1.Introduction to Constraints

2.Related Work

3.Our Approach

4.Extraction of Parallelism

Compiler

5.Examples and Performance Results

6.Compiler Enhancements: Cyclic Constraints

7.Summary

8.Future Work

A constraint is a relationship between a set of variables.

E.g. 1

C == (F-32) * 5/9

E.g. 2

F == (9/5) * C + 32

== *is equality as opposed to assignment*

$1 \equiv 2$

Constraint Programming vs. Imperative Programming

Constraint Programming	Imperative Programming
C == (F-32) * 5/9	Program 1 C = (F-32) * 5/9
	Program 2 F = 32 + (9/5) * C

Related Work **(Representative Examples)**

Constraint Programming

- *Thinglab, Kaleidoscope*
- *Consul*
- *CCP, Oz*

Parallel Programming

Imperative Programming

- *HPF*
- *CC++*

Equational Programming

- *Unity*

Logic Programming

- *PCN*
- *Strand*

Functional Programming

- *SISAL*

Definitions

Constraint specification = *Set of relationships between variables V of a system*

Input set = *Set of input variables,* $I \subseteq V$

Constraint program = *Constraint specification* + *Input set*

Dependence Graph = Parallel computation structure, computes values for $O \subseteq V - I$

Granularity = Ratio of computation to communication

Task Parallelism = Parallelism in different computations,

$x=f(\ldots), \;\; y=g(\ldots)$

Data Parallelism = Parallelism in computations for partitions of a regular data structure,

$x[1]=f(\ldots), \;\; x[2]=f(\ldots)$

Approach

Constraint program (constraint specification & input set)

COMPILE

Dependence graph

MAP

Sequential and parallel C programs generated for

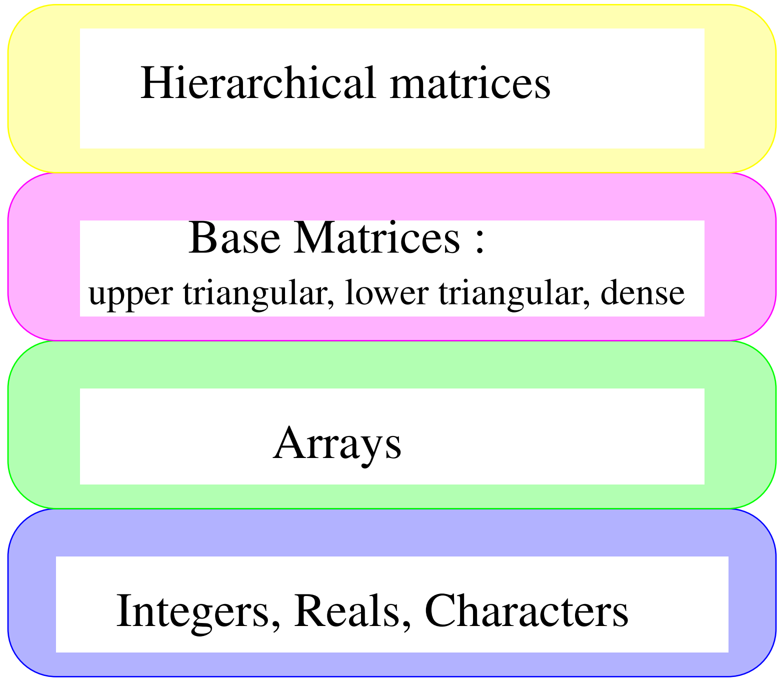
CRAY J90, SPARCcenter 2000, Sequent,

PVM,

MPI (under development)

CODE - Browne, Newton et. al.

Hierarchical Type System



- Constructs new types at higher levels from existing types at lower levels.
- Provides a tool for controlling granularity.
- Each high level type may incorporate domain information in its definition.
- Different efficient algorithms can implement operations depending on the type.
- We have a translator for specifications over a rich matrix type set to parallel programs.

Operator Expressions

- the building blocks for constraints

Expressions involving:

1. Scalar operators, pure function invocations

   E.g. $a + b * sqr(c)$

2. Matrix operators

   E.g. $M_1 * M_2 - M_3$

3. Indexed operators: <op> FOR (<index> <b1> <b2>) <expression>

   E.g. $+ \ FOR \ (i \quad 1 \quad 5) \quad A$[i]

   Equivalent to $A[1] + A[2] + A[3] + A[4] + A[5]$

Constraint Representation

Simple Constraints: relational operators (<, <=, >, >=, ==, !=) over operator expressions

E.g. $a + b > c$

$M_1 * M_2 == M_3$ | only == for matrices

Propositional Operators: NOT, AND, OR, on constraints:

E.g. *a + b > c AND a == 5*

Constraints over indexed sets:

AND/OR FOR (<index> <b1> <b2>) { $C_1, C_2, \ldots, C_n$ } C_i is a constraint

E.g. *AND FOR (i 1 5) { A[i] == A[i-1]+1, B[i] == A[i] }*

- represents 10 constraints

Constraint Modules: encapsulation of constraints over parameters

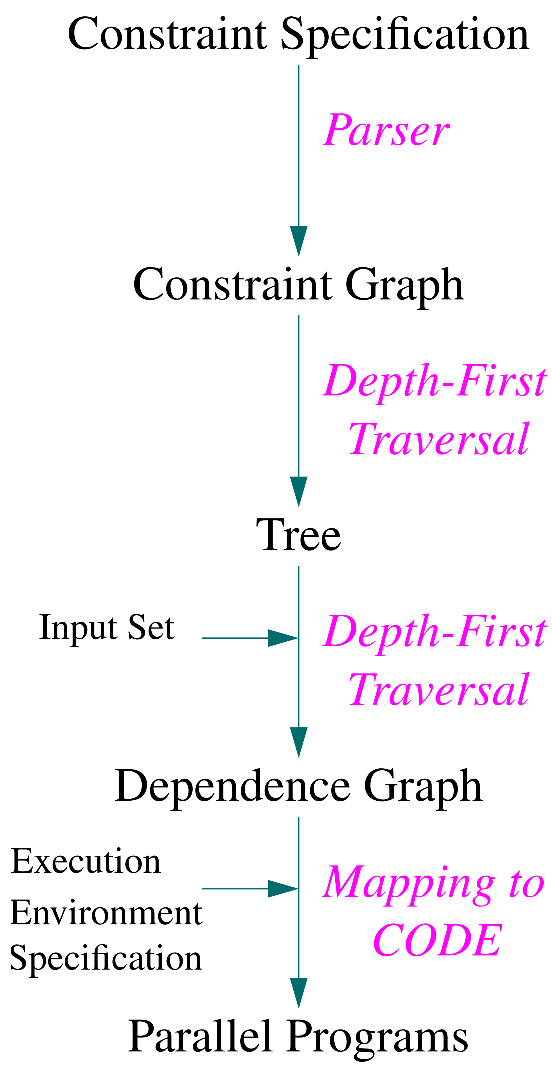
E.g. *EquivalentTemp(F,C)*

Quadratic Equation Solver

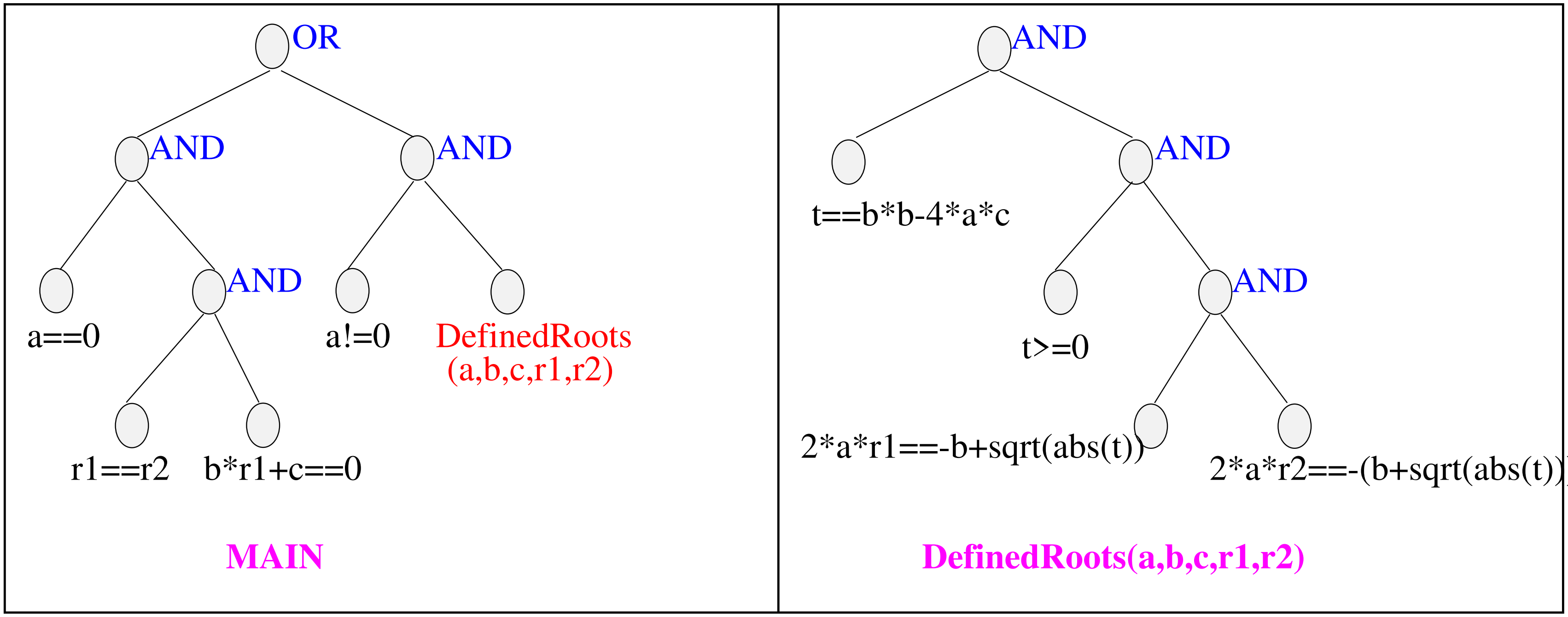
$$2ax = -b \pm \sqrt{b^2 - 4ac}$$

```
/*  Main */
        a == 0  AND  r1 == r2  AND  b*r1 + c == 0

        OR

        a != 0 AND DefinedRoots(a, b, c, r1, r2)
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
/* Constraint module */

  DefinedRoots(a, b, c, r1, r2)
        t == sqr(b)- 4*a*c
        AND t>=0
        AND 2*a*r1==-b+sqrt(abs(t))
        AND 2*a*r2==-(b+sqrt(abs((t))))
```

Compilation



Constraint Graph for Quadratic Equation Example

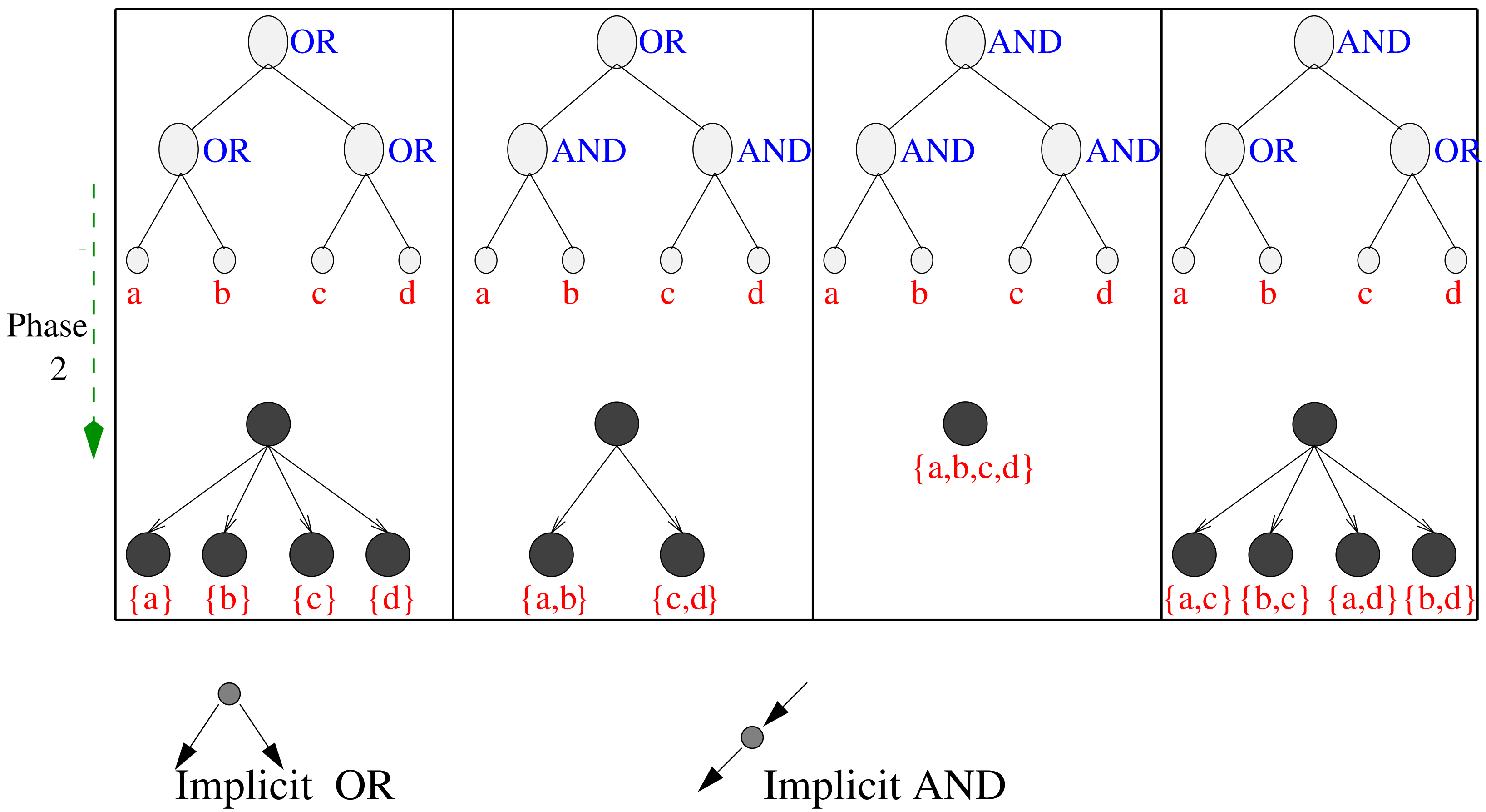


Assign a node for

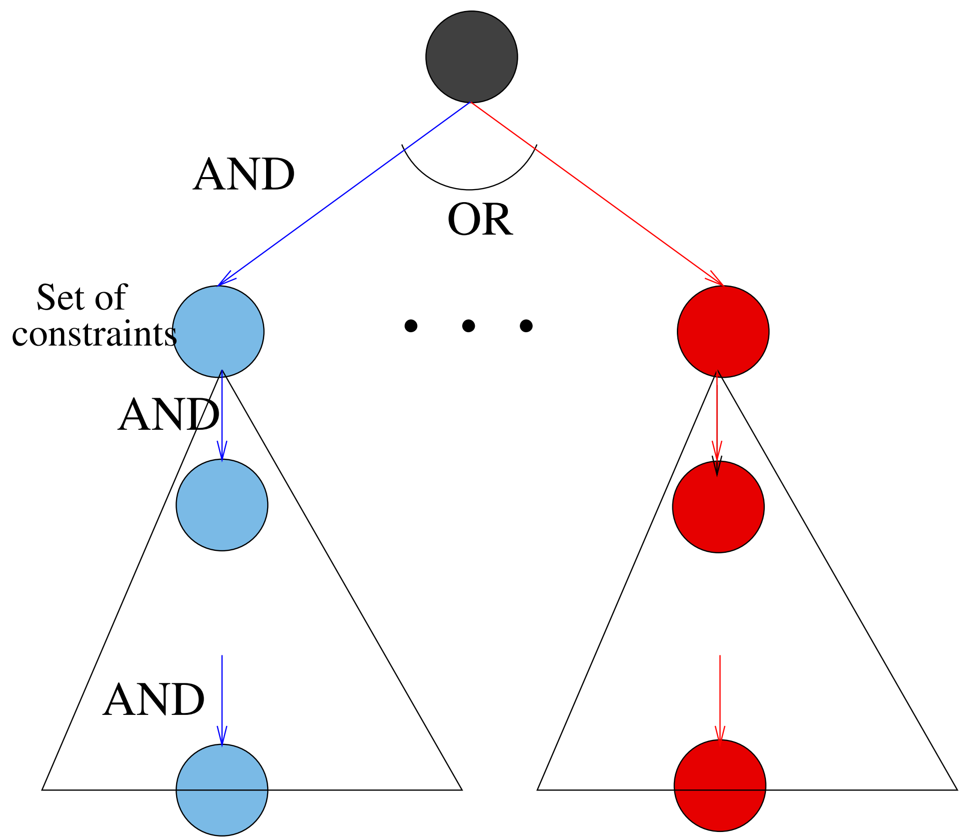
- every simple constraint
- every propositional operator, AND, OR, NOT
- call to a constraint module
- indexed set

Depth-First Traversal (1)
Constraint Graph ---> Tree

generates a reduced tree without AND & OR nodes

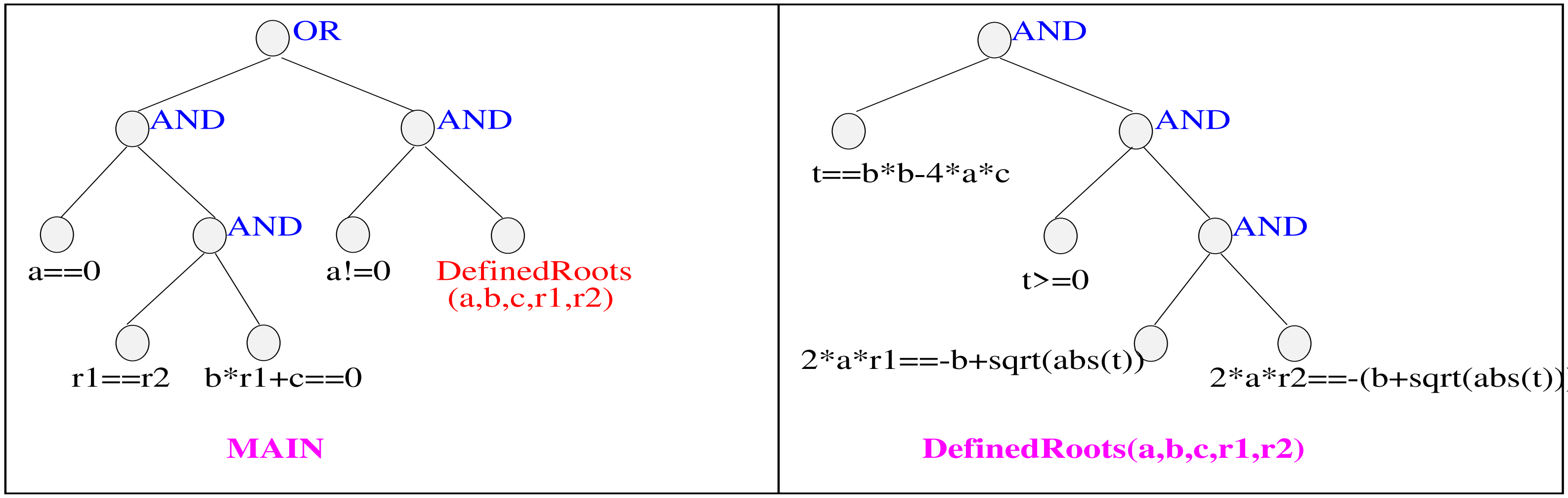


Tree from Depth-First Traversal (1)

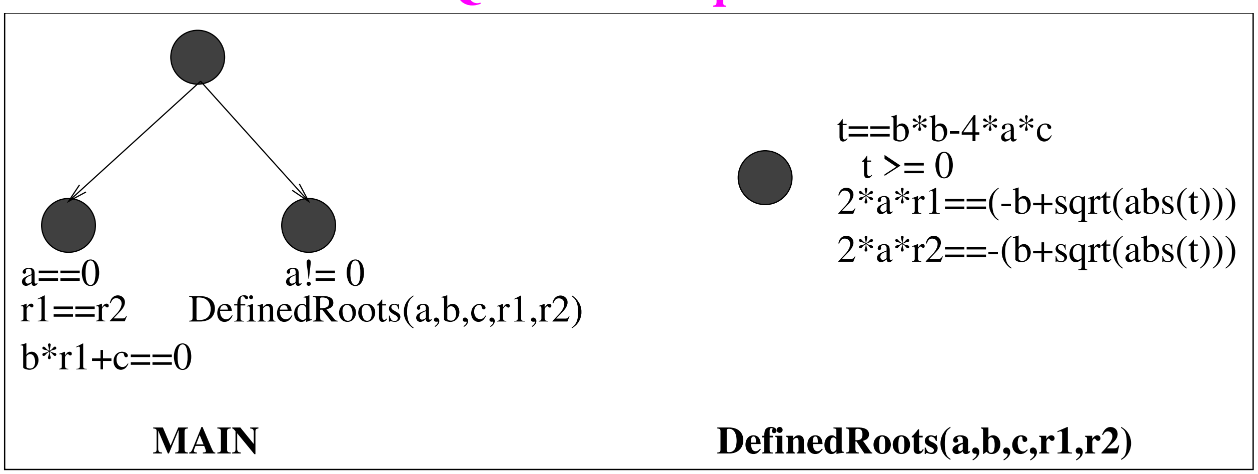


Each path may yield a solution.

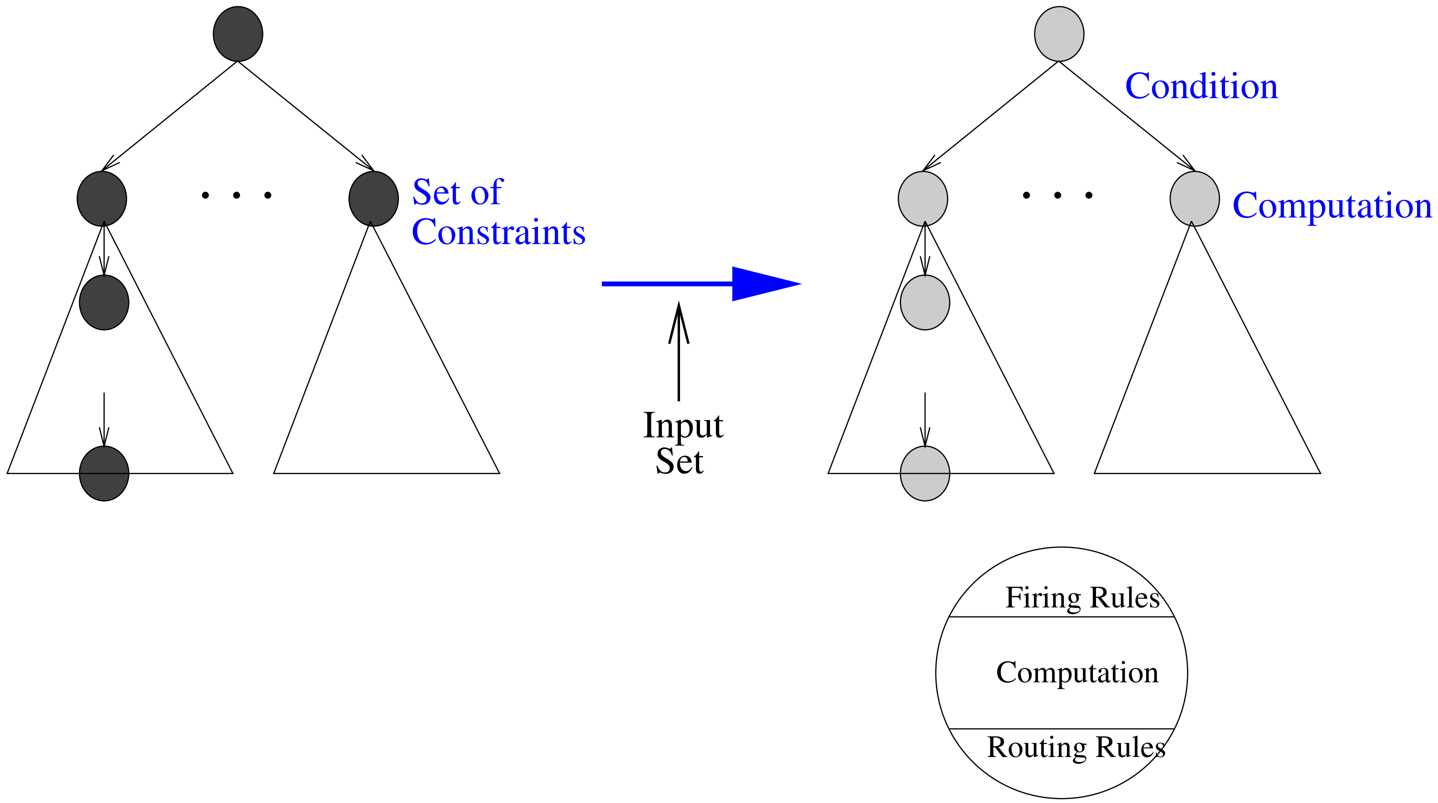
Constraint Graph for Quadratic Equation Solver



Tree for Quadratic Equation Solver



Depth-first Traversal (2)
Tree ------> Dependence Graph



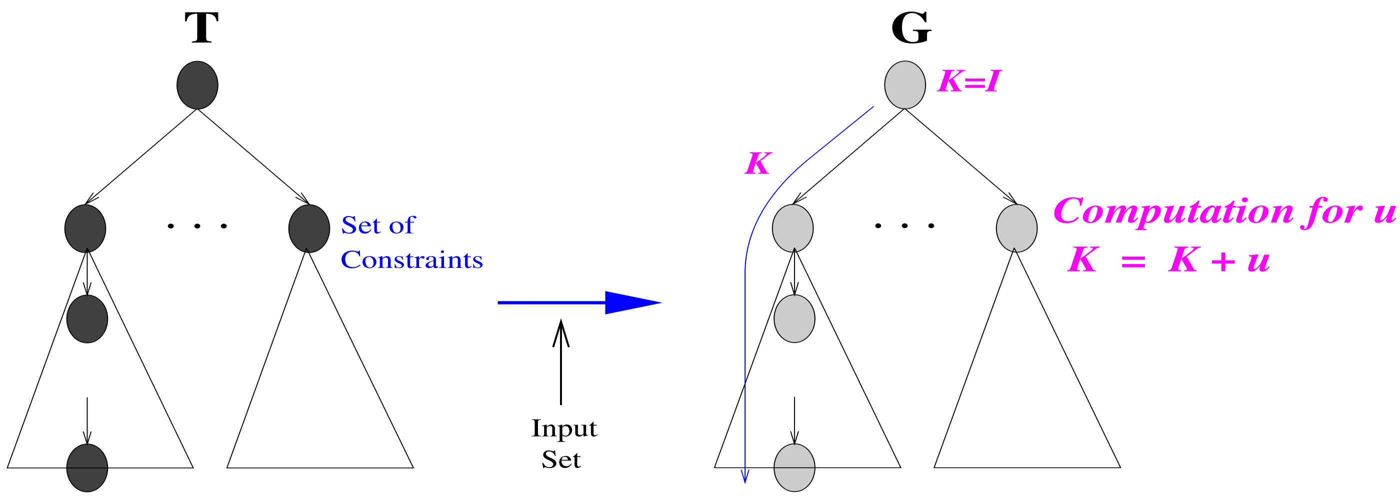
Dependence Graph

Nodes are computational units.

Nodes execute dependent on certain *conditions* and the availability of data.

Resolution of Constraints

A *known-set* *K* is propagated to each node in *G*.
(Every variable in *K* is a *known* at a node; every other variable is an *unknown*.)

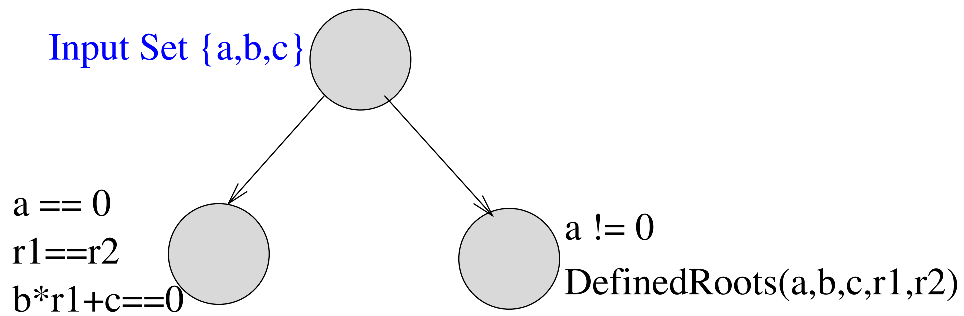


A simple constraint *C* is *resolved* if it can be classified as

- a condition (*C* involves no unknowns)
- a computation (*C* involves an equality & a single unknown *u*)
  - computation for *u* is extracted from *C*, *u* is added to *K*

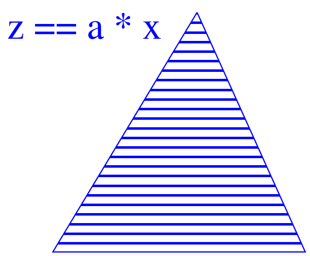
else *C* is *unresolved*.

No deletions are performed on *K* => *Single assignment variables*



a == 0 ---> Condition {a==0}

b*r1 + c==0, r1==r2 ---> Computation { r1=-c/b, r2=r1}



z == a * x ---> Unresolved

Unresolved Constraints

- Unresolved constraints are propagated down all paths of tree.
- Types of unresolved constraints:

    1. Simple constraints with one or more unknowns

    2. Unresolved calls to constraint modules

    3. Unresolved indexed sets

- Paths ending in unresolved constraints at leaves:

    * corresponding path in dependence graph can be deleted (1-3)

    * change input set to resolve the constraints (1-3)

    * use iterative methods (1-3)

    * invoke solvers such as MATLAB (1)

Resolution of Constraint Modules

Constraint Modules can be compiled to procedures

input and output parameters determined by the known set at the point of call

E.g. Constraint Module *DefinedRoots(a, b, c, r1, r2)* can be compiled to

three procedures:

- *DefinedRoots(a, b, c, r1, r2)*
- *DefinedRoots(a, b, r1, c, r2)*
- *DefinedRoots(a, b, r2, c, r1,)*

Red: Inputs, Blue: Outputs

Resolution of Constraints over Indexed Sets

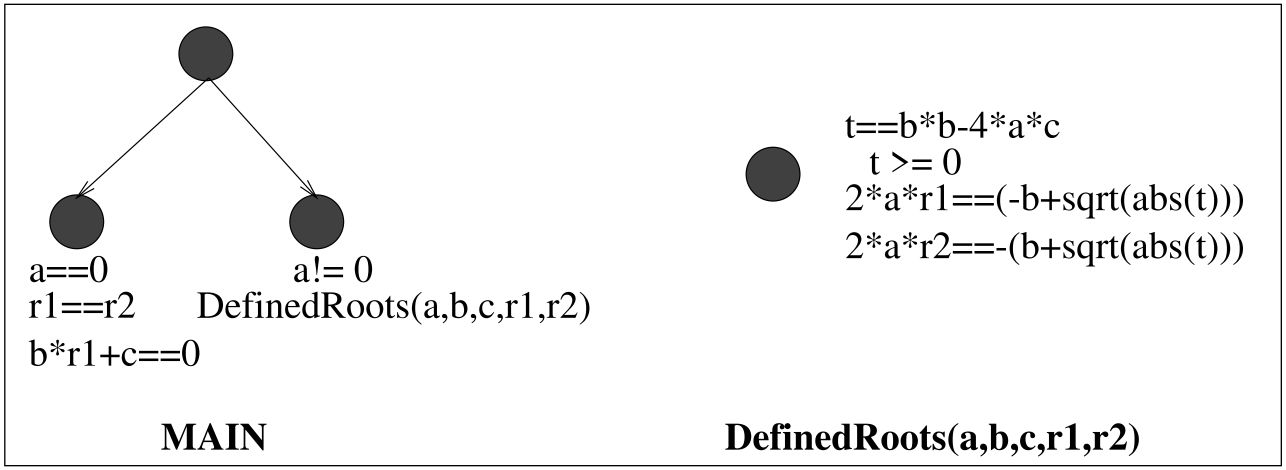
Constraints over Indexed Sets can be compiled to loops
over computations extracted from constraints

```
AND FOR ( i  1  10 )  {
      A[i]  +  A[i-1]   ==   0,
      A[i-1] - B[i]     ==   2
}

Input A[0]
```

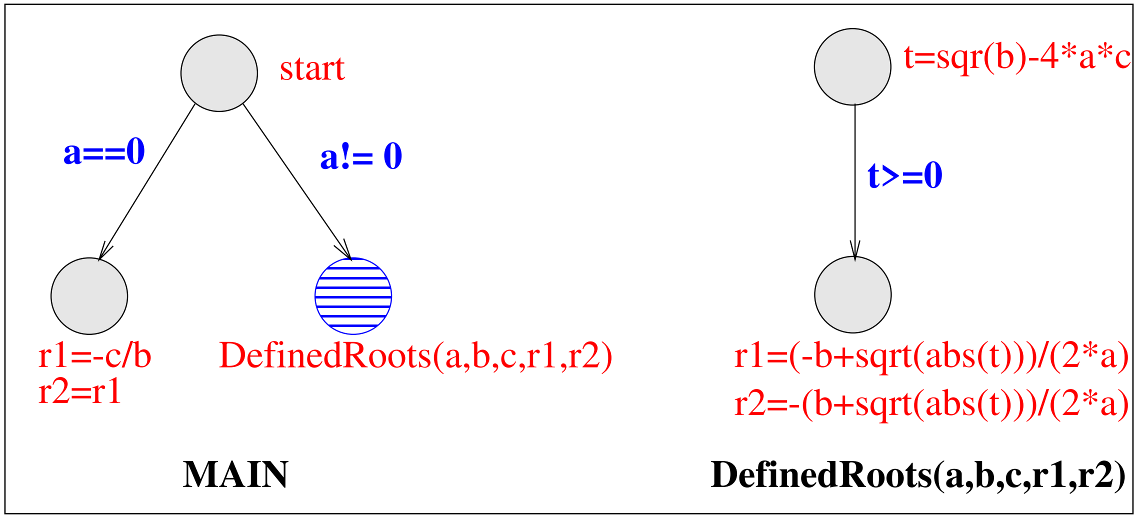
→

```
for ( i=1;   i<=10;  i++)  {
      A[i]  =  -A[i-1];
      B[i]  =   A[i-1] - 2;
}
```

Resolution of Constraints for Quadratic Equation Solver

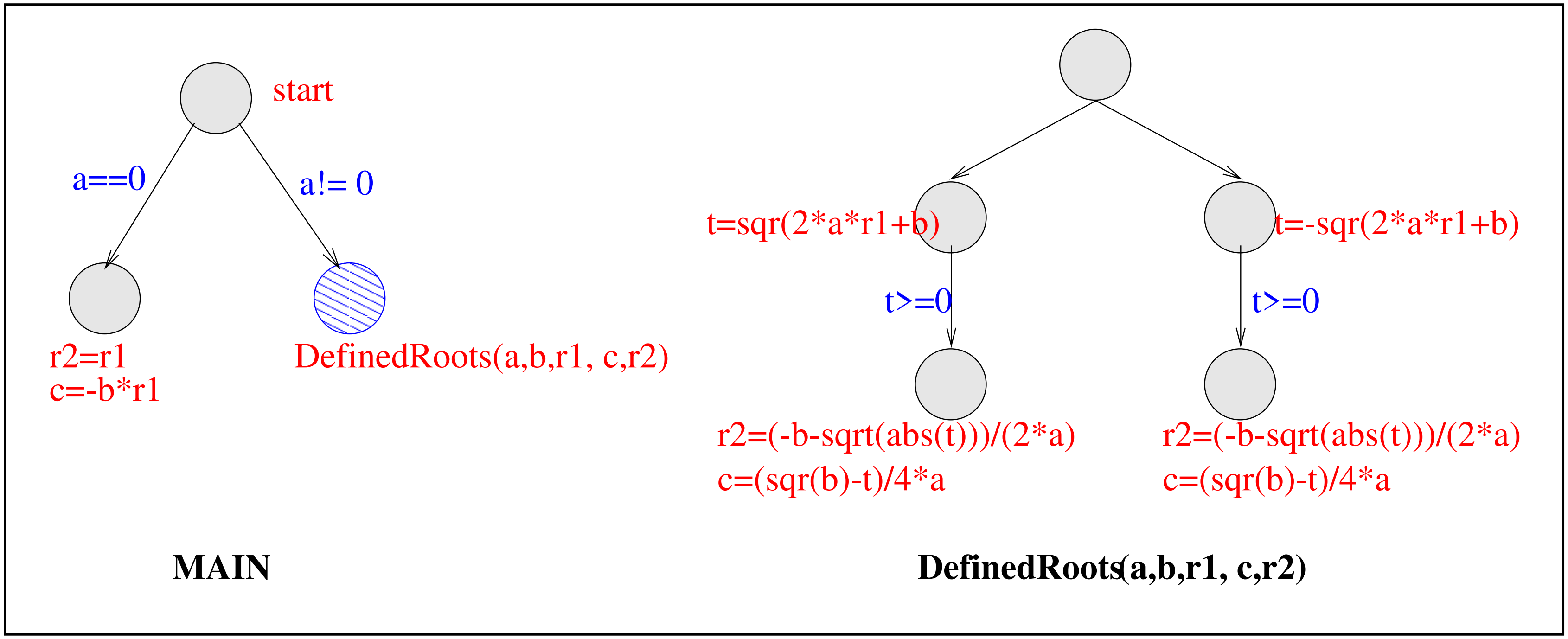


Inputs {a, b, c} , Outputs {r1, r2}



Dependence Graphs for Quadratic Equation Example

Inputs: {a, b, r1}, Outputs: {r2, c}



Properties of Compilation

Single assignment system

Extraction of effective (1 solution) and complete (all solutions) programs

Extraction of Parallelism

Types of Parallelism

AND-OR

Task

Data

Parallelism extracted by Compiler

1. in different paths of dependence graph (OR, Task)

2. between computations at a node (AND, Task)

Implemented

3. in operations within a computation (Task)

   $$\underline{M_1 * M_2} + \underline{M_1 * M_3}$$

4. in iterations of loops (Data)

Future

5. in primitive operations over base types (Data)

Control over Granularity

Modules

Functions

Hierarchical Types

Data Partitioning in Scalar Operations

Execution Environment Specification

Block Odd Even Reduction System (BOER)

Solution of $A * x == d$, where

$$A \;==\; \begin{bmatrix} B & C & 0 & 0 & . & . & . & 0 & 0 & 0 \\ C & B & C & 0 & . & . & . & 0 & 0 & 0 \\ 0 & C & B & C & . & . & . & 0 & 0 & 0 \\ . & . & . & . & & . & & . & . & . \\ 0 & 0 & 0 & 0 & & & & C & B & C \\ 0 & 0 & 0 & 0 & & & & 0 & C & B \end{bmatrix}$$

B, C are block matrices of order $n >= 2$

A is of order $M*n$, where $M = 2^k-1$, $k >= 2$

Constraint Program for BOER

1 → $BP[k-1] * x[2^{k-1}] == dP[2^{k-1}][k-1]$ /*Single-Solution*/

AND

2 → AND FOR (j 1 k-1) { /*Reduction*/

$2 * CP[j-1] * CP[j-1] == BP[j] + BP[j-1] * BP[j-1]$

$CP[j-1] * CP[j-1] - CP[j] == 0$

AND FOR (i 0 2^{k-j}-2) {

$CP[j-1] * (dP[f_1(i,j)][j-1] + dP[f_2(i,j)][j-1])$

$== dP[i*2^j][j] + BP[j-1] * dP[i*2^j][j-1]$ }}

AND

3 → AND FOR (j k-1 1) { /*Back-Substitution*/

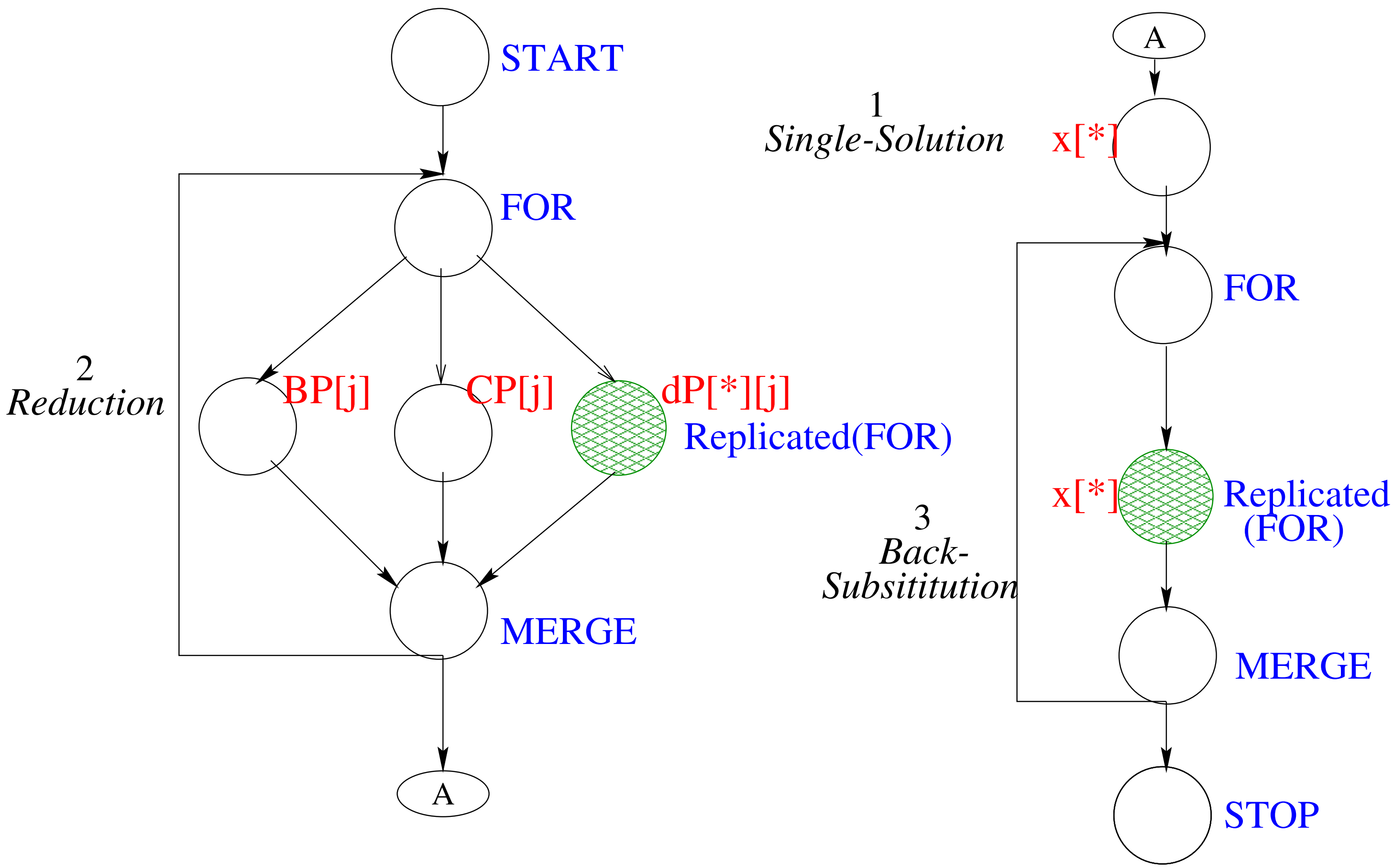
AND FOR (i 0 2^{k-j}-1) {

$CP[j-1] * (x[(i+1)*2^j] + x[i*2^j])$

$== dP[f_3(i,j)][j-1] - BP[j-1] * x[(i+1)*2^j-2^{j-1}]$ }}

Inputs=
{ BP[0],
CP[0],
dP[0][*]
}

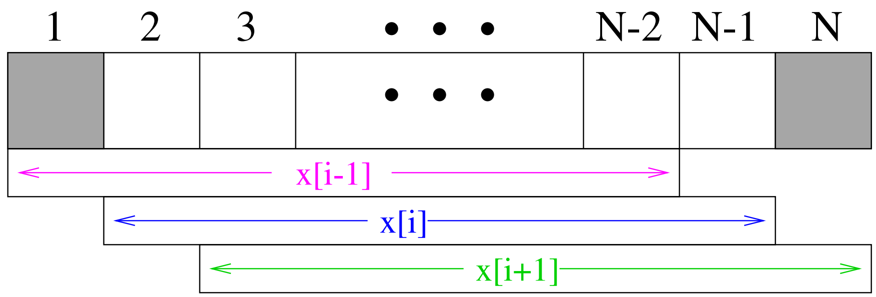
Dependence Graph for BOER System



Performance Results

BOER Program

Resolution of Cyclic Constraints through Iterative Methods



CONSTRAINTS	*COMPUTATIONS*
AND FOR (i 2 N-1) { *x[i-1] + x[i+1] == x[i] }* *Inputs: x[1] and x[N]* x[1] + x[3] == x[2] x[2] + x[4] == x[3] x[3] + x[5] == x[4] ...	*for (i 2 N-1) {* *x[i] = x[i-1] + x[i+1] }*

Issues

1. Selection of the term to be computed

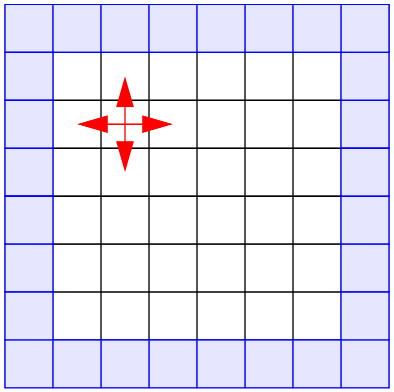
2. Checking convergence

3. Relaxation Methods

4. Mapping single assignment to mutable variables

Resolution of Cyclic Constraints

Laplace Equation for a 4-point stencil



Constraint specification
Inputs: Boundary Elements

AND FOR (i 2 N-1) {
AND FOR (j 2 N-1) {
*x[i-1][j] + x[i+1][j] == 4*x[i][j] - x[i][j-1] - x[i][j+1]   }}}*

Relaxation: Extracted Code

Jacobi Relaxation for Laplace Equation

```
while (check_accuracy(x,old_x)) {
  copy_values(old_x,x);
  for (i 2 N-1) {
    for (j 2 N-1) {

      x[i][j] = (old_x[i-1][j] + old_x[i+1][j] + old_x[i][j-1] + old_x[i][j+1])/4 ;
} } }
```

Gauss-Seidel Relaxation for Laplace Equation

```
while (check_accuracy(x,old_x)) {
  copy_values(old_x,x);
  for (i 2 N-1) {
    for (j 2 N-1) {

      x[i][j] = (x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1])/4 ;
} } }
```

Performance Results for Laplace

Summary

Constraint systems & appropriate sets of input variables

$\downarrow$

generalized dependence graphs

$\downarrow$

efficient parallel programs.

Granularity is controlled through
- modules,
- functions,
- hierarchical types.

Both task and data parallelism can be targeted.

Constraints:
attractive choice for a representation for problem specification

Why ?

Declarative specification

Compiler has great freedom in the generation of efficient parallel structures

Extraction of different programs from same constraint specification

Generation of complete or effective programs

Conclusion

Feasibility of compilation of efficient parallel procedural programs from constraint programs (at least for matrix computations) has been established.

Future Work

Short Term

- Incorporate powerful data partitioning mechanisms.
- Complete development of an execution environment specification.
- Relax restrictions on indexed sets

Long Term

- Extraction of algorithms for applications where algorithms specification is difficult.
- How to write “good” specifications ?
- Which path should be selected in effective programs ?
- Investigate other application domains.
- Applications:
    1. Boundary Matching Problems for Decomposed Domains
    2. Data Mediation
    3. Back Tracing for Circuits

Publications

Ajita John, J.C.Browne. Compilation of Constraint Specifications to Procedural Parallel Programs: Resolution of Cyclic Constraints. International Journal of Parallel Programming, 1997, To Appear.

Ajita John, J.C.Browne. Compilation to Parallel Programs from Constraints. Third International Conference on High Performance Computing, December, 1996.

Ajita John, J.C.Browne. Extraction of Parallelism from Constraint Specifications. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) , Vol. III, pages 1501-1512, August, 1996.

Ajita John, J.C.Browne. Compilation of Constraint Systems to Procedural Parallel Programs. Lecture Notes in Computer Science, No. 1239, Springer-Verlag, pages 501-518, Workshop on Languages and Compilers for Parallel Computers, August, 1996.