A Constraint-based Parallel Programming Language

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

TR95-42 March 27, 1996

Abstract

This paper describes the first results from research on the compilation of constraint systems into task level procedural parallel programs. Algorithms are expressed as constraint systems. A dataflow graph is derived from the constraint system and a set of input variables. The dataflow graph, which exploits the parallelism in the constraints, is mapped to the target language CODE 2.0, which represents parallel computation structures as generalized dependence graphs. Finally, parallel C programs are generated. The granularity of the derived data flow graphs depends upon the complexity of the operations directly represented in the constraint system. To extract parallel programs of appropriate granularity, the following features have been included: (i) modularity, (ii) operations over structured types as primitives, (iii) definition of sequential C functions. A prototype of the compiler has been implemented. The domain of matrix computations has been targeted for applications. Some examples have been programmed. Initial results are encouraging.

1 Introduction

Representing an algorithm or computation as a set of constraints upon the state variables defining the solution is an attractive approach to specification of programs, but there has been little success previously in attaining efficient execution of parallel programs derived from constraint representations. There are however, 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 [8] for an explanation of the benefits which derive from postponing imposition of program structure.) for a computation, thereby offering the compiler 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 goal of this research is to explore a non-traditional approach to parallel programming: using constraint systems for specification of numerical computation algorithms. This paper reports early results from this radical approach. The next two sections outline our approach and its conceptual results. This is followed by a description of the programming language features and the compilation algorithm. We conclude the paper with performance results for some examples and directions for future work.

2 Approach

The approach we take specifies algorithms as constraint systems. A dataflow graph is derived from a constraint specification of an algorithm and an input set of variables. The dataflow graph is mapped to the target language CODE 2.0 [24], which expresses parallel structure over sequential units of computation declaratively as a generalized dependence graph. Finally parallel C programs for the Sequent shared memory machine and the distributed memory PVM system can be generated. Sequential C programs can also be generated. The granularity of the data flow graphs derived from the constraint systems depends upon the type system directly represented as primitives in the constraint representation. 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 general requirements for a constraint representation which can be compiled to execute efficiently, include: (i) modularity for reusable modules, (ii) definition of sequential functions, and (iii) a rich type set. The main features of our approach are outlined in the rest of the section.

(i) Constraint Representation: In our system a program is expressed as a set of constraints between the program variables. A constraint is a relationship between a set of variables. E.g. $A + B == C$ is a constraint expressing the equality between C and the sum of A and B. A constraint program specification enumerates the different relationships that must be established or maintained by the executing code. Our system handles linear arithmetic constraints composed

by AND/OR/NOT operators.

(ii) Constraint Modules: To develop programs for large scale applications we have incorporated modularity by defining *constraint modules* with formal parameters. Constraint modules are encapsulations of relationships between parameters and can be invoked within another constraint. Formal parameters are names with associated types. The body is a constraint specification. There follows a program for finding the non-complex roots of a quadratic equation, $ax^2 + bx + c == 0$ which uses a module specification. *"U"* denotes an undefined value. *sqr*, *sqrt*, and *abs* are user-defined functions. A program specification also identifies the set of inputs to the program. In the quadratic equation solver example this could be $\{a, b, c\}$.

```
/* Constraint module */
DefinedRoots(a, b, c, r1, r2)
t == sqr(b) − 4ac AND r == sqrt(abs(t))
AND t ≥ 0 AND r1 == (−b + r)/2a AND r2 == (−b − r)/2a

/* Main */
a == 0 AND r1 == "U" AND r2 == "U"
OR
a ≠ 0 AND DefinedRoots(a, b, c, r1, r2)
```

(iii) Compilation to a Procedural Language: Encapsulated within the constraint $A+B == C$ are three procedural (computational) statements: $A = C-B$, $B = C-A$, $C = A+B$. One of these three assignment statements can be extracted out of the initial constraint depending on which two of the three variables $\{A, B, C\}$ are inputs or *known* variables. If all three variables are inputs the constraint can be transformed into a conditional which can be checked by a program for satisfiability. If less than two variables are inputs the constraint is unresolved and no resulting program can be extracted. The derived dataflow graph establishes the constraints by computing values for some or all of the non-input variables. Generation of a dataflow graph is attempted by classifying constraints as conditionals or computational statements at different points in the dataflow graph. Dataflow graph generation is explained in greater detail in Section 5.

Our translation exploits the AND-OR parallelism in the constraint specification. The resulting program has single-assignment variables and can generate multiple solutions on alternate paths of the dataflow 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. We construct only the dataflow graph for a specified input set. The constraint specification can be reused for generating the dataflow graphs for different sets of inputs.

(iv) Domain Specification: The Hierarchical Type System: The semantic domain chosen for our application programs is matrix computation. To customize the environment for this domain we have a built-in matrix type with its associated operations of matrix addition, subtraction, multiplication and inverse. The matrix subtypes currently included in our system are lower triangular, upper triangular and generalized matrices. We plan to extend to more specialized matrices including hierarchical matrices [10]. Specialized algorithms based on the structure of the matrix can be invoked for the matrix subtypes.

(v) Separate Specification of Compilation Options and the Execution Environment: To obtain architecturally optimized programs we plan to incorporate features such as the following as part of an execution environment specification separate from the constraint specification.

- The programmer can override the default option of parallelizing a particular module if the default leads to a smaller than desired granularity.

- The programmer can have the option of selecting certain operations for executing in parallel. For instance, if matrix multiplications are chosen to be compiled for parallel execution the multiplications in the computation $M_1 * M_2 + M_3 * M_4$ will be computed in parallel.

- Selected parallel algorithms can be chosen to execute some of the operations.

- In a shared memory machine, the inputs to the main program and to a module can be declared as shared variables. Since ours is a single-assignment system these variables will only be read and not written to.

- Parallelization methods can be chosen for loops over constraints.

3 Potential Advantages of a Successful Compilation of Parallel Programs from Constraint Systems

(i) Ease of Use: The programmer can think in terms of relationships between the objects of the domain rather than the details of how relationships will be implemented. The programmer is given a representation for reasoning in high level specifications, with no issues of control flow or synchronization with which to deal. Domain specifications enable a programmer to reason about familiar entities.

(ii) Portability: Separating the specifications for computation and the execution environment makes the former portable. The constraint specification is free from issues of parallelism. Hence there is complete freedom for the compiler to choose appropriate mechanisms for implementation of parallelism. This is important since different architectures will not support the same communication/synchronization primitives with equal performance.

(iii) Performance: To generate parallel programs of competitive efficiency with that of hand-coded parallel programs it is imperative to exploit the architectural characteristics of the executing machine. The execution environment specification can be used by the compiler to choose execution environment specific synchronization/communication mechanisms. By compiling to a procedural language, we avoid the slow execution environments plaguing typical declarative languages, which involve time-consuming search techniques, or other constraint programming languages, which use interpretive techniques like local propagation. By narrowing our semantic domain, the system has the added flexibility of choosing optimal algorithms for executing operations.

(iv) Scalability: To build large programs in a non-tedious manner, scalability is important in two respects: to scale to larger applications and to scale to larger problem sizes. Both these features are supported in our system through parameterization of abstract types, modularity and use of C functions within arithmetic expressions.

4 Constraint Representation

Our basic type system consists of integers, reals, characters, and arrays. The associated operations are the simple arithmetic operators of addition, subtraction, multiplication, division, div and mod. An arithmetic expression appearing in a constraint can be

(i) an *integer/real value or variable or a function call*

(ii) *(X_1), X_1 O X_2* X_1, X_2 are arithmetic expressions, and $O \in \{+, -, *, /, \text{div}, \text{mod}\}$

(iii) *Operator* loops of the form *< op > FOR (< index > < low > < high >) X* $op \in \{+, -, *, /\}$, *index* is an identifier, *low* and *high* are range bounds for *index*, X is an arithmetic expression. This construct allows the specification of operations over terms involving *index* in a concise manner. E.g. + FOR (i 1 5) $A[i]$ specifies the sum of the elements in the array A from positions 1-5. The index bounds for the outermost loop have to be integers. Nested loops can have expressions involving outer level indices as index bounds.

The constraints can be constructed by the application of the following rules.

Rule 1 : Relations of the form below are constraints :

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

(ii) $M_1 == M_2$
M_1, M_2 are linear expressions involving matrices and the matrix operators +, -, *, and Inverse

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

(i) NOT A
(ii) A AND/OR B
A and B are constraints

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

Rule 4 : Loops constructed over a list of constraints are constraints. Two such types of loops are defined as: OR/AND FOR (<index> <low> <high>) $A_1, A_2, \ldots, A_n$ where *index* is an identifier, *low* and *high* are range bounds for *index*, and $A_1, A_2, \ldots, A_n$ are constraints formed

from Rule 1 or Rule 4. AND and OR loops express terms connected by propositional connectives in a concise manner. They are very useful in defining constraints over matrices or arrays. Nested loops are allowed. The range limits for the outermost loop have to be integers. Nested loops can have arithmetic expressions involving outer indices as index ranges.

E.g. AND FOR (i 1 5) { A[i] == A[i-1] B[i+1] == A[i] } specifies the constraint construct
$A[1] == A[0]$ AND $A[2] == A[1]$ AND ... AND $A[5] == A[4]$ AND
$B[2] == A[1]$ AND $B[3] == A[2]$ AND ... AND $B[6] == A[5]$.

Constraints formed from the use of 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 loops, or by declaring a constraint module (Rules 2-4).

A program in our system consists of the following sections:

- Program name, Global variable declarations, Global Input variables.

- User-defined function signatures: These are the signatures of C functions (linked during execution) which may be called within an arithmetic expression.

- Constraint Module definitions: The definition of a constraint module includes a name, listing of formal parameters and their types, local variable declarations, and a body. The body is similar in syntax to the main body.

- Main body of the program: The body of the program consists of a set of constraints connected with AND/OR/NOT operators.

5 Compilation

The compilation algorithm consists of the following 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 [21].

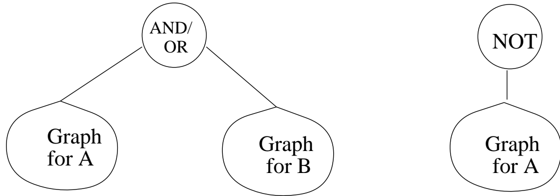

Figure 1: Rule 2

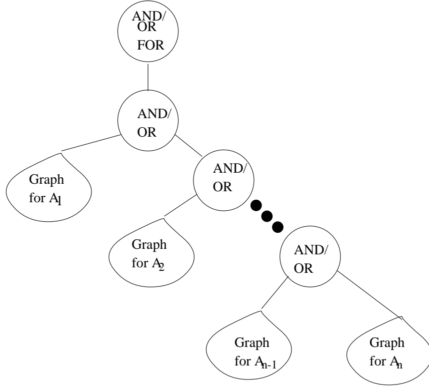

Figure 2: Rule 4

Phase 2. A depth-first search algorithm transforms the undirected graph to a directed graph.

Phase 3. With a set of input variables the directed graph is traversed by a depth-first search to map the constraints to firing rules and computations for nodes of a generalized dataflow graph.

Phase 4. Use the execution environment specification to optimize the dataflow graph. This phase is yet to be completely implemented in the system.

Phase 5. The data flow graph is mapped to the CODE 2.0 parallel programming environment [25] to produce parallel programs in C as executable for different parallel architectures.

Phase 1: Generation of Constraint Graphs

The textual source program is transformed into a source graph for the compiler. Starting from an empty graph, for each application of Rules 1-4, 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 with the attached constraint. For each application of Rule 2, the graph is expanded as shown in Figure 1. For each application of Rule 3, a node is created with the attached constraint module call and the actual parameters. For each application of Rule 4 with the loop specified over a list of constraints $A_1 \ldots A_2$, the graph is expanded as shown in

Figure 2. The index and its range information are attached to the loop node.

The different kinds of nodes in the constraint graph are (i) *Simple constraint* nodes corresponding to simple constraints like $X_1\ R\ X_2$ (ii) *Operator* nodes corresponding to AND/OR/NOT connectives, (iii) *Call* nodes corresponding to Constraint Module Calls, and (iv) *Loop* nodes corresponding to For Loops. In the first phase, a set of constraint graphs is constructed corresponding to the main body and the constraint module bodies. Each graph is constructed in a hierarchical fashion. Simple constraint nodes, call nodes, and loop nodes occur at lower levels. and operator nodes migrate to higher levels to connect one or two subgraphs, simple constraint nodes, call nodes or loop nodes. There will be a unique node at the highest level.

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 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 because all constraints have to be satisfied for an AND operator to hold and any one of the constraints needs to be satisfied for an OR operator to hold. The algorithm *dfs* is a generalization of Figures 3 and 4. Let v_1 be the unique node at the highest level of the input graph. Each output tree G^* 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. Otherwise, the NOT node is moved down the tree by changing nodes in its path till it reaches a simple constraint. The rules for changing the nodes are as follows: AND becomes OR and OR becomes AND.

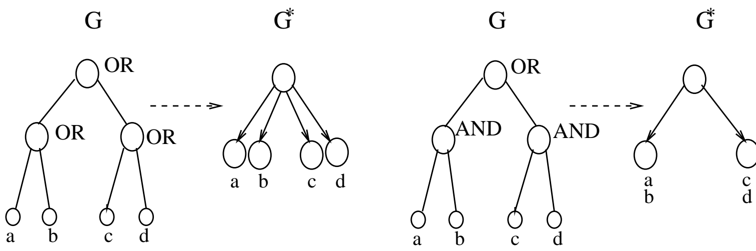

Figure 3: Phase 2 for an OR node

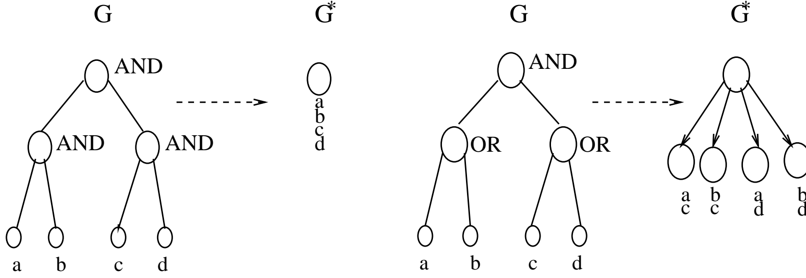

Figure 4: Phase 2 for an AND node

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^*;

Loop Node : create node, u^*, in G^* as child of ${v_c}^*$

Attach node index, range bounds to u^*

create node ${u_1}^*$ as child of u^*

dfs(unvisited neighbor of v_c, ${u_1}^*$);

end;

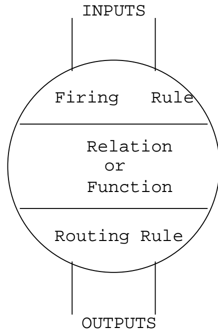

Figure 5: Generalized Data flow graph node

Phase 3: A dataflow graph generation for the initial constraint and input set specification is attempted 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. A node has the form: firing rule, computation, routing rule. The general form of a dataflow node is shown in Figure 5. 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 as a result of the computation at the current node.

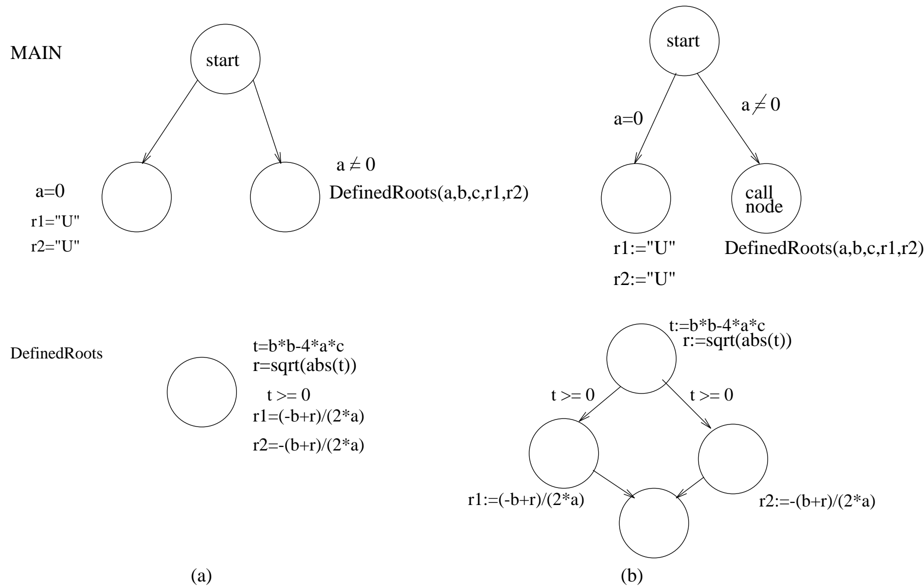

Figure 6: (a) Phase 2 (b) Phase 3 for Example

When a constraint is classified as a computation, it is mapped to an equation. 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 categories (i)-(iii) is retained in an unresolved set of constraints which is propagated down the tree. Examination of each constraint at a node and in the unresolved set of constraints continues 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 inconsistency of the initial specification. Constraints involving inequalities must be resolved as firing/routing rules.

5.0.1 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. Each e_i can contain at most one unknown. Otherwise the constraint module call is unresolved. If all the variables in $e_1, \ldots, e_n$ are in the known set the call becomes a conditional. If one or more variables are not in the known set then a dataflow graph generation is attempted from the constraint module definition. The graph for the constraint module is traversed with a new known set = {all formal parameters such that the variables of their corresponding actual parameters $\in$ old known set}. The output parameters are considered to be all formal parameters not in the new 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). For a more detail exposition of this phase refer to [19]. If the traversal returns "False" (a dataflow graph is not generated) the current search path is discarded. Each constraint module invocation is translated as a separate program module. Redundant translations can be eliminated by maintaining a table for each module with entries showing the dataflow graphs generated for combinations of parameter inputs.

5.0.2 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. Hence a particular node may be split into several nodes to exploit the data parallelism in the computation at the node. 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 will appear before any reads 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.

5.0.3 Phase 3 for AND For loops

Our system design handles loops with specified patterns of access. As of now the implemented system does not include OR loops. To classify a constraint in a loop as a firing/routing rule or computation, it is evaluated for all index values in the specified range of the loop. We shall refer to this as instantiations of the loop in this phase. The restrictions on the loop structure in order to be compiled successfully in our system are specified below:

- A constraint has to have the same classification in all instantiations of the loop.
- If a constraint is classified as computation, the same unique general term in the constraint

has to be the unknown in all instantiations of the loop.

An example of a construct that will be compiled successfully is

$$A[0] == 0 \text{ AND}$$
$$\text{AND FOR i 1 5 } \{$$
$$A[i] == A[i-1] + B[i] \ \}$$
with B known and A unknown.

An example of a construct that will not be compiled successfully is

$$\text{AND FOR i 1 5 } \{$$
$$A[1] == A[i] + B[i]$$
with A unknown and B known.

This is because in the first iteration both the terms $A[1]$ and $A[i]$ are unknown whereas subsequent iterations have only $A[i]$ as an unknown.

To extract parallelism from constraints classified as computation, the patterns of access within a constraint are studied. Throughout this discussion the case of array accesses will be detailed. The case of scalar accesses in loops will follow trivially. The different types of patterns of access for a loop with a single constraint and the corresponding structures that they are compiled to are detailed as follows.

(i) If the array on the left-hand side of an assignment (computation) does not occur on the right-hand side then all instantiations of the loop at runtime are independent of each other and can be run in parallel. The parallel structure compiled for such a loop is shown in Figure 7(a). The node performing the computation and the arc connecting the parent to it are replicated N times where N is the range of the loop index. The results of the computation performed by the parallel nodes are merged (not shown in figure).

(ii) If the array on the left-hand side of an assignment does occur on the right-hand side then the set of accessed indices of the array are generated for both the left and right sides. If these sets are disjoint the instantiations of the loop are independent for this constraint and the structure compiled in this case is also Figure 7(a).

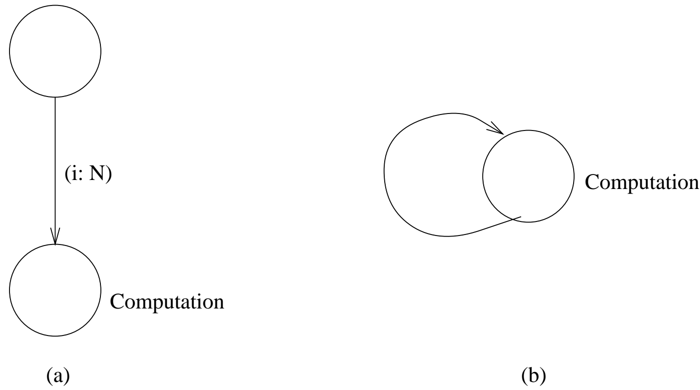

Figure 7: Loop: (a) Parallel Execution (b) Sequential Execution

(iii) If cases (i) and (ii) do not hold, the loop instantiations are inter-dependent and are run sequentially. The compiled structure for this is shown in Figure 7(b). The node performing the computation is invoked a number of times equal to the number of iterations of the specified loop.

Phase 5: 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 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

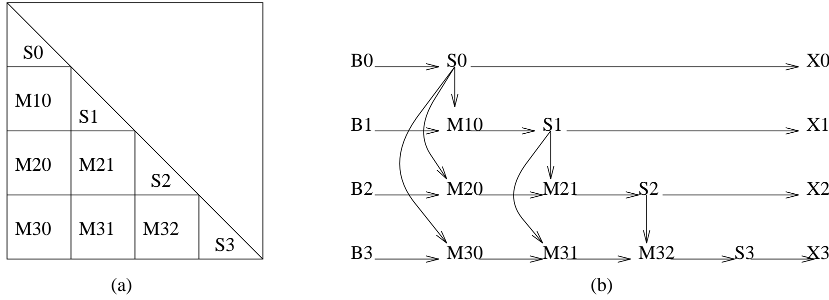

Figure 8: (a) Partitioned Lower Triangular Matrix (b) Block Triangular Solver DataFlow

modules are stored as separate AST's in CODE2.0. These can be invoked by the corresponding call nodes in the program. CODE2.0's translator for the Sequent produces parallel C programs that use parallel programming primitives from the FastThreads library [1].

6 Programming Examples and Results

A prototype of the compiler has been implemented using C++. A small number of examples have also been programmed and executed on the Sequent Symmetry machine and the PVM system. Performance results are described in the next subsection.

6.1 Block Triangular Solver

The example chosen is the solution of a triangular matrix which is a commonly used example for illustration of parallel computations [25]. It solves the $Ax = b$ linear algebra problem for a known lower triangular matrix A and vector b. The parallel algorithm [14] is quite simple and involves dividing the matrix into blocks as shown in Figure 8(a). Figure 8(b) shows the form of the dataflow in the algorithm. The $S_0 \ldots S_3$ represent lower triangular sub-matrices that are solved sequentially, and the $M_0 \ldots M_3$ represent sub-matrices 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.

A constraint specification for a problem instance split into 4 blocks is as follows:

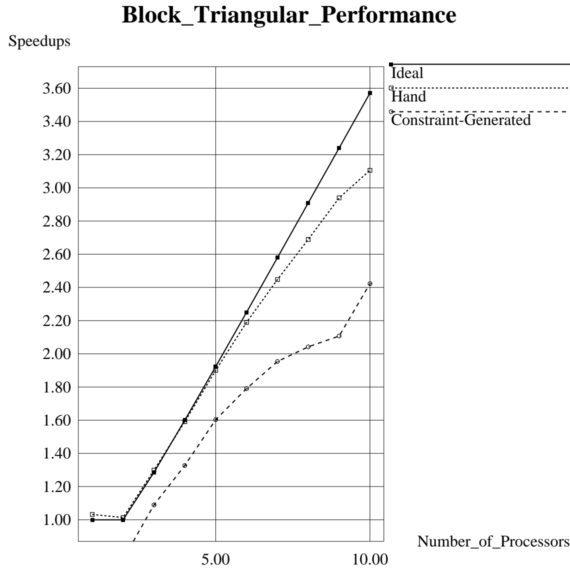

Figure 9: Performance Results for the Block Triangular Solver

(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)

Using operator loops the BTS algorithm can be conveniently expressed as AND FOR (i 0 3) { + FOR (j 0 i) { $A[i][j] * x[j]$ } == $b[i]$ } where the subscripts for A, x, and b define partitions on the matrices. 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}.

6.1.1 Performance Results

Figure 9 gives the speedups for a 1200 $\times$ 1200 matrix on a shared memory Sequent machine. It is seen that the performance of the constraint generated code is comparable to the hand coded program's 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 will be included in later versions of the constraint compiler.

6.2 The Block Odd-Even Reduction Algorithm

Consider a linear tridiagonal system $Ax = b$ where

$$A = \begin{bmatrix} B & C & 0 & 0 & \dots & 0 & 0 & 0 \\ C & B & C & 0 & \dots & 0 & 0 & 0 \\ 0 & C & B & C & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & C & B & C \\ 0 & 0 & 0 & 0 & \dots & 0 & C & B \end{bmatrix}$$

is a block tridiagonal matrix and B and C are square matrices of order $n \geq 2$. It is assumed that there are M such blocks along the principal diagonal of A, and $M = 2^k - 1$, for some $k \geq 2$. Thus, $N = Mn$ denotes the order of A. It is assumed that the vectors x and d are likewise partitioned, that is, $x = (x_1, x_2, \dots, x_M)^t$, $d = (d_1, d_2, \dots, d_M)^t$, $x_i = (x_{i1}, x_{i2}, \dots, x_{in})^t$, and $d_i = (d_{i1}, d_{i2}, \dots, d_{in})^t$, for $i = 1, 2, \dots, M$. It is further assumed that the blocks B and C are symmetric and commute.

Equations of this type are known to arise in discretizing a certain class of partial differential equations of the elliptic type, using the idea of separation of variables [7]. A version of the parallel algorithm specification taken from [20] is given below. The algorithm has a reduction phase in which the system is split into two subsystems - one for odd-indexed (reduced system) and another for even-indexed terms (eliminated system). The reduction process is repeatedly applied to the reduced system. The indexed terms for B, C, d refer to computed terms during iterations of the reduction process. After $k-1$ iterations the reduced system contains the solution for a single term. The rest of the terms can be obtained by back-substitution.

B(0) = B; C(0) = C; $d_i(0) = d_i$; */* INITIALIZATION */*

FOR j=1 TO k-1 STEP 1 DO IN PARALLEL / REDUCTION PHASE */*

  *B(j) = 2 ** $C^2(j-1)$ - $B^2(j-1)$

  C(j) = $C^2(j-1)$

  $d_i(j) = C(j)[d_{i-h}(j-1) + d_{i+h}(j-1)] - B(j-1)d_i(j-1)$,

  where $h = 2^{j-1}, i = 2^j, 2 \times 2^j, 3 \times 2^j, (2^{k-j} - 1)2^j$

Solve for $x_{2^{k-1}}$ *in* $B(k-1)x_{2^{k-1}} = d_{2^{k-1}}(k-1)$

FOR j=k-1 TO 1 STEP -1 DO IN PARALLEL / BACK-SUBSTITUTION PHASE */*

Solve E(j)w(j) = y(j), where

$$E(j) = \begin{bmatrix} B(j-1) & 0 & 0 & \dots & 0 & 0 \\ 0 & B(j-1) & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & B(j-1) & 0 \\ 0 & 0 & 0 & \dots & 0 & B(j-1) \end{bmatrix}, \; w(j) = \begin{bmatrix} x_{t-s} \\ x_{2t-s} \\ \vdots \\ x_{it-s} \\ \vdots \\ x_{2^{k-j}}t - s \end{bmatrix},$$

$$y(j) = \begin{bmatrix} d_{t-s}(j-1) - C(j-1)x_t \\ d_{2t-s}(j-1) - C(j-1)[x_{2t} + x_t] \\ \vdots \\ d_{it-s}(j-1) - C(j-1)[x_{it} + x_{(i-1)t}] \\ \vdots \\ d_{2^{k-j}t-s}(j-1) - C(j-1)x_{(2^{k-j}-1)t} \end{bmatrix}$$

where $t = 2s = 2^j$.

A constraint program specification for this example for $k = 3$ in our system is given below. The results of the reduction process are stored in BP, CP, dP. *pow* is a C function implementing the arithmetic power function. Without getting into the details of the program specification, it can be seen that the specification is very similar to the algorithm specification. This example has been programmed and successfully translated by our system. At the current time, the shared memory Sequent machine is not available for access. Hence we moved to the distributed memory PVM system. The performance results for $M = 31$ and $n = 150$ are shown in Figure 10.

This speedups obtained on the PVM system are not very impressive as this is a very communication-intensive problem and careful examination of the generated code shows that the native pvm program would not do any better. This problem is ideal for executing on a shared memory architecture with computation being done on shared variables.

```
BP[0] == B AND CP[0] == C AND
AND for (i 1 7)  dP[i][0] == d[i]  AND
AND for (j 1 2)
          BP[j] == 2 * CP[j-1] * CP[j-1] - BP[j-1] * BP[j-1]
          CP[j] == CP[j-1] * CP[j-1]
          AND for (i 1 pow(2,3-j)-)1
              dP[i*pow(2,j)][j] == CP[j-1] * ( dP[i*pow(2,j) + pow(2,j-1)][j-1] +
                          dP[i*pow(2,j) - pow(2,j-1)][j-1] ) + BP[j-1] * dP[i*pow(2,j)][j-1]  AND
```

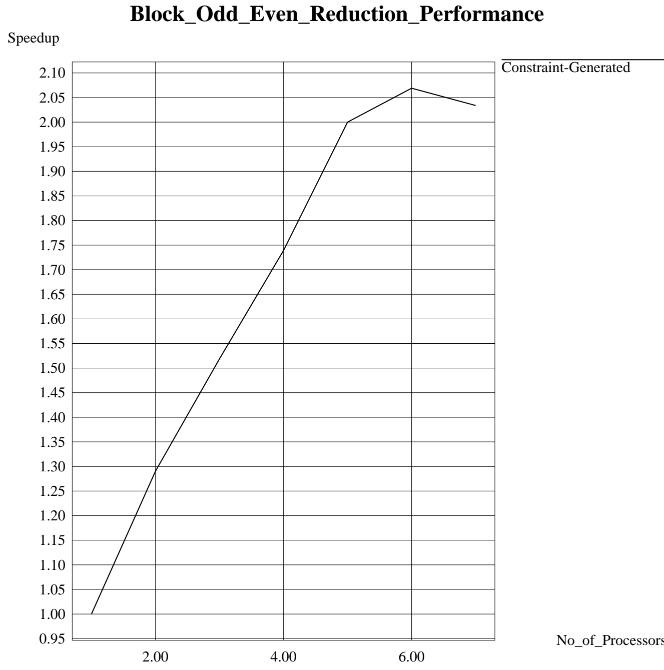

Figure 10: Performance Results for the Block Odd-Even Reduction Algorithm

```
BP[2] * x[4] == dP[4][2] AND
AND for (j 2 1)
        AND for (i 0 pow(2,3-j)-1)
            BP[j-1] * x[(i+1)*pow(2,j)-pow(2,j-1)] ==
                dP[(i+1)*pow(2,j)-pow(2,j-1)][j-1] - CP[j-1] * ( x[(i+1)*pow(2,j)] - x[i*pow(2,j)] )
```

7 Related Work

There has been considerable work in constraint programming in the past decade. Three major pieces of work that are related to the research are described in this section. Consul [2] is a parallel constraint language that was developed as a vehicle to experiment with implicitly parallel, constraint-based programming. The ThingLab and the Kaleidoscope projects [6, 5, 23] came from the University of Washington. Concurrent constraint programming (CCP) languages [26] established the foundations of concurrent computation with constraints.

Consul: This work resembles our work in that the goal is to extract parallelism from constraints [2]. But the approach is different in that local propagation is used to find satisfying values for the system of constraints. This approach has little hope of extracting efficient programs

and offers offers speedups only in the range of logic languages.

Thinglab and Kaleidoscope: Thinglab [6, 5] was a constraint-oriented graphic simulation laboratory. Constraints were compiled to sequential procedural code. Constraint Imperative Paradigm (CIP) was born out of the integration of imperative programming and constraint programming. It could be basically seen "either as adding constraints to imperative programs, or as adding control flow to constraint programs" [4]. Kaleidoscope [4, 3, 23] integrates object-oriented programming with constraints. Both these projects extracted procedural code from constraints but no parallelism was exploited.

Concurrent Constraint Programming: There has been much recent work in the integration of logic programming with constraints leading to the development of Constraint Logic Programming Languages. This has involved the merging of two domains - the Herbrand universe for predicates and some other domain like reals for constraints. CLP(D) [17, 9] defines a general scheme for constraint logic programming. Different implementations of the scheme are described in languages like Prolog III, CLP(R), and CHIP [12, 18, 13, 16, 22] Vijay Saraswat described a family of concurrent constraint logic programming languages,the cc languages [26]. The logic and constraint portions are explicitly separated with the constraint part acting as an active data store for the logic half. The logic communicates with the constraint part only through constraints either by a "tell" operation (a new constraint is added to the store) or an "ask" operation (to check if a constraint is consistent with the store).

8 Summary and Future Research

In conclusion we claim that constraint programs offer a rich, relatively untapped resource for parallelism. Constraint systems with appropriate initialization specification can be mapped to a generalized dataflow graph. Coarse-grain parallelism can be extracted through modularity, operations over structured types, and specification of arithmetic functions. By giving the programmer

control over compilation choices for the execution environment, we assist in generation of architecturally optimized parallel programs. The first stage of research has established that constraint systems can be compiled to efficient coarse grained parallel programs for some plausible examples. In fact, we have been quite surprised at the ease of attaining these results from such a radical representation.

This paper reports on the first step in the quest for 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 be 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 one possibility. A simpler and more algorithmic basis for definition of constraints over partitions of matrices is to utilize a simple version of the hierarchical type theory for matrices recently published by Collins and Browne [10]. The hierarchical type model for matrices establishes a compilable semantics for computations over hierarchical matrices.

The next steps in this research are:

a) Design and implement a "block" structure in which destructive update to variables is allowed. This structure will be encapsulated in the sense that the interface will be like a constraint. This feature will allow compact specification of many parallel numerical algorithms which are awkward using only single assignment variables.

b) Implement hierarchical matrices.

c) Extend the AND loop construct to handle more general forms of constraints.

d) Define the semantics of and implement recursion in constraint module calls.

e) Include the execution environment specification as part of the input specification. This will provide compiler optimizations which take advantage of architectural characteristics of specific

execution environments.

f) To validate the compiler by performance measurements of applications on several parallel architectures.

References

[1] Anderson, T.E. et. al. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Trans. on Computers*, vol. 38, no. 12, Dec. 1989.

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

[3] B. Freeman-Benson and Alan Borning. The design and implementation of Kaleidoscope '90, a constraint imperative programming language. Proceedings of the IEEE Computer Society International Conference on Computer Languages, pages 174-180, April 1992.

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

[5] 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.

[6] 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.

[7] B.L. Buzbee, G.H.Golub, and C.W. Nielson. On Direct Methods for Solving Poisson's Equations. *SIAM Journal on Numerical Analysis*, Vol 7, pp 627-655.

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

[9] Jacques Cohen. Constraint Logic Programming. Communications of the ACM, 33(7):52-68, July 1990.

[10] 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.

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

[12] A. Colmerauer. An introduction to Prolog III. Presented at the Workshop on Languages and Constraints, Brandeis University, Aprill 1988.

[13] M. Dincbas, P. van Hentenryck, H. Simonis, F. Berthier. The constraint logic programming language CHIP. *Proceedings of the FGCS Conference*, November 1988.

[14] 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.

[15] R. Eigenmann and W. Blume. An Effectiveness Study of Parallelizing Compiler Techniques. *Proc. Intl. Conf. Par. Proc., 1991, pp. II 17-25.*

[16] Pascal Van Hentenryck. *Constraint Satisfaction for Logic Programming.* MIT Press, Cambridge, MA, 1989.

[17] Joxan Jaffer and Jean-Louis Lassez. Constraint Logic Programming. *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 111-119, ACM, January 1987.

[18] Joxan Jaffar and Spiro Michaylov. Methodology and Implementation of a CLP system. Proceedings of the 4th International Conference on Logic Programming, pages 196-218, Melbourne, May 1987.

[19] Ajita John, J. C. Browne. Compilation of a Constraint Program into Parallel C Programs. Technical Report TR95-1, Department of Computer Sciences, Unversity of Texas at Austin, 1995.

[20] S. Lakshmivarahan and Sudarshan K. Dhall. Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems. McGraw-Hill Series in Supercomputing and Parallel Processing, 1990.

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

[22] Pierre Lim and Peter J. Stuckey. A Constraint Logic Programming Shell. Technical report, Department of Computer Science, Monash University, 1989.

[23] 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.

[24] 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.

[25] 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.

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