Compilation of Constraint Systems to Procedural Parallel Programs

Ajita John and J. C. Browne

Dept. of Computer Sciences
University of Texas, Austin, TX 78712
{ajohn,browne}@cs.utexas.edu

Abstract. This paper describes the first results from research[1] on the compilation of constraint systems into task level parallel programs in a procedural language. This is the only research, of which we are aware, which attempts to generate efficient parallel programs for numerical computations from constraint systems. Computations are expressed as constraint systems. A dependence graph is derived from the constraint system and a set of input variables. The dependence graph, which exploits the parallelism in the constraints, is mapped to the target language CODE, which represents parallel computation structures as generalized dependence graphs. Finally, parallel C programs are generated. The granularity of the derived dependence graphs depends upon the complexity of the operations represented in the type system of the constraint specification language. 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 execution environment or software architecture is specified separately from the constraint system. The domain of matrix computations has been targeted for applications. Some examples have been programmed. Initial results are very encouraging.

1 Introduction

Representing a computation as a set of constraints upon the state variables defining the solution and choosing an appropriate subset of the state variables as the input set 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 [1]. There are however, both motivations for continuing research in this direction and reasons for optimism concerning success. Constraint systems have attractive properties for compilation to parallel computation structures. A constraint system gives the minimum specification (See [2] for the benefits from postponing imposition of program structure) for a computation, thereby offering the compiler freedom of choice

[1] This work was supported in part through a grant from the Advanced Research Projects Office/CSTO, subcontract to Syracuse University #3531427

for derivation of control structure. Constraint systems offer some unique advantages as a representation from which parallel programs are to be derived[11]. Both "OR" and "AND" parallelism can be derived. Either effective or complete programs can be derived from constraint systems on demand. Programs for different computations can be derived from the same constraint specification by different choices of the input set of variables.

The focus of this research is to derive from constraint representations, parallel programs of execution efficiency competitive with procedural languages. This paper reports early results from this approach. The next two sections outline our approach and related work, respectively. Subsequent sections describe the constraint specification language and the compilation algorithm. We conclude the paper with performance results for some example programs and directions for future work.

2 Approach

A program is expressed as a set of constraints between the program variables and an input set consisting of a subset of the program variables. A dependence graph is derived from the program and mapped to the target language CODE [14], which expresses parallel structure over sequential units of computation declaratively as a generalized dependence graph. The software architecture or execution environment to which CODE is to compile is separately specified (SMP, DSM, NOW, etc). Finally, sequential and parallel C programs for shared memory machines like the CRAY J90, SPARCcenter 2000, and Sequent and the distributed memory PVM [10] system can be generated. An MPI [7] backend for CODE is also available.

The granularity of the derived dependence graphs depends upon the types directly represented as primitives in the constraint representation. The introduction of structured types and operations on structured types as primitives in the constraint representation give natural units of computation at a granularity appropriate for task level parallelism and avoids the problem of name ambiguity in the derivation of dependence graphs from loops over scalar representation of arrays. It also supports implementation of data parallelism, if desired[11]. 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 detailed in the rest of the section.

2.1 Constraint Representation

A constraint is a relationship between a set of variables. E.g. $A + B == C$ is a constraint expressing equality between C and the sum of A and B. A constraint system enumerates the relationships between the variables of the computation.

2.2 Constraint Modules

Modularity is provided by *constraint modules*, which are encapsulations of relationships between parameters and can be invoked as a constraint. Figure 1 shows a constraint specification (excluding declarations) for the non-complex roots of a quadratic equation, $ax^2 + bx + c == 0$. The specification uses a module, *DefinedRoots*. $''U''$ denotes an undefined value. *sqr*, *sqrt*, and *abs* are library functions. A program specification also identifies the set of inputs. In the example, it could be $\{a, b, c\}$, or $\{a, b, r1\}$, or $\{a, b, r2\}$.

```
/* 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)))


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

Fig. 1. Quadratic Equation Solver

2.3 Translation to a Compilable Language

Encapsulated within the constraint $A + B == C$ are three assignments: $A = C - B$, $B = C - A$, and $C = A + B$; and a conditional, $A + B == C$. One of the three assignments can be extracted depending on which two of the three variables $\{ A, B, C \}$ are inputs. If all three variables are inputs, the constraint can be classified as a conditional to be checked for satisfaction. If fewer than two variables are inputs, the constraint is unresolved and no resulting program can be extracted.

A dependence graph can be derived from a set of constraints and an input set of variables. This graph ensures satisfaction of the constraints by computing values for some or all of the non-input (output) variables. In addition, parallelism in the computations is exploited. The compiler generates single-assignment variables and can extract multiple solutions on alternate paths of the dependence graph. The derivation of dependence graphs is explained in detail in Section 5.

A constraint specification represents a family of dependence graphs. Generation of all possible dependence graphs can result in combinatorial explosion. We construct only the dependence graph for a specified input set. The constraint specification can be reused for generating dependence graphs for different sets of inputs.

The dependence graph for the quadratic equation solver with the input set $\{a, b, c\}$ is shown in Figure 2. If the conditionals on an arc are satisfied, the

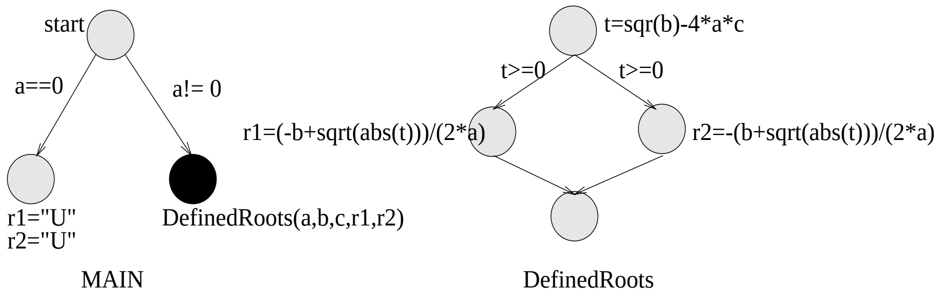

start
a==0
a!= 0
r1="U"
r2="U"
DefinedRoots(a,b,c,r1,r2)
MAIN
t=sqr(b)-4*a*c
t>=0
t>=0
r1=(-b+sqrt(abs(t)))/(2*a)
r2=-(b+sqrt(abs(t)))/(2*a)
DefinedRoots

Fig. 2. Dataflow Graphs for Quadratic Equation Solver Example

corresponding destination node is executed. If a node is executed, the computations (assignments) at that node are performed. The node annotated by *DefinedRoots* in *MAIN* invokes the dependence graph corresponding to the module *DefinedRoots*. The assignments $r1 = (-b + sqrt(abs(t)))/(2 * a)$ and $r2 = -(b + sqrt(abs(t)))/(2 * a)$ are performed in parallel and the results are collected by a node.

2.4 Domain Specification: The Hierarchical Type System

The semantic domain chosen for our application programs is matrix computation. We have a built-in matrix type with its associated operations of addition, subtraction, multiplication and inverse. The matrix subtypes currently implemented in our system are lower and upper triangular and dense matrices. We plan to extend the type system to a richer class of matrices including hierarchical matrices [4]. Specialized algorithms based on the structure of the matrix can be invoked for the matrix subtypes. Other structured types such as lists, queues, trees etc. along with their associated operations could be included to broaden the class of programs which can be compactly represented.

2.5 Separate Specification of Compilation Options and the Execution Environment (Software Architecture)

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.
(a) Specification of architecture-specific mechanisms (E.g. shared variables or messages for communication, etc).
(b) Selection of the level of granularity for operations.
(c) Option of not parallelizing a particular module.
(d) Option of selecting certain operations for executing in parallel.
(e) Choices among parallel algorithms to execute some of the operations.

3 Related work

We shall briefly sketch related pieces of work in this section.

Consul[1] is a parallel constraint language that uses an interpretive technique (local propagation) to find satisfying values for the system of constraints. This system offers performance only in the range of logic languages. Declarative extensions have been added as part of High-Performance Fortran(HPF) [15], a portable data-parallel language with some optimization directives. HPF does not support task parallelism. Also, existing control flow in its procedural programming style makes analysis for parallelism difficult. Thinglab [9] transforms constraints to a compilable language rather than to an interpretive execution environment as in many constraint systems. Kaleidoscope[8] integrates constraints with an object-oriented language and uses partial compilation for the constraints. Neither Thinglab nor Kaleidoscope is concerned with extraction of parallel structures. Vijay Saraswat described a family of concurrent constraint logic programming languages, the *cc* languages [16]. The logic and constraint portions are explicitly separated with the constraint part acting as an active data store for the logic portion of the program. Oz is a concurrent programming language based on an extension of the basic concurrent constraint model provided in [16]. The performance reported for the system is only comparable with commercial Prolog and Lisp systems [13]. Parallel logic programming [3, 6] is another area of related work. PCN [3] and Strand [6] are two parallel programming representations with a strong component of logic specification. However, both require the programmer to provide explicit operators for specification of parallelism and the dependence graph structures which could be generated were restricted to trees. Equational specifications of computations is a restriction of constraint specifications. Unity [2] is the equational programming representation around which Chandy and Misra have built a powerful paradigm for the design of parallel programs. Again, Unity requires addition of explicit specifications for parallelism.

4 Language Description

This section describes the different components of the programming system. It includes the types and their associated operations, the rules for constructing constraints, and the structure of a complete program in the system. The notations used are similar to those in the C programming language.

4.1 Types

The lowest level of the type system consists of integers, reals, characters, and arrays with the operators of addition, subtraction, multiplication, division on integers and reals. At the next level of the type hierarchy are matrices with their associated operations of addition, subtraction, multiplication and inverse. As mentioned in Section 2, the system currently supports specialized matrix types like lower and upper triangular. At the highest level of the type system are hierarchical matrices, whose individual elements are matrices.

4.2 Constraints

In our system, arithmetic expressions can be formed by using arithmetic operators and calls to library and user-defined functions (functions must have defined inverses, otherwise only a limited form of compilation can be done). In addition, expressions of the following form, using *indexed operators*, are allowed.

$$< op > \ FOR \ (< index > < b1 > < b2 >) \ X$$

An indexed operator applies a binary $op \in \{+, -, *, /\}$ over an arithmetic expression, X, through a range of values, $b1 \ldots b2$, for an integer variable, *index*. E.g. + FOR (i 1 5) $A[i]$ specifies the sum of the elements in array A between positions 1-5. *b1* and *b2* have to be statically bound.
Constraints can be constructed by application of the following rules.

Rule 1: (i) $X_1 \ R \ X_2$, is a constraint,
where $R \in \{ <, <=, >, >=, == , != \}$, X_1, X_2 are arithmetic expressions.
(ii) $M_1 == M_2$ is a constraint,
where M_1, M_2 are expressions involving matrices and the matrix operators +, -, *, and Inverse.

Rule 2: (i) A AND/OR B (ii) NOT A are constraints,
where A and B are constraints.

Rule 3: Calls to user-defined constraint modules are constraints. In Figure 1, the call *DefinedRoots(a,b,c,r1,r2)* in *Main* is an application of this rule.

Rule 4: Constraints over indexed sets have the form:

$$\text{AND/OR FOR (<index> <b1> <b2>) } \{ A_1, A_2, \ldots, A_n \}$$

The above construct groups a set of constraints, $A_1, A_2, \ldots, A_n$, to be connected by an AND/OR connective through a range of values, $b1 \ldots b2$, for an integer variable, *index*. *b1* and *b2* have to be statically bound. This condition will be relaxed in later versions of the compiler.
An instance of Rule 4 is AND FOR (i 1 2) { A[i] == A[i-1], B[i+1] == A[i] }. This example succinctly represents the constraints
$A[1] == A[0]$ AND $A[2] == A[1]$ AND $B[2] == A[1]$ AND $B[3] == A[2]$.

Constraints constructed from applications of Rule 1 are referred to as *simple constraints*, which form the building blocks for constraints constructed from applications of Rules 2-4.

4.3 Programs

A program in our system consists of the following constituents.
(i) Program name, global variable declarations, global input variables.

(ii) User-defined function signatures: signatures of C functions (linked during execution), which may be invoked in an arithmetic expression.

(iii) Constraint Module definitions: module name, formal parameters and their types, local variable declarations, and a body constructed from applications of Rules 1-4 in Section 4.2.

(iv) Main body of the program: constraints formed from applications of Rules 1-4 in Section 4.2.

5 Compilation

The compilation algorithm consists of the following phases.

Phase 1. The textually expressed constraint system is transformed to an undirected graph representation.

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 conditionals and computations for nodes of a generalized dependence graph.

Phase 4. Specifications of the execution environment are used to optimally select the communication and synchronization mechanisms to be used by CODE [14]. This phase is yet to be completely defined.

Phase 5. The dependence graph is mapped to the CODE parallel programming environment to produce sequential and parallel programs in C as executable for different parallel architectures.

Phases 1-5 are described in detail in Sections 5.1-5.5.

5.1 Phase 1

The textual source program is transformed to a source graph for the compiler. Starting from an empty graph, for each application of Rules 1-4 in Section 4.2, an undirected constraint graph can be constructed by adding appropriate nodes and edges to the existing graph. For each instance of a simple constraint (Rule 1), a node is created with the constraint attached to it. For each application of Rule 2, the graph is expanded as shown in Figures 3(a)-(b). For each application of Rule 3, a node is created with the constraint module call and the actual parameters attached to it. For each application of Rule 4, the graph is expanded as shown in Figure 3(c).

The different kinds of nodes in the constraint graph are (i) *simple constraint* nodes, (ii) *operator* nodes corresponding to AND/OR/NOT connectives, (iii) *call* nodes corresponding to Constraint Module Calls, and (iv) *for* nodes corresponding to indexed sets.

A set of constraint graphs is constructed from the main body and the constraint module bodies. Each graph is constructed in a hierarchical fashion. *Simple constraint* nodes and *call* nodes occur at lower levels. At higher levels, *operator* and *for* nodes connect one or more subgraphs. There will be a unique node

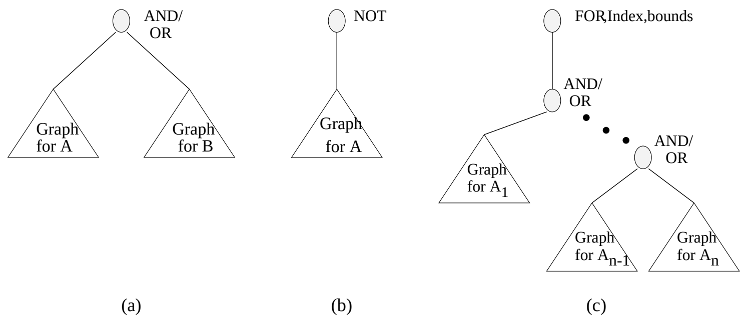


Fig. 3. Construction of Constraint Graphs for (a),(b) Rule 2 (c) Rule 4

at the highest level. The constraint graph obtained for a particular constraint specification is unique. The constraint graphs for the quadratic equation solver are shown in Figure 4.

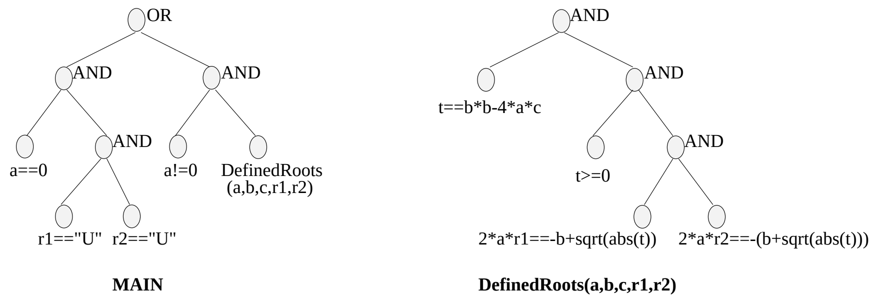


Fig. 4. Constraint Graphs for Quadratic Equation Solver

5.2 Phase 2

A depth-first traversal of each constraint graph constructs a set of directed graphs (trees). The traversal assigns constraints connected by AND operators in a constraint graph to the same node in the corresponding tree and constraints connected by OR operators in a constraint graph to nodes on diverging paths in the corresponding tree. The satisfaction of all the constraints along a path from the root to a leaf in a tree represents a satisfaction of the constraint system represented by the tree.

Figure 5 illustrates phase 2 for four base cases, where a, b, c, and d are simple constraints. The algorithm *dfs* is a generalization of Figure 5. Let v_1 be the unique node at the highest level of the input constraint graph, G. Each output tree, G^*, is initialized to a root, v_1^*. Each node in G^* can hold a list of constraints. An indexed set of constraints within a node in G^* has an associated tree obtained from the depth-first traversal of the constraint graph corresponding to constraints in the indexed set. v_c and v_c^* are the nodes currently being visited in G and G^*, respectively. dfs is invoked with the call dfs(v_1, v_1^*). The tree obtained for the quadratic equation solver is shown in Figure 6.

The case of *operator* node, NOT, has been omitted from the description of *dfs*. However, it is implemented in the system as follows. A NOT operator node operates on a single (constraint) subgraph. If the subgraph is a simple constraint, the NOT node is removed by negating the simple constraint. Otherwise, a NOT node is moved down all the levels of the subgraph by changing nodes (AND to OR and OR to AND) traversed in its path until it reaches a simple constraint, which is negated.

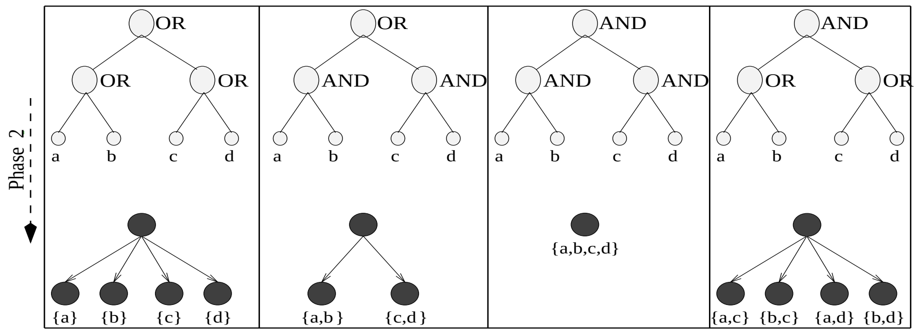


Fig. 5. Phase 2

```
ALGORITHM 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 is an unvisited OR neighbor, $u_1$, of $v_c$
                let $u_2$ be the other neighbor of $v_c$;
                let $u_{11}$ and $u_{12}$ be the two unvisited neighbors of $u_1$;
                /* transform ($u_{11}$ OR $u_{12}$) AND $u_2$ to ($u_2$ AND $u_{11}$) OR ($u_2$ AND $u_{12}$) */
                visited[$v_c$] = false;
                change type of $v_c$ to OR, remove $u_1$, $u_2$ as neighbors of $v_c$;
                create two unvisited AND neighbors, $and_1$, $and_2$, for $v_c$;
                make $u_2$ and $u_{11}$ the neighbors of $and_1$;
                make $u_2$ and $u_{12}$ the neighbors of $and_2$;
                dfs($v_c$, $v_c^*$);
            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 $v_c^*$;
        For Node : attach indexed set with index and bounds to $v_c$*
                create new root $v_i$* for tree corresponding to indexed set;
                let $v_i$ be the node at highest level of constraint graph for indexed set;
                dfs($v_i$, $v_i$*);
end;
```

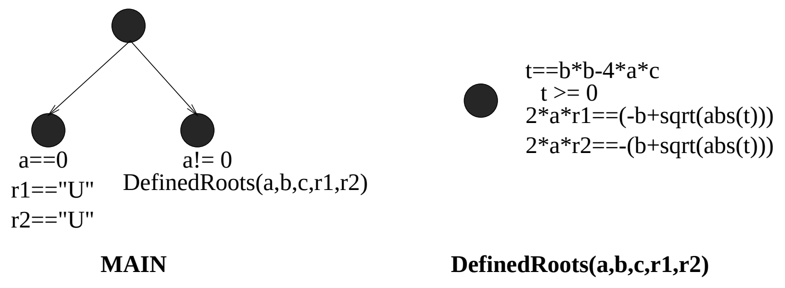

a==0
r1=="U"
r2=="U"
a!= 0
DefinedRoots(a,b,c,r1,r2)
MAIN
t==b*b-4*a*c
t >= 0
2*a*r1==(-b+sqrt(abs(t)))
2*a*r2==-(b+sqrt(abs(t)))
DefinedRoots(a,b,c,r1,r2)

Fig. 6. Trees from Phase 2 for Quadratic Equation Solver

5.3 Phase 3

Using the input set specification, a depth-first traversal of the tree from phase 2 attempts to generate a dependence graph. The generated dependence graph is a directed graph, in which nodes are computational elements and arcs between nodes express data dependency. It has a unique *start* node. A path from the start node in the graph is a computation path. A node in the dependence graph has the form: firing rule, computation, routing rule (see Figure 7(a)). A firing rule is a condition that must hold before the node can be enabled for execution. The computation at a node is performed when the node is executed. A routing rule is a condition that must hold for the node to send data on its outgoing paths. The nodes and arcs in the tree from phase 2 correspond to the nodes and arcs, respectively, in the dependence graph.

A *known* set is associated with each node in the dependence graph. The variables in the known set at a node are *knowns* at that node. The values of these variables are known at runtime at that node. All variables not in the known set at a node are *unknowns* at that node. The input set is cast as the *known* set for the start node.

The depth-first traversal tries to generate a dependence graph with nodes that contain computations for the *unknowns*. Each node, v, in the tree from phase 2 has a set of simple constraints attached to it. When v is visited, each constraint can be *resolved* as one of the following for the corresponding node, v^*, in the dependence graph.
(i) Firing Rule: To be so classified, a constraint must have no unknowns when v 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 for v^*.
(iii) Routing Rule: To be a routing rule, all unknown variables in the constraint must become knowns through computations at v^*.

An indexed set, AND/OR FOR (<index> <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$, is resolved if every constraint A_i resolved for all values of *index* in $b1 \ldots b2$. Resolved indexed sets are compiled to loops which iterate over values of *index* in $b1 \ldots b2$ The restrictions on the indexed set structure to be compiled successfully in our system are as follows. For all values of *index* in $b1 \ldots b2$ (a) a constraint has to have the same classification, (b) if a constraint is classified as computation, a single unique term in the constraint has to be the unknown.

Constraints involving inequalities must be resolved as firing/routing rules. Any constraint which cannot be resolved is retained in an unresolved set of constraints which is propagated down the tree. A node in the dependence graph receives the known set of its parent as its known set. Examination of each constraint at a node and in the unresolved set of constraints continues until no new variables are added to the known set. Any path in the tree that results in a leaf with unresolved constraints is abandoned. If all paths in the tree are abandoned, the user is informed of the restrictiveness of the initial input set.

When a constraint is classified as computation, it is mapped to an equation. All terms involving the single unknown in the computation are moved to the left-hand side of the equation. Currently, our system solves equations in linear unknown terms. In the future we plan to incorporate solvers for scalar types that will solve for higher powers of the unknown. If the variables in the computation are matrices, the computation 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 invoked 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.

The dependence graphs for the quadratic equation solver with the input set $\{ a, b, c \}$ is shown in Figure 7(b).

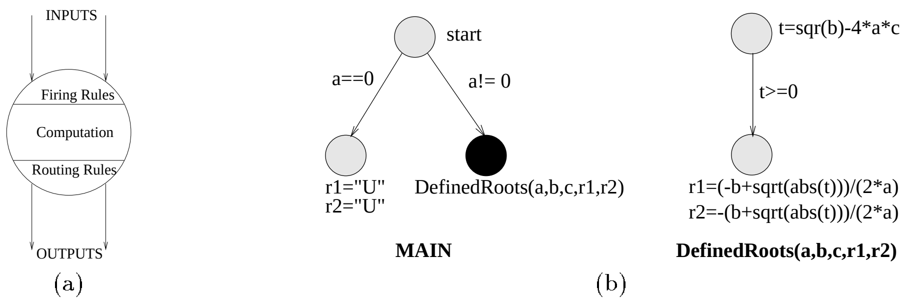


Fig. 7. (a) Dependence Graph Node (b) Dependence Graphs for Quadratic Equation Solver

5.3.1 Phase 3 for Constraint Module Calls

A constraint module call has the form $ModuleName(e_1, e_2, \ldots, e_n)$ where e_i, $1 \le i \le n$, is an arithmetic expression and an actual parameter. Let the formal parameters corresponding to $e_1, e_2, \ldots, e_n$ be $f_1, f_2, \ldots, f_n$, respectively. Let K be the known set at the node where the constraint module is invoked. An attempt is made to generate a dependence graph from the constraint module definition. The tree from phase 2 for the constraint module is traversed with a new known set, $K_{module} = \{ f_i \mid \{\text{all variables in } e_i\} \subseteq K \}$. The unknowns are considered to be all formal parameters not in K_{module} and the local variables in the constraint module. The depth-first traversal of the tree corresponding to the constraint module returns "True" if all the constraints in the module are resolved and every unknown parameter is computed at the end of at least one path in the resulting dependence graph (All paths not satisfying this condition are discarded). This condition is different in the dependence graph generation of the main module where all unknown variables need not be computed. The reason for imposing this condition is that the actual parameters are bound to the formal parameters at the point of call. If different sets of variables are computed in different paths of the dependence graph (as in the constraint $a == c$ OR $b == c$ with c known) it is not possible to determine statically the set of actual parameters computed in the constraint module call.

If the dependence graph generation is successful, a new set of constraints corresponding to each computed parameter is 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 actual computed parameters. An attempt is made to resolve this set of constraints with $Z_1 \ldots Z_p$ in the known set. The set of constraints will have to be resolved as computation for all the unknowns in $e_{k1} \ldots e_{kp}$. If this is done, a call node (which invokes the dependence graph for the constraint module call) 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 dependence 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 dependence graphs generated for combinations of parameter inputs. Redundant translations can be eliminated this way.

5.3.2 Extraction of OR parallelism

Multiple paths in the dependence graph have the potential to be executed in parallel. These paths have resulted from the extraction of computation from constraints connected by OR operators.

5.3.3 Extraction of AND parallelism

The computational statements that are assigned to a node have the potential for parallel execution. Parallelism is exploited by keeping in mind that the compiler generates a single-assignment system and the lone write to a variable will appear before any reads to it. The granularity of such a scheme depends on the complexity of the functions invoked in the statements and the complexity of the operators. We have further exploited the complexity of matrix operations by splitting up the specifications, performing computations in parallel and composing them. For example, if $x = m*y+m*z$, where x, m, y, z, and b are matrices, $m*y$ and $m*z$ can be done in parallel. This leads to significant speedup since multiplication of matrices is an O(N^3) operation (m, y, z being order $N \times N$).

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. For a detailed description of the types of parallelism extracted see [11].

Parallelism in AND indexed sets: To extract parallelism, the computations within the compiled loop structures corresponding to *AND* indexed sets are examined. We discuss the case of loops with a single computation. The discussion can be generalized to the case of loops with multiple computations. Throughout this discussion, the case of array accesses will be detailed. The case of scalar accesses in loops will follow trivially since they do not involve indexed terms.

(i) If the array corresponding to the computed term is not accessed anywhere in the computation, all iterations of the loop can be executed in parallel.

(ii) If the array corresponding to the computed term is accessed in the computation and the set of accessed indices of the array are disjoint from the set of computed indices of the array, all iterations of the loop can be executed in parallel.

(iii) If cases (i) and (ii) do not hold, the loop iterations are inter-dependent and are executed sequentially.

5.4 Phase 4

We have yet to completely define Phase 4. As of now, there are provisions in the system to select certain program variables as shared variables in a shared memory environment. Also, some operations (e.g. matrix multiplication) can be chosen for parallel execution.

5.5 Phase 5

Our target for executable for constraint programs is the CODE [14] parallel programming environment. CODE takes a dependence graph as its input. The form of a node in a CODE dependence graph is given in Figure 7(a). It is seen that there is a natural match between the nodes of the dependence graph developed by the constraint compilation algorithm and the nodes in the CODE graph. The arcs in the dependence graph in CODE are used to bind names from one node to another. This is exactly the role played by arcs in the dependence graph generated by our translation algorithm. CODE produces sequential and parallel C programs for a variety of architectures.

6 Programming Examples and Results

A prototype of the compiler has been implemented in C++. A small number of examples have also been programmed and executed on the Cray J90, SPARC-center 2000, Sequent Symmetry machine and the PVM system. The next two subsections describe two examples programmed in our system.

6.1 Block Triangular Solver(BTS)

The example chosen is the solution of the $AX = B$ linear algebra problem for a known lower triangular matrix A and vector B. The parallel algorithm [5] involves dividing the matrix into blocks and a constraint program (excluding declarations) for a problem instance split into 4 blocks is shown in Figure 8. $S_0 \ldots S_3$ represent lower triangular sub-matrices along the diagonal of A that are solved sequentially, and $M_{10}, M_{20}, \ldots M_{32}$ represent dense sub-matrices within A that must be multiplied by the X sub-vector from above and the result subtracted from the X sub-vector from the left. The vector multiplications for all Ms within a column may be done in parallel. This parallelism yields an ideal asymptotic speedup of $P^2/(3P-2)$, where P is the number of processors. Ideal speedup assumes zero communication and synchronization times.

$$\boxed{\begin{array}{l} (\ S_0 * X_0 == B_0\ AND \\ M_{10} * X_0 + S_1 * X_1 == B_1\ AND \\ M_{20} * X_0 + M_{21} * X_1 + S_2 * X_2 == B_2\ AND \\ M_{30} * X_0 + M_{31} * X_1 + M_{32} * X_2 + S_3 * X_3 == B_3\ ) \end{array}}$$

Fig. 8. Specification of the BTS Algorithm as a Constraint System

The input set can be chosen as $\{\ S_0, \ldots, S_3, M_{10}, M_{20}, \ldots, M_{32}\ \}$. The output set will be detected as $\{X_0, X_1, X_2, X_3\}$. Using an indexed set of constraints and an indexed operator, an alternate compact program is
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.

Figure 9(a) gives the speedups for a 1200×1200 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. (These results were obtained with an earlier version of the constraint compiler before any provision for specification of execution environments were implemented and the Sequent computer system is no longer available, now that the optimizing compiler is available.) Figure 9(b) gives the speedups for a 8800×8800 matrix (number of blocks=11, block size=800) on an 8-processor SPARCcenter2000. This program has been optimized for shared variables. It is to be noted that the constraint generated code performs quite well for the small number of processors available. The speedups are slightly higher than the ideal (the ideal is asymptotic) until about 7 processors, after which it drops.

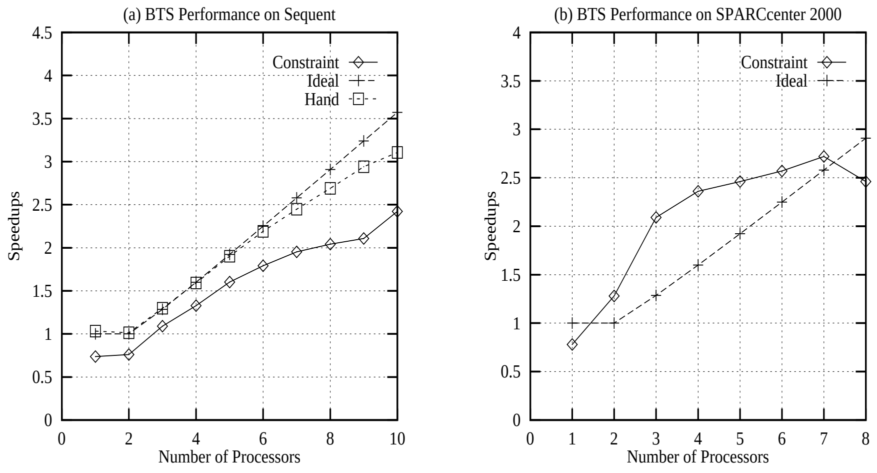


Fig. 9. Performance Results for BTS on (a) Sequent (b) SPARCcenter 2000

6.2 The Block Odd-Even Reduction Algorithm(BOER)

Consider a linear tridiagonal system $Ax = d$ 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.

A version of the parallel algorithm (taken from [12]) has a reduction phase in which the system is split into two subsystems: one for odd-indexed (reduced system) and another for even-indexed (eliminated system) terms. The reduction process is repeatedly applied to the reduced system. 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. The constraint specification for the problem is shown in Figure 10. The variable names, BP, CP, dP correspond to the indexed terms B,C,d in [12] and are examples of the hierarchical data type in our system (elements of BP, CP and dP are matrices). The inputs to the system are $BP[0]$, $CP[0]$ and $dP[i][0]$. *pow* is a C function implementing the arithmetic power function. The terms in bold in Figure 10 are detected by the compiler in phase 3 as the terms to be computed.

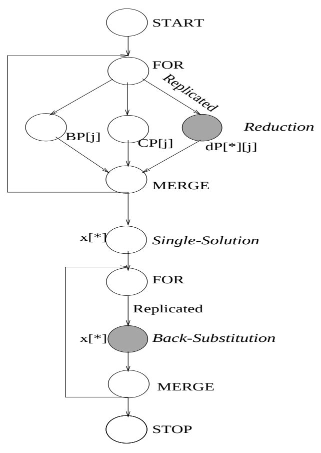
```
BP[k-1] * x[pow(2,k-1)] == dP[pow(2,k-1)][k-1] AND

AND FOR (j 1 k-1) {
    2 * CP[j-1] * CP[j-1] == BP[j] + BP[j-1] * BP[j-1] ,
    CP[j] - CP[j-1] * CP[j-1] == 0 ,
    AND FOR (i 0 pow(2,k-j)-2) {
        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] ) ==
                dP[i*pow(2,j)][j] + BP[j-1] * dP[i*pow(2,j)][j-1] }} AND


AND FOR (j k-1 1) {
    AND FOR (i 0 pow(2,k-j)-1) {
        CP[j-1] * ( x[(i+1)*pow(2,j)] + x[i*pow(2,j)] ) ==
            dP[(i+1)*pow(2,j)-pow(2,j-1)][j-1] - BP[j-1] * x[(i+1)*pow(2,j)-pow(2,j-1)] }}
```

Fig. 10. Specification of the BOER Algorithm as a Constraint System

The resulting dataflow graph is shown in Figure 11(a) and corresponds to the dataflow in the algorithm in [12]. The *START* and *STOP* nodes initiate and terminate the program, respectively. A *FOR* node initiates the different iterations of a loop. The two such nodes in the figure correspond to the two outer indexed sets for index j in the constraint specification. The annotation "Replicated" on the arcs specify that the annotated arc and the destination node (shaded in Figure 11(a)) are dynamically replicated for parallel execution. The two such annotated arcs correspond to the two nested indexed sets (for index i) in the constraint specification. The nodes annotated by *BP, CP, dP*, and x compute values for parts of the corresponding variable. The nodes annotated by "Merge" collect computed results from parallel executions. It is to be noted that our compiler automatically detects the parallelism in the *for* loops in the reduction and back-substitution phases. Furthermore, it is capable of detecting the parallelism within the expression $2 * CP[j-1] * CP[j-1] - BP[j-1] * BP[j-1]$ in the computation for $BP[j]$. The authors in [12] have mentioned that the single-solution step is the major bottleneck in the algorithm. But, in our experiments we found the reduction phase resistant to scalability. This is due to the fact that the computation for $BP[j]$ and $CP[j]$ involve matrix-matrix multiplication: an $O(n^3)$ operation. This dominates the scalable part of the loop: the computation of dP, which is $O(n^2)$. In later versions of the compiler, we plan to incorporate parallel algorithms for matrix-matrix multiplication, which will overcome this bottleneck. Figure 11(b) presents the speedups over a sequential implementation of the algorithm on an 8-processor SPARCcenter 2000 for n=200 and $k = 7 (M = 127)$ and $k = 8 (M = 255)$ for the back-substitution phase of the algorithm. Note the attainment of near-linear speedup for this (relatively small) number of processors.



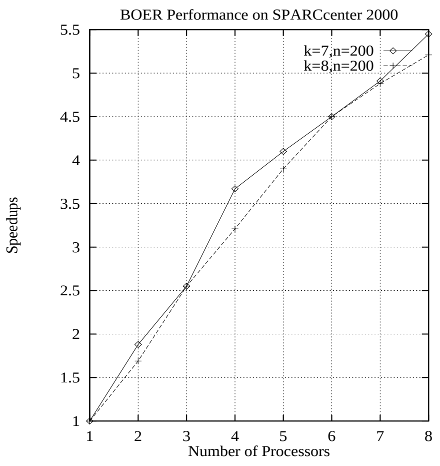


Fig. 11. BOER Program: (a)Dataflow Graph (b) Performance Results

7 Summary

Constraint programs offer a rich, relatively untapped representation of computation, from which parallelism can be readily extracted. Constraint systems with appropriate sets of input variables can be mapped to generalized dependence graphs. Coarse-grain parallelism can be extracted through modularity, operations over structured types, and specification of arithmetic functions. Data parallelism is introduced through the parallel execution of iterations of loops over computations on different partitions of matrices. 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.

8 Future Research

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 of Collins and Browne [4].

The next steps in addition to inclusion of the hierarchical matrix type are as follows. a) Enhance the compiler with the capability of converting single-assignment variables to "destructive-update" variables so that excessive memory usage can be avoided in iterative numerical algorithms. b) Extend the AND indexed set construct to handle more general forms of constraints. c) Define and implement completely the execution environment specification.

References

1. Doug Baldwin. A status report on consul. In Nicolau Gelernter and Padua, editors, *Languages and Compilers for Parallel Computing.* MIT Press, 1990.
2. K.M. Chandy and J. Misra. *Parallel Program Design : A Foundation.* Addison-Wesley, 1989.
3. K.M. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, 1992.
4. T.S. Collins and J.C. Browne. Matrix++: An object-oriented environment for parallel high-perfomance matrix computations. In *Proc. of the Hawaii Intl. Conf. on Systems and Software*, 1995.
5. J.J. Dongarra and D.C. Sorenson. Schedule: Tools for developing and analyzing parallel fortran programs. Technical Report 86, Argonne National Laboratory, November 1986.
6. I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.
7. Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.
8. B. Freeman-Benson and Alan Borning. The design and implementation of Kaleidoscope '90, a constraint imperative programming language. In *Computer Languages*, pages 174–180. IEEE Computer Society, April 1992.
9. Bjorn N. Freeman-Benson. A module compiler for Thinglab II. In *Proc. 1989 ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1989.
10. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine:A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.
11. Ajita John and J.C. Browne. Extraction of parallelism from constraint specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume III, pages 1501–1512, August 1996.
12. S. Lakshmivarahan and Sudarshan K. Dhall. *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems.* Supercomputing and Parallel Processing. McGraw-Hill, 1990.
13. Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An abstract machine for Oz. In *Programming Languages, Implementations, Logics and Programs, Seventh Intl. Symposium, LNCS*, number 982. Springer-Verlag, September 1995.
14. P. Newton and J. C. Browne. The CODE 2.0 graphical parallel programming environment. In *Proc. of the Intl. Conf. on Supercomputing,* pages 167–177, 1992.
15. Harvey Richardson. High Performance Fortran: History, overview and current developments. Technical Report TMC 261, Thinking Machines Corporation.
16. Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon, School of Computer Science, Pittsburgh, 1989.