Experiments in Extraction of Coarse Grain Parallelism from Constraint Programs

Ajita John and J.C. Browne
Dept. of Computer Science
University of Texas, Austin, TX 78701
email: {ajohn,browne}@cs.utexas.edu

TR95-1 May 1, 1995

Abstract

This paper reports on experimental research on extraction of coarse grain parallelism from constraint systems. Constraint specifications are compiled into task level procedural parallel programs in C. Three issues found to be important are: (i) inclusion of operations over structured types as primitives in the representation, (ii) inclusion of modularity in the constraint systems, and (iii) use of functions in the constraint representation. The role of these issues is described. The compilation process focuses on determining the parallel structure defined by the constraints and creates a parallel program in the format of the CODE 2.0 parallel programming environment. The compilation algorithm is described. An example compilation of a constraint system defining a simple numerical algorithm to a parallel C program is given and performance results are reported. Directions for future enhancement of the compilation process are suggested.

1 Introduction and Background

Interest in parallel programming has sparked interest in alternative representations for expressing computations. It is safe to say that there is as yet no widely accepted representation from which efficient parallel programs can be compiled for substantially different parallel execution environments. This paper reports on experiments on compilation of procedural parallel programs from constraint specifications of algorithms. The key elements of the approach include: (i) inclusion of modularity in the constraint specifications, (ii) allowing functions in constraints and (iii) extension of the type system to include structured data elements as primitives. These extensions to constraint specifications, together with recognition that assignment of values to any consistent set of variables in a constraint system induces a data flow graph from which a parallel program is readily obtained, has made it surprisingly straightforward to compile procedural parallel programs from constraint specifications. The target language is CODE 2.0 [14] which represents parallel computation structures as generalized dependence graphs. Performance results from a simple matrix algorithm are given.

1.1 Constraint Systems as a Programming Language

Representing an algorithm or computation as a set of constraints upon the state variables defining the solution is an attractive approach to specification of parallel programs. Previous efforts at parallel execution of constraint systems [1] have, to our knowledge focused on partitioning the constraint systems and applying standard constraint resolution algorithms in parallel. This approach, while interesting conceptually, has little hope of generating programs which will be of competitive efficiency with programs compiled to procedural representations of the same computations. There has also been considerable work in the logic programming community in concurrent constraint programming [15]. Our programming model is significantly different from the *ask* and *tell* framework of CCP languages. We adopt the *store-as-valuation* concept as in the imperative programming paradigm as opposed to the *store-as-constraint* [16] notion in concurrent constraint programming.

There has been considerable research in compiling constraint systems to sequential procedural representations [3, 10, 5]. Our work is similar in that procedural statements are extracted from a constraint specification. But the difference lies in our focus on extracting parallelism out of constraints. Recent work by Borning [2, 12] and others have integrated constraint and imperative models of programming with a view towards attaining the advantages of both. Benson [2] gives a useful survey of constraint programming models and their relationships.

1.2 Compilation of Parallel Programs from Constraint Systems

There are both motivation for continuing research in this direction and reasons for some optimism concerning success. Constraint systems have attractive properties for compilation to parallel computation structures. A constraint system gives the minimum specification (See [6] for an explanation of the benefits which derive from postponing imposition of program structure.) for a computation and thus offers the compiler great freedom of choice for derivation of control structure. Constraint systems offer some unique advantages as a representation from which parallel programs are to be derived. Both "OR" and "AND" parallelism can be derived. Either effective or complete programs can be derived from constraint systems on demand.

The granularity of the data flow graphs derived from the constraint systems depends upon the granularity of the operations directly represented in the constraint system. Introduction of matrix types and operations as primitives in the constraint representation gives natural units of computation at granularity appropriate for task level parallelism and avoids the problem of name ambiguity in the derivation of dependence (data flow) graphs from loops over scalar representation of arrays. The CODE 2.0 programming system [14], which expresses parallel structure over sequential units of computation declaratively in terms of a generalized dependence graph, provides a natural target language for compilation of parallel computation structures from constraint systems. The requirements for a constraint representation which can be compiled to a CODE 2.0 program which will execute efficiently include:

(i) modularity - The constraint system must have a modular structure which allows identification of reusable modules with which code modules can be identified.
(ii) functions - functions must be admissible in the constraint representation.
(iii) rich type set - the primitive type set should include structured types so that the units of sequential computation are of appropriate granularity and to avoid name ambiguity which precludes development of dependence graphs.

The next section describes our approach. This is followed by a description of the constraint representation. Section 4 details the compilation process. An example of a compiled program and its execution behavior are then given. The paper concludes with directions for future research.

2 Approach

A constraint is a relationship between a set of variables. E.g. $A + B = C$. Encapsulated within this constraint are three procedural assignment statements: $A := C - B$, $B := C - A$, $C := A + B$. Each of these statements can be extracted out of the initial constraint depending on which two of the three variables are inputs. If all three variables are inputs, the constraint can be transformed into a conditional which can be checked by a program for satisfiability. In both cases we classify the constraint as being *resolved.* If the number of inputs is less than two, the constraint can be left as *unresolved.*

A program specification in our system expresses a relationship between the variables of the program. Our approach consists of extracting a dataflow graph from the initial set of constraints and input set of variables (initial *known set*). The dataflow graph establishes the constraints by computing values for some or all of the non-input (output) variables. Generation of a dataflow graph is attempted by reordering of constraints and their classification as conditionals or computational statements at different points in the dataflow graph. Unresolved constraints are propagated down the graph in the hope of getting resolved at later points. The dataflow graph generation is explained in greater detail in Section 4. In a successful generation no constraints are unresolved at the end of some path in the dataflow graph. It is not necessary that every output variable be computed. Ours is a single-assignment system and multiple solutions to an output variable could be generated by alternate paths in the dataflow graph. All paths with unresolved constraints are pruned from the graph.

A constraint specification represents a family of dataflow graphs: one for each input set from which a dataflow graph establishing the constraints can be generated. Generation of all possible dataflow graphs can result in combinatorial explosion. Hence we concentrate on constructing a dataflow graph for a specified input set. The constraint specification can be reused for generating programs for different sets of inputs. Our translation exploits the AND-OR parallelism in the constraint specification.

The dataflow graph is mapped to the execution environment of the CODE 2.0 graphical parallel programming system [13] and parallel programs in C are generated. CODE 2.0 generates programs for the shared memory Sequent machine as well as the distributed memory PVM system. In addition sequential C programs can also be generated.

3 Constraint Representation

The initial types supported for the variables includes integers, reals, characters, and arrays. Towards the goal of supporting a rich typeset, we have implemented matrices and their associated addition, subtraction, multiplication and inverse operations. We also support a simple type system for matrices that at present includes lower triangular, upper triangular, and general matrices.

The constraints handled in our system are linear arithmetic constraints connected by the logical AND, OR and NOT operators. An arithmetic expression appearing in a constraint can be an integer/real value or variable, or is of one of the following forms: (X_1), $X_1\ O\ X_2$, or *fn_call* where X_1, X_2 are arithmetic expressions, $O \in \{$ +, -, *, /, div, mod $\}$, and *fn_call* is a function call. The constraints can be constructed by the programmer by the application of the following rules.

Rule 1 : Relations of the form below are constraints :

(a)

$$X_1\ R\ X_2,$$
$$R \in \{ <, \leq, >, \geq, = , \neq \},\ X_1,\ X_2 \text{ are arithmetic expressions}$$

(b)

$$M_1 = M_2$$
$$M_1, M_2 \text{ are linear expressions involving matrices and the matrix operators } +, -, *, \text{ and Inverse}$$

Rule 2 : Propositional formulas of the form below are constraints :

NOT A

A AND B

and

A OR B

A and B are constraints

Rule 3 : Calls to user-defined constraint modules are constraints.

Constraints formed from the use of just arithmetic expressions and relational operators (Rule 1) are referred to as *simple constraints.* These constraints form the building blocks for *non-simple* constraints which are formed by connecting simple constraints with logical AND/OR/NOT operators or by declaring a constraint module (Rules 2,3).

A program in our system in consists of the following parts:

- Program name.
- User-defined function signatures: These are the signatures of C functions which may be called within an arithmetic expression. These functions are defined in a separate file which is linked during execution.
- Global variable declarations: This section contains the name and type declaration of the global variables in the program. The scope rules for these variables are similar to those of a standard C program.
- Input variables: This section identifies the global input variables. All the variables in this section must be declared in the variable declaration section.
- Constraint Module definitions: This section contains zero or more definitions for constraint modules. The definition of a constraint module includes a name, listing of formal parameters and their types, local variable declarations, and a body. The local variable declarations and body are exactly similar in syntax to the global variable declarations and the main body, respectively. The Constraint Module definitions are described in greater detail in Section 3.1.
- Main body of the program: The body of the program consists of a set of constraints connected with AND/OR/NOT operators.

3.1 Constraint Modules

Our system provides for modularity by allowing the user to define *constraint modules* with formal parameters. This feature allows the development of large systems in a non-tedious manner. Formal parameters are names with associated types. A Constraint Module definition is semantically similar to a constraint in that it enforces a relationship (between its parameters). In addition, a Constraint Module encapsulates computation and provides a parameter interface. At the point of call to a constraint module, the actual parameters which are in the known set become inputs to the module and the rest of the parameters become outputs. A call to a constraint module is resolved if the compiler generates a non-empty dataflow graph from the constraint module definition.

3.2 Sample Program Specification

We present a sample program specification in this subsection. The header information containing the program name and variable declarations has been omitted. The problem is one of finding the non-complex roots of a quadratic equation, $a \times x^2 + b \times x + c = 0$. $"U"$ denotes an undefined value. *sqr*, *sqrt*, and *abs* are the square, squareroot and absolute functions.

/* definition of constraint module */

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

$t = sqr(b) - 4 \times a \times c$ AND $r = sqrt(abs(t))$

AND $t \geq 0$ AND $r1 = (-b + r)/2 \times a$ AND $r2 = (-b - r)/2 \times a$

/* Main*/

$a = 0$ AND $r1 = “U"$ AND $r2 = “U"$

OR

$a \neq 0$ AND DefinedRoots(a, b, c, r1, r2)

4 Phases of the Compiler

The compilation algorithm consists of four phases, which are described in greater detail in this section.

Phase 1: The textually expressed constraint system is transformed to an undirected graph representation as for example given by Leler [11].

Phase 2: A depth-first search algorithm is used to transform the undirected graph to a directed graph.

Phase 3: A set of input variables is chosen and the directed graph is again traversed by depth-first search to determine a mapping of constraints to firing rules and computations for nodes of a generalized data flow graph.

Phase 4: The data flow graph is mapped to the CODE 2.0 parallel programming environment [13]. CODE 2.0 accepts generalized dependence graphs of the form generated herein as source and produces parallel programs in C as executable for different parallel architectures.

4.1 Phase 1

The textual source program is transformed into a source graph for the compiler. Starting from an empty graph, for each application of Rules 1-3, an undirected constraint graph can be constructed by adding appropriate nodes and edges to the existing graph. For each instance of a simple constraint, a node is created and the constraint is attached to the node. For

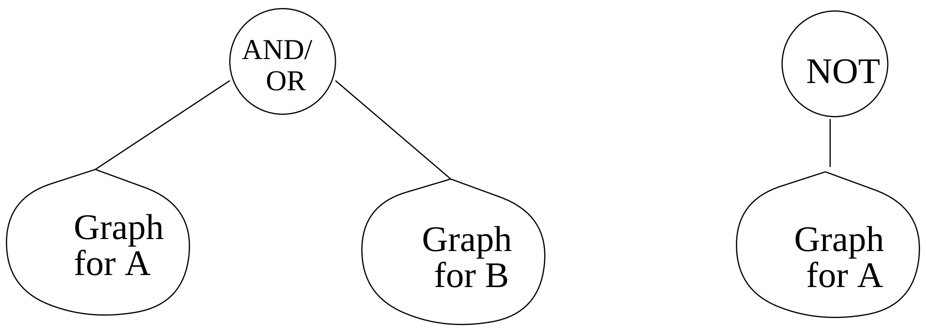


Figure 1: Rule 2

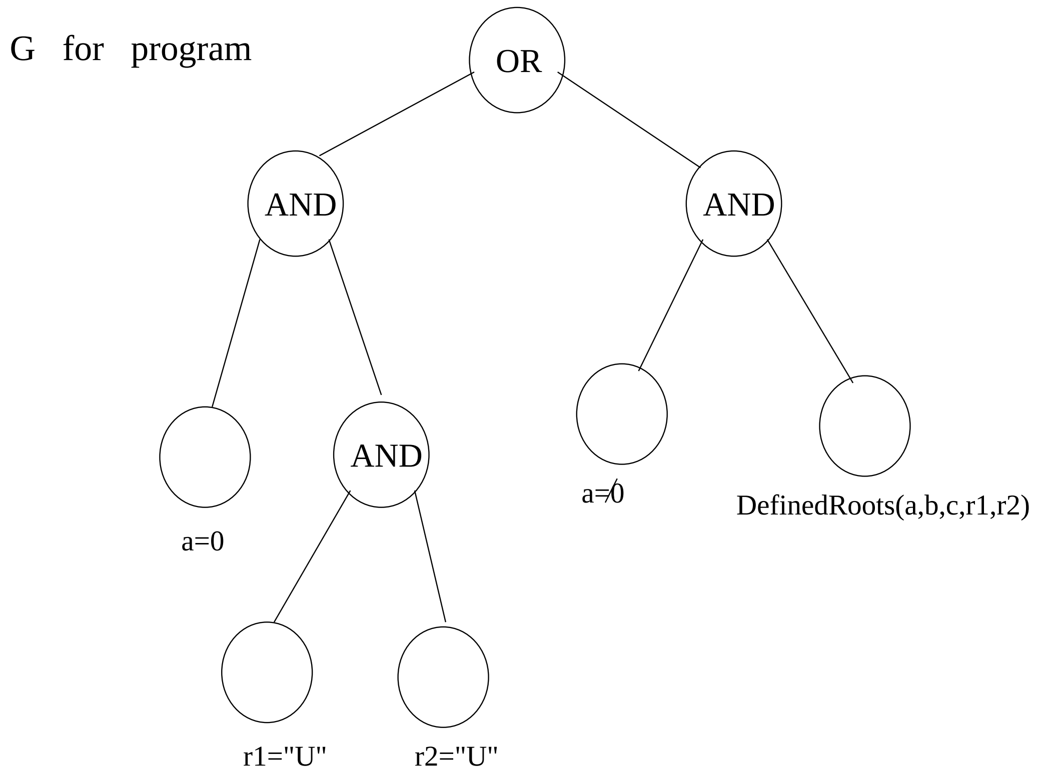
each instance of a constraint formed according to Rule 2, the graph is expanded as shown in Figure 1. For each instance of a constraint formed according to Rule 3, a node is created and the constraint module call (with the actual parameters) is attached to the node. Hence, the different kinds of nodes in the constraint graph are (i) $X_1\ R\ X_2$ (ii) AND/OR/NOT and (iii) Constraint Module Calls. Nodes of type (i), (ii), and (iii) are referred to as *simple constraint* nodes, *operator* nodes, and *call* nodes, respectively.

In the first phase, a set of constraint graphs is constructed. The first graph corresponds to the constraint specification in the main body of the program. The other graphs correspond to the constraint specifications in the body of the constraint module definitions. Evidently, each graph is constructed in a hierarchical fashion with simple constraint and call nodes at lower levels and operator nodes migrating to higher levels to connect one or two subgraphs, simple constraint nodes or call nodes. There will be a unique operator node at the highest level. The constraint graphs for the quadratic equation solver are shown in Figure 2.

4.2 Phase 2

A depth-first traversal of each of the graphs obtained at the end of phase 1 is done to generate a set of trees. The construction of these trees simplifies the initial constraint specification as illustrated in Figures 3 and 4, where a, b, c, and d are simple constraints. Nodes are collapsed in the graph such that constraints connected by AND operators are collected at the same node and constraints connected by OR operators are collected at nodes on diverging paths. This follows the rule that only one of the constraints needs to be satisfied for the OR constraint to hold and both the constraints have to be satisfied for the AND constraint to hold. The algorithm *dfs* is a simple generalization of Figures 3 and 4. Let v_1 be the unique node at the highest level of the graph. Each tree is initialized to one node, v_1^*. v_c and v_c^* are the nodes currently being visited in the graph and the tree, respectively. dfs is initially invoked with the call dfs(v_1, v_1^*).

The operator NOT has been omitted from the notation but it is implemented in the system. A NOT operator node will have a single subgraph or simple constraint as its child. If the child is a simple constraint, the NOT node is removed by negating the simple constraint. If the child is a NOT node, both NOT nodes are removed from the graph. Otherwise, the NOT



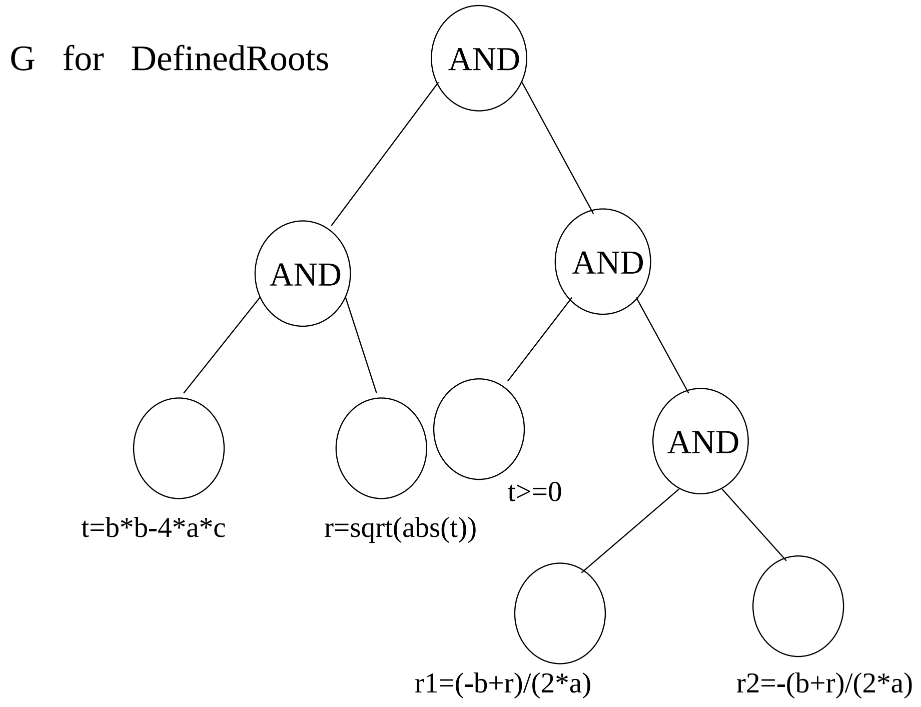


Figure 2: Constraint Graph for Quadratic Equation Example

node is moved down the tree by changing nodes in its path till it reaches a simple constraint or another NOT node. The rules for changing the nodes are as follows: AND becomes OR and OR becomes AND.

```
dfs ( $v_c$, $v_c^*$)
**begin**
        visited[$v_c$] := true;
        **Case** type($v_c$) **of**
            OR :
                **for** each unvisited neighbor $u$ of $v_c$ **do**
                    **if** type($u$) = OR dfs( $u$, $v_c^*$ )
                    **else** create node,$u^*$, in $G^*$ as child of $v_c^*$;
                        dfs( $u$, $u^*$ )
            AND :
                **if** there are two unvisited OR neighbors, $u_1$ and $u_2$, of $v_c$
                    create 4 nodes, $u_1^*$, $u_2^*$, $u_3^*$, and $u_4^*$ in $G^*$ as children of $v_c^*$;
                    /*let the 2 unvisited neighbors of $u_1$ be $u_{11}$ & $u_{12}$ and of $u_2$ be $u_{21}$ & $u_{22}$*/
                    visited[$u_1$] := true; visited[$u_2$] := true;
                    dfs( $u_{11}$, $u_1^*$ ); dfs( $u_{21}$, $u_1^*$ );
                    dfs( $u_{11}$, $u_2^*$ ); dfs( $u_{22}$, $u_2^*$ );
                    dfs( $u_{12}$, $u_3^*$ ); dfs( $u_{21}$, $u_3^*$ );
                    dfs( $u_{12}$, $u_4^*$ ); dfs( $u_{22}$, $u_4^*$ );
                **else for** each unvisited neighbor, $u$, of $v_c$ **do** dfs( $u$, $v_c^*$ );
            simple_constraint :
                attach constraint to $v_c^*$;
            Call Node :
                attach constraint module call to $v_c^*$;
**end**;
```

4.3 Phase 3

An attempt to generate a dataflow graph for the initial constraint and input set specification is made in this phase. In the generated dataflow graph, nodes are computational elements and arcs between nodes express data dependency. A path from the root of the graph to a leaf is a possible computation path that may be taken during execution as a result of values to variables. The general form of a dataflow node is shown in Figure 5.

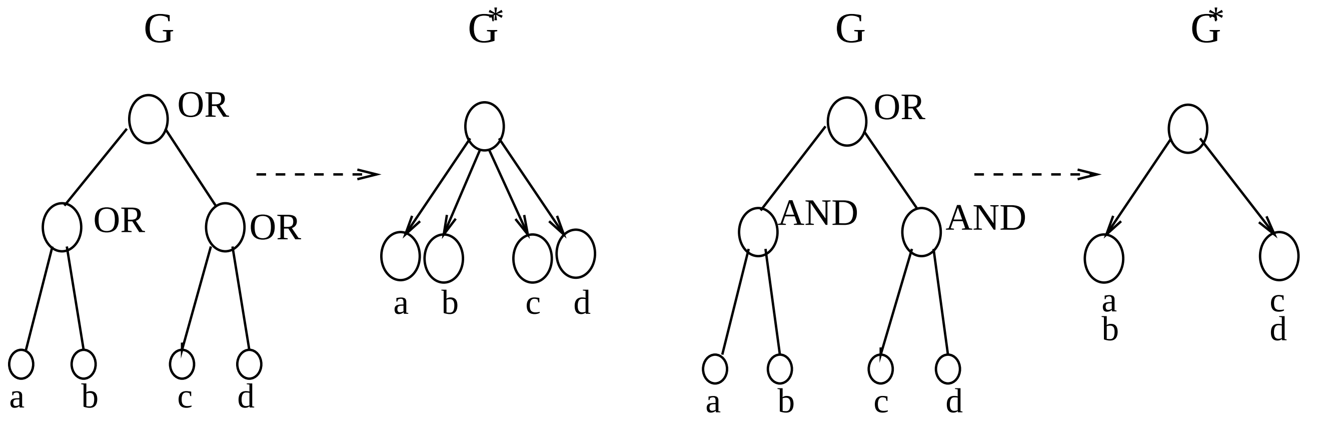


Figure 3: Phase 2 for an OR node

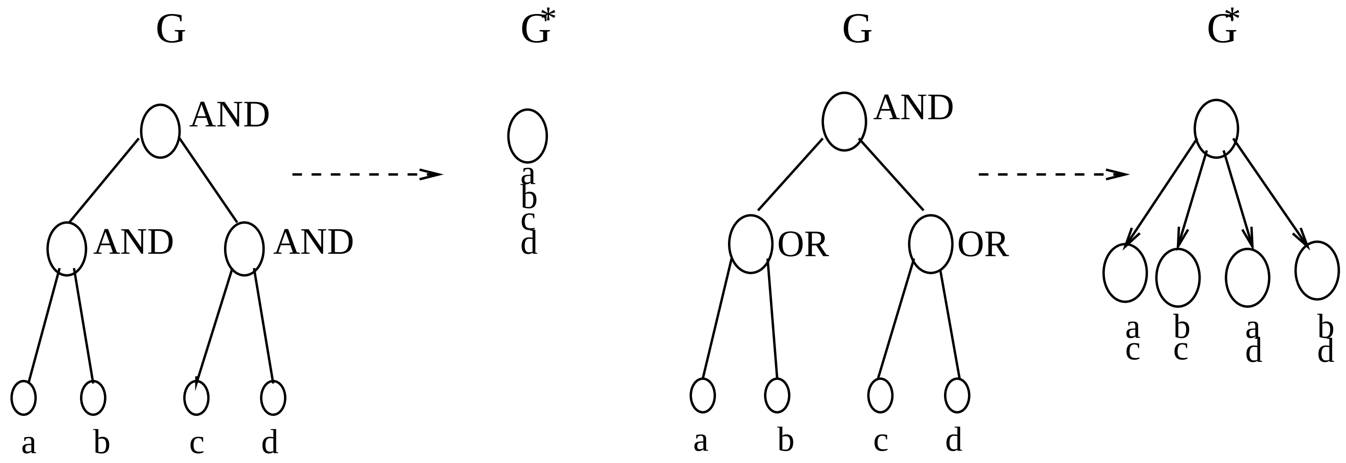


Figure 4: Phase 2 for an AND node

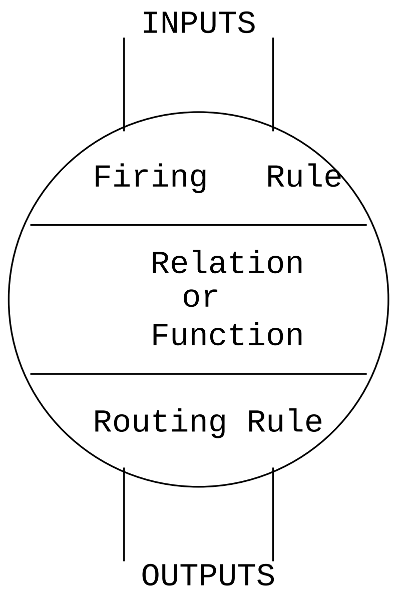


Figure 5: Generalized Data flow graph node

A depth-first traversal of the tree corresponding to the main program is done starting at the root. The traversal starts with the information that variables in the input set are known and tries to generate computation paths that assign values to variables in the output set. Each node in the tree has a set of constraints associated with it. When a node is visited, each constraint is examined for classification as one of the following: (i) Firing Rule: a condition that must hold before the current node can fire. To be classified as a firing rule, a constraint must have no unknowns when the node is visited. (ii) Computation: To fall into this category, a constraint must involve an equality and have a single unknown. The unknown is added to the known set and is retained in it for the subtree rooted at the current node. (iii) Routing Rule: a condition that must hold for this node to send out data on its outgoing paths. To be a routing rule a constraint must have no unknowns after the computation at the current node is executed.

When a constraint is classified as computation, it is reordered using symbolic algebra. If the variables in the computation are simple types the single unknown is moved to the left-hand side of the assignment statement. If the variables in the computation are matrices the assignment statement is replaced by calls to specialized matrix routines in C. For example the statement $A * x + b1 = b2$ with x as the unknown is first transformed into $A * x = b2 - b1$ and then a routine is called to solve for x. If A is lower (upper) triangular, then forward (backward) substitution is used to solve for x. Otherwise x is solved through an LU decomposition of A.

Any constraint not falling into one of the above set of categories is retained in an unresolved set of constraints which gets propagated down the tree. Examination of each constraint at a node and in the unresolved set of constraints loops till a stable state is reached. Any path that results in a leaf with unresolved constraints is abandoned. If all paths in the tree are abandoned the user is informed of the under-specification of the initial input set. Constraints involving inequalities must be resolved as firing/routing rules.

4.3.1 Phase 3 for Constraint Module Calls

A constraint module call has the form $ModuleName(e_1, e_2, \ldots, e_n)$ where e_i, $1 \leq i \leq n$ is an arithmetic expression. Let the corresponding formal parameters be $f_1, f_2, \ldots, f_n$. Any function call within e_i must have all known parameters else the constraint module call is unresolved. If all the variables in $e_1, \ldots, e_n$ are in the known set the call falls into category (i) or (iii) above. The formal parameters in the body of the constraint module are replaced by the corresponding e_i and the resulting conditional becomes a firing/routing rule.

If one or more variables in $e_1, \ldots, e_n$ are not in the known set then an attempt is made to generate a dataflow graph from the constraint module definition. The graph for the constraint module is traversed with a new known set = { all formal parameters whose corresponding actual parameters $\in$ old known set}. The output parameters are considered to be all formal parameters not in the old known set. The traversal returns "True" if all the constraints in the constraint module are resolved and every output parameter is computed at the end of at least one path in the resulting dataflow graph (other paths are discarded). This condition is different in the dataflow generation of the main program where all output variables need not be computed. The reason for imposing this condition is that the actual parameters are bound to the values returned in the output formal parameters at the point of call. If different sets of output variables are computed in different paths of the dataflow graph (as in the constraint $a = c$ OR $b = c$ with c known) it is not possible to determine statically which of the actual output parameters will be bound to the formal output parameters at runtime.

If a successful dataflow generation takes places a new set of constraints corresponding to each output parameter are generated as follows:
$e_{k1} = Z_1$, $e_{k2} = Z_2$, ..., $e_{kp} = Z_p$, where Z_i, $1 \leq i \leq p$, are new variables generated by the compiler and $e_{k1} \ldots e_{kp}$ are the output parameters. An attempt is made to resolve this set of constraints. It is to be noted that there can be at most one unknown in each of parameters $e_{k1} \ldots e_{kp}$ and the set of constraints will have to be resolved as computation for each of these unknowns. In this case a call node which calls the generated dataflow graph is generated. A child node of the call node receives values computed by the call node and binds them to $Z_1 \ldots Z_p$ and performs the computation generated from the new set of constraints.

If the traversal returns "False" (a dataflow graph is not generated) then the current search path is discarded. Each constraint module invocation is translated as a separate program module. It might seem that many redundant translations would be performed. But a table can be maintained for each module which contains entries showing the dataflow graphs generated for combinations of parameter inputs. Redundant translations can be eliminated this way.

4.3.2 Extraction of AND parallelism

The computational statements that are assigned to a node have the potential for parallel execution. For instance, the assignments $a := b + c$ and $x := b + 2$ are independent and can be done in parallel. If another computation statement accesses the value of a then there is a data dependency between the statement $a := b + c$ and the current statement. If a is the only data dependency, the current statement is assigned to the node which computes a. Else a new node is created for the current statement and a data arc brings in the value of a. Evidently, the granularity of such a scheme depends on the complexity of the functions called within the statements and the complexity of the operators.

We have further exploited the complexity of the matrix operations by splitting up the specifications, performing computations in parallel and composing them. For example if $x := m * y + b$ and x, m, y, and b are matrices $m * y$ can be done in parallel. Our system splits the above computation into two statements: (i) $Z := m * y$ (ii) $x := Z + b$ in view of the fact that multiplication of matrices is an $O(N^3)$ operation. This will be significantly more costly to compute than addition of matrices. Since $m * y$ is a primitive operation, a procedure which implements a parallel algorithm for $m * y$ can be invoked. In a later version of the compiler provision will be made for user specification of parallelism for operations over structures.

Hence data parallelism is exploited by keeping in mind that ours is a single-assignment system and the lone write to a variable (assignment with variable on left-hand side) will appear before any reads (variable on right-hand side of assignment statement) to the variable. Computations assigned to a node are thus split up as computations to different nodes running in parallel. Results are collected by a merging node.

The structures obtained at the end of Phases 2 and 3 from the quadratic equation example are shown in Figure 6. $r1$ and $r2$ are computed in parallel and then merged. This example does not exhibit coarse grained parallelism. It was used to illustrate the compilation algorithm in a simple manner. The programming example in Section 5 is a good example for coarse grained parallelism.

4.4 Phase 4

Our target for executable for constraint programs is the CODE 2.0 parallel programming environment. CODE 2.0 takes a dataflow graph as its input. The form of a node in a CODE dataflow graph is given in Figure 5. It is seen that there is a natural match between the nodes of the dataflow graph developed by the constraint compilation algorithm and the nodes in the CODE graph. The arcs in the dataflow graph in CODE are used to bind names from one node to another. This is exactly the role played by arcs in the dataflow graph generated by the translation algorithm.

The CODE 2.0 programming interface is drawing and annotating of the directed graph on a workstation. This annotated directed graph is converted to a graph-format file, which is then passed through several translations to obtain an executable. The graph-format file stores an abstract syntax tree (AST) which represents in a hierarchical form the CODE program that

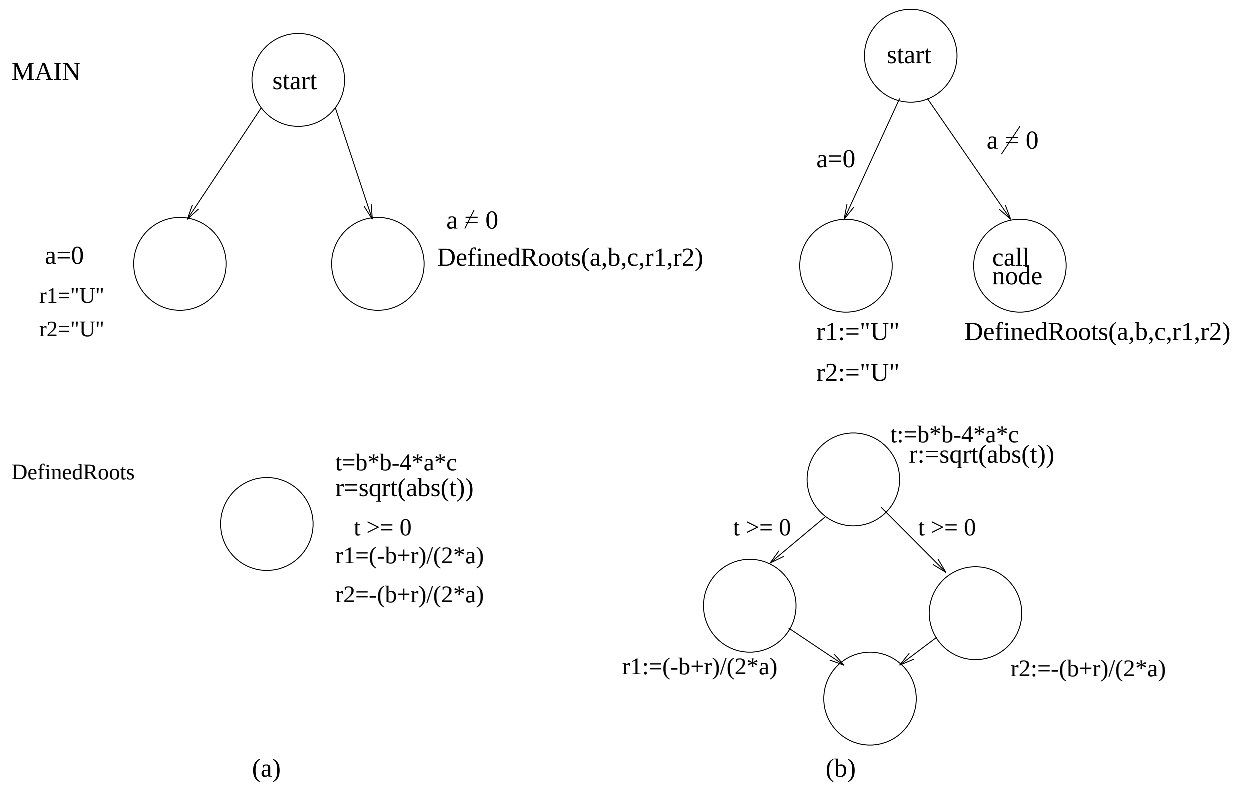


Figure 6: (a) Phase 2 (b) Phase 3 for Example

is to be translated. The output of the translator for the constraint systems is the AST. This AST is passed through the same translations as an AST from a CODE2.0 program. The final output is an executable in the form of a parallel C program. The dataflow graphs generated from the constraint modules are stored as separate AST's in CODE2.0. These can be invoked by the corresponding call nodes in the program.

5 Programming Example: Block Triangular Solver

The example chosen is the solution of a triangular matrix which is a commonly used example for illustration of parallel computations [13]. It solves the $Ax = b$ linear algebra problem for a known lower triangular matrix A and vector b. The parallel algorithm [9] is quite simple and involves dividing the matrix into blocks as shown in Figure 7 (a). Figure 7 (b) shows the dataflow of the algorithm. The S's represent lower triangular submatrices that are solved sequentially, and the M's represent submatrices that must be multiplied by the vector from above and the result subtracted from the vector from the left. The arcs represent the dependencies between these operations.

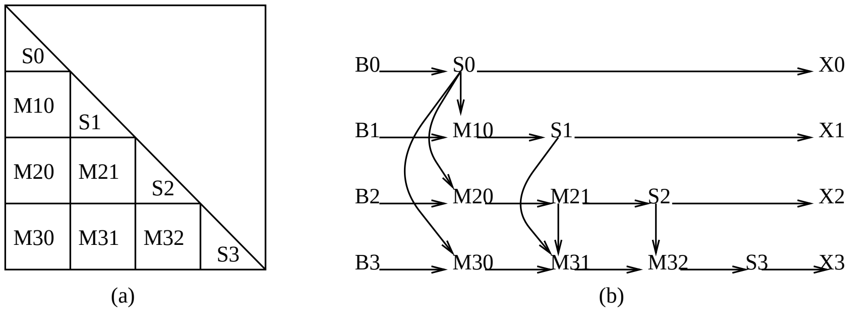


Figure 7: (a) Partitioned Lower Triangular Matrix (b) Block Triangular Solver DataFlow

The constraint specification in our program for a problem split into 4 blocks is as follows:

```
( s0*x0 == b0 AND
m10*x0 + s1*x1 == b1 AND
m20*x0 + m21*x1 + s2*x2 == b2 AND
m30*x0 + m31*x1 + m32*x2 + s3*x3 == b3
)
```

The input set is given as { s0, s1, s2, s3, b0, b1, b2, b3, m10, m20, m30, m21, m31, m32 }. The output set is detected as {x0, x1, x2, x3}.

5.0.1 Performance Results

The speedups given in Figure 8 are for a 1200×1200 matrix. The executable was run on the shared memory Sequent machine. It is seen that the performance of the constraint generated code is comparable to the hand coded performance. The difference in speedups is mainly due to the fact that the hand coded program is optimized for a shared memory execution environment. Architectural optimization can easily be included in later versions of the constraint compiler.

6 Conclusions and Directions for Future Research

This first stage of research has established that constraint systems over structured types can be compiled to efficient coarse grained parallel programs for some plausible examples. This is, however, only a first step in demonstration of a practical compiler for constraint systems to parallel programs. It is clearly necessary to be able to express constraints on partitions of matrices if large scale parallelism is to derived from constraint systems without use of the cumbersome techniques derived for array dependence analysis of scalar loop codes over arrays. There are several promising approaches: object- oriented formulations of data structures are

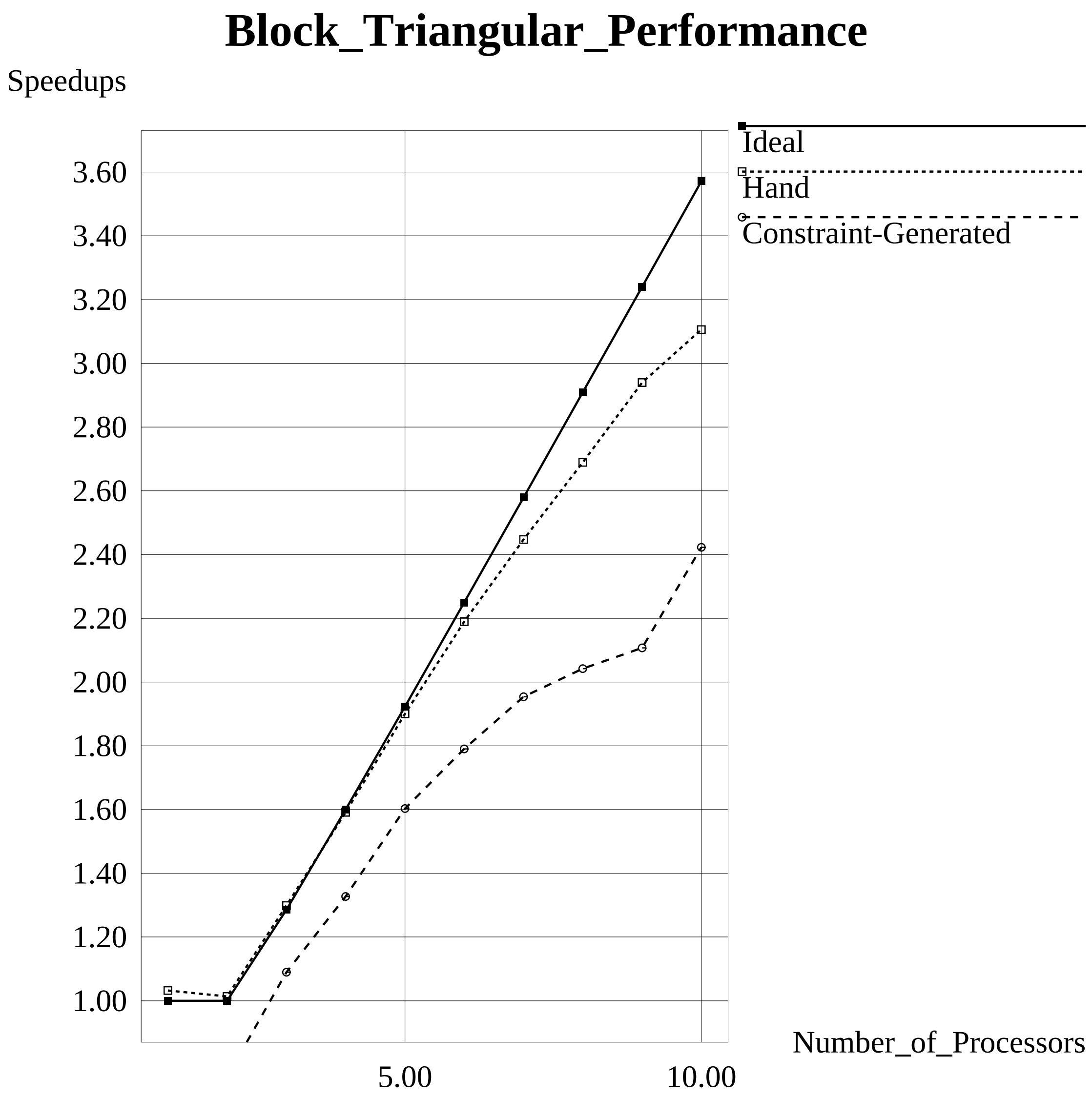


Figure 8: Performance Results for the Block Triangular Solver

one possibility. A simpler and more algorithmic basis for definition of constraints over partitions of matrices is to utilize the hierarchical type theory for matrices recently published by Collins and Browne [7]. The hierarchical type model for matrices establishes a compilable semantics for computations over hierarchical matrices. Collins and Browne [8] have designed and implemented a translator which transforms pseudo-equational representations for computations expressed in the hierarchical type model for matrices into parallel programs.

The next steps in this research are:

a) To extend the current compiler to incorporate a richer spectrum of data and constraint types (including hierarchical matrices).

b) Inclusion of Forall quantifiers in the constraint representation.

c) To formulate constraint systems which integrate search and computation to advantage for applications such as visualization, image analysis and adaptive computational algorithms.

d) To validate the compiler by measurement of performance of the applications on several parallel architectures.

e) Provide compiler optimizations which take advantage of architectural characteristics of specific execution environments.

f) To include constraint hierarchies [4].

References

[1] Doug Baldwin. Consul: A Parallel Constraint Language. *IEEE Software* 1989.

[2] Freeman-Benson, B.N. *Constraint Imperative Programming* Technical Report 91-07-02 University of Washington, Department of Computer Science and Engineering, August, 1991.

[3] Bjorn N. Freeman-Benson. A Module Compiler for Thinglab II. *Proc. 1989 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, New Orleans, October 1989. ACM.

[4] Alan Borning, Robert Duisberg,Bjorn N. Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint Hierarchies. *Proc. 1987 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987, ACM.

[5] Alan Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October 1981, pp 353-387.

[6] Chandy, K.M and Misra, J. *Parallel Program Design : A Foundation* Addison-Wesley, Reading, 1989.

[7] Collins, T.S. and Browne, J.C. MaTrix++; An Object-Oriented Approach to the Hierarchical Matrix Algebra *In Proceedings of the Second Annual Object-Oriented Numerics Conference* , Sun River, OR, April, 1994.

[8] Collins, T.S. and Browne J.C. MaTrix++; An Object-Oriented Environment for Parallel High-Performance Matrix Computations To appear in the *Proceedings of the 1995 Hawaii International Conference on Systems and Software*

[9] J.J. Dongarra and D.C. Sorenson. *SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs.* Argonne National Laboratory MCSD Technical Memorandum No. 86, Nov. 1986.

[10] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, Ken Doyle. Fabrik : A Visual Programming Environment. *Proc. 1988 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1988. ACM.

[11] William Leler. *Constraint Programming Languages.* Addison-Wesley, 1988.

[12] Gus Lopez, Bjorn Freeman-Benson, Alan Borning. Kaleidoscope : A Constraint Imperative Programming Language. *Constraint Programming*, B. Mayoh, E. Tougu, J. Penjam (Eds.), NATO Advanced Science Institute Series, Series F: Computer and System Sciences, Springer-Verlag, 1993.

[13] P. Newton and J. C. Browne. The Code 2.0 Graphical Parallel Programming Environment. *Proceedings of the 1992 International Conference on Supercomputing* (Washington, DC, July 1992), pp 167-177.

[14] P. Newton and J.C. Browne. *A Graphical Retargetable Parallel Programming Environment and its Efficient Implementation.* Technical Report TR93-28, Dept. of Computer Sciences, Univ. of Texas at Austin, 1993.

[15] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon, Pittsburgh, 1989. School of Computer Science.

[16] Vijay A. Saraswat, M. Rinard, and Prakash Panangadan. Semantic foundations of Concurrent constraint programming. *Proc. Principles of Programming Languages Conf.*, pages 333-352, 1991.