Extraction of Parallelism from Constraint Specifications [1]

Ajita John
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, U.S.A.

J. C. Browne
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, U.S.A.

Abstract

This paper describes the extraction of parallel procedural programs from specifications of computations as constraint systems and an initial set of input variables. It is shown that all types of parallelism - AND-OR, task and data - can be extracted from constraint representations. Both computation and logic can be spanned in a single representation. Additionally, constraint programs can be compiled to parallel executables to run on a variety of architectures and the specification of execution environment directives assists in the generation of architecturally optimized code. A prototype compiler has been developed. Results for some examples are presented.

Keywords: Parallel, Constraint, Language

1 Overview

Automatic parallelization of a problem-oriented specification of a computation is an attractive goal for application developers who wish to take advantage of parallel execution environments but do not wish to learn the intricacies of parallel programming. Unfortunately, despite substantial investments in compiler technology, it has not proved possible to implement fully satisfactory automatic parallelization of conventional sequential procedural programming languages. The choice of representation for specification of programs has often strongly effected the type and amount of parallelism which can be extracted from a sequential program. We argue that early specification of procedurality is a major inhibitor to automatic parallelization. Constraint systems with an input set are a representation of computations which contain no explicit control flow. In this paper we show that specification of a computation as a constraint system and the choice of some subset of variables as inputs is a representation from which parallelism of all forms is readily extracted. We then demonstrate that, in at least some cases and perhaps surprisingly, parallel programs which are realized by compilation of constraint systems to procedural programs execute with efficiency competitive with programs in which the parallel structure has been hand-coded.

Examples are given illustrating extraction of AND-OR parallelism and task and data parallelism. The examples also show how the granularity of parallelism can be controlled through the use of extended types in the representation basis for the constraint systems. In addition

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

to the extraction of parallel structuring, constraint systems allow derivation of multiple parallel programs from a single constraint specification by selecting different sets of variables as inputs. We also note that the compilation process can be directed to produce either: (i) a program which is "effective", that is, produces exactly one assignment of values to variables which satisfies the constraint system or (ii) to produce a program which is "complete", that is, a program which generates all assignments of values to variables which satisfy the constraint system.

The compilation algorithm has been described elsewhere [12] and will not be repeated here. However, certain compilation problems such as avoiding excessive use of memory are discussed later. Also, it is noted that the compiler can be given a separate specification of the execution environment which it can use to optimize the implementation of the parallel structure which is derived.

Some of the examples have been compiled and executed and the results of the comparison of execution behavior to hand-coded programs are given. The programs compiled from constraint systems are surprisingly efficient for the few relatively simple examples attempted thus far.

2 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) [17], 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 [3] transforms constraints to a compilable language rather than to an interpretive execution environment as in many constraint systems, but is not concerned with extraction of parallel structures. Vijay Saraswat described a family of concurrent constraint logic programming languages, the cc languages [18]. 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 [18]. The performance reported for the system is only comparable with commercial Prolog and Lisp systems [15]. Parallel logic programming [5, 9, 13] is another area of related work. PCN [5] and Strand [9] 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 [6] 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.

Some reasons why our approach has greater potential for obtaining efficient parallel programs than parallel logic languages, compilation of functional languages to parallel execution and/or concurrent constraint logic programming languages include: (i) A constraint specification system can provide a richer set of primitives than is given in current logic languages. (ii) Functional languages restrict dataflow to be unidirectional whereas constraint systems impose no constraints on dataflow. (iii) Data parallelism is more readily expressed in constraint specifications than in pure functional languages. (iv) Concurrent constraint satisfaction systems currently rely on interpretive methods for evaluation of constraint satisfaction whereas we compile to direct procedural code. (v) Narrowing the target domain and direct use of semantic domain knowledge enable the compiler to choose efficient algorithms for the derived computations.

3 Approach

This section outlines our approach to specifying programs using constraints and gives a flavor of the compilation algorithm. For more detail about the compilation algorithm, refer to [12].

A program in our system 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 constraint specification and the input set of variables. The dependence graph is mapped to the target language, CODE [16], 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 such as the CRAY J90, SPARCcenter 2000, and Sequent and the distributed memory PVM [11] system are generated by CODE. An MPI [10] backend for CODE is under development. The next two subsections describe the hierarchical type system and the constraint representation, respectively.

3.1 Domain Specification: The Hierarchical Type System:

The lowest level of the type system consists of integers, reals, and characters with the usual arithmetic operators on integers and reals. The next level of the type hierarchy contains arrays. The semantic domain for most of our application programs is matrix computation. Therefore, we add semantic specifications to arrays to construct matrix types with their 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 [7]. 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. with their associated operations could be included to broaden the class of programs which can be compactly represented. Note that inclusion of structured types in constraint systems leads directly to component reuse since the constraint translation is at the level of operations of the structured types.

3.2 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 program enumerates the different relationships that must be established or maintained by the executing code. Our system handles linear arithmetic constraints and linear (and with some restrictions, stated in [12], non-linear) constraints on matrices. The usual arithmetic expressions and calls to library and user-defined C functions (defined externally and linked during execution) are allowed in a constraint. In addition, indexed operators of the following form are allowed:
$< op > FOR\ (< index > < low > < high >)\ X$
where $op \in \{+, -, *, /\}$, $index$ is an integer variable, low and $high$ are range bounds for $index$, and X is an arithmetic expression. low and $high$ must be statically bound.

The constraints can be constructed by application of the following rules.
Rule 1: (i) $X_1\ R\ X_2$, is a constraint,
where $R \in \{ <, \leq, >, \geq, ==, \neq \}$, 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) NOT A (ii) A AND/OR B are constraints, where A and B are constraints.
Rule 3: Modularity is provided by *constraint modules*, which are encapsulations of relationships between parameters and can be invoked within a constraint. Calls to user-defined constraint modules are constraints.
Rule 4: Constraints over indexed sets have the form:

OR/AND FOR (<index> <low> <high>) { $A_1, A_2, \ldots, A_n$ }

where *index* is an integer variable , *low* and *high* are range bounds for *index*, and $A_1, A_2, \ldots, A_n$ are constraints. *low* and *high* must be statically bound. This condition will be relaxed in later versions of the compiler.
E.g. for Rule 4: AND FOR (i 1 2) { A[i] == A[i-1], B[i+1] == A[i] } captures the constraints $A[1] == A[0]$ AND $A[2] == A[1]$ AND $B[2] == A[1]$ AND $B[3] == A[2]$

3.3 Translation to a Compilable Language:

Encapsulated within $A + B == C$ are three computations: $A = C - B$, $B = C - A$, $C = A + B$; and a conditional, $A + B == C$. One of the three computations can be extracted depending on which two of the three variables are inputs or *known* variables. If all three are inputs, the constraint can be transformed into a conditional to be checked for satisfiability. If fewer than two variables are inputs, the constraint is unresolved and no resulting program can be extracted. Computations are associated with nodes and conditionals with arcs in the dependence graph. A derived dependence graph can establish the constraints by computing values for some or all of the non-input or output variables. Constraints over indexed sets are compiled to loops. The derivation of dependence graphs is explained in detail in [12]. Our translation exploits the different types of parallelism implicit in the constraints. The compiler generates single-assignment variables. A constraint specification represents a family of dependence graphs. Generation of all possible dependence graphs can result in combinatorial explosion. We construct the dependence graph only for a specified input set.

4 Advantages of using Constraint Specifications

4.1 Ease of Use

The main appeal of constraint programming is the significant reduction of the gap between the informal description of the problem and its computer specification. The emphasis is on the description of the problem than on how it will be solved. The programmer is given a representation for reasoning in high level specifications and does not have to deal with issues of control flow or synchronization. Programming is still required and is critical, but it takes place at a higher level.

4.2 Generation of either Effective or Complete Programs

The presence of the "or" operator in a constraint system results in the possibility that there exists more than one assignment of values to the variables which will result in satisfaction of the constraint system. (A given input set for a program with an "or" operator may or may not allow multiple assignments which satisfy the constraint system.) A program which is effective generates exactly one set of assignments of values to variables which satisfies the constraint system. A program which is complete generates all of the sets of assignments of values to variables which will satisfy the constraint system. The compilation process can be told to generate the executable either for exactly one "or branch" of the dependence graph or to generate the executable for all paths which lead to valid assignments. Thus, the compilation process can produce programs which are either effective or complete. A program which is complete realizes "or" parallelism, as will be further discussed in Section 5.1.

4.3 Extraction of Alternative Programs from Different Initial Conditions

Part of the appeal of a constraint programming language is its multi-directional nature - the facility to extract values for different sets of variables depending on the composition of the input set. While this is a secondary aspect for parallel programming concerns as compared

to the ease of use aspect, this is still important in many application domains. We illustrate the extraction of two programs from a single constraint specification in the next subsection.

4.3.1 Quadratic Equation Solver

Consider a formulation for the real roots of a quadratic equation, $ax^2 + bx + c == 0$, in Figure 1. The *Main* program variables are $\{ a, b, c, r1, r2 \}$. The constraint module *DefinedRoots* has the parameters $\{ a, b, c, r1, r2 \}$. t is a local variable in *DefinedRoots*. *sqr, sqrt*, and *abs* are library functions corresponding to the arithmetic functions of square, square root and absolute. The inverses to these functions are defined. *"U"* denotes an undefined value.

```
/* 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)
```

Figure 1: Constraint Specification for Quadratic Equation Solver

The input set to the program could be chosen as $\{ a, b, c \}$. The extracted dependence graphs for *Main* and the constraint module *DefinedRoots* for this input set are shown in Figure 2. The conditionals on the arcs have to be satisfied for destination nodes to execute. The node,*DefinedRoots*, invokes the corresponding dependence graph. Assignments at a node are computations associated with the node.

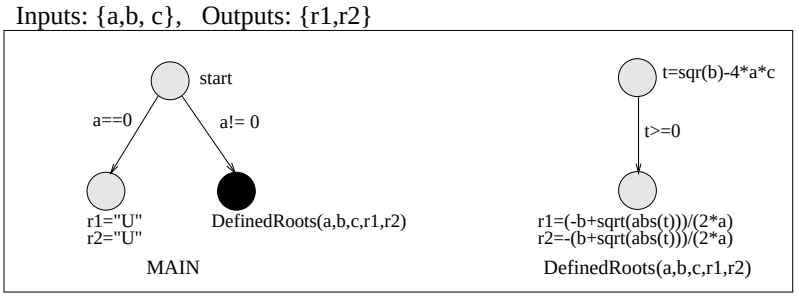

Figure 2: Dependence Graphs for Quadratic Equation Solver Example with inputs a,b,c

Alternatively, the input set to the program could be chosen as {a, b, r1 }. The extracted dependence graphs for *Main* and the constraint module *DefinedRoots* for this input set are shown in Figure 3. The dependence graphs compute values for variables c and $r2$. The inverses of the functions *sqrt* and *abs* have been applied to derive the computations for t. The compiler can be optimized to detect that the path starting from the node computing $t = -sqr(2 * a * r1 + b)$ can never be traversed to completion. Also, the constraint specification in Figure 1 can be enhanced to specify relationships containing the components of imaginary roots ($t < 0$). The granularity is too fine-grained in this example, which has been chosen just for illustrative purposes.

Figures 2 and 3 show that the same constraint program specification can be reused to derive the dependence graphs for different input sets. However, not all input sets can lead to dataflow graphs. For example, no dependence graph can be generated with the input set $\{ a, r1 \}$.

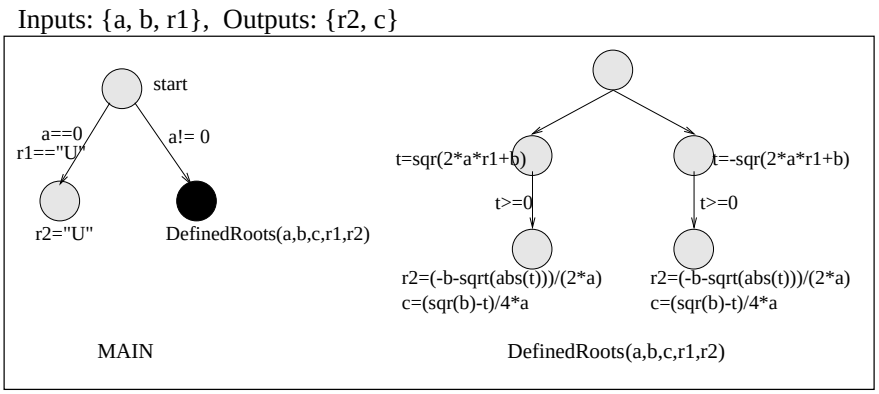

Figure 3: Dependence Graphs for Quadratic Equation Solver Example with inputs a,b,r1

5 Extraction of Parallelism

This section details the different issues in the extraction of parallelism from constraint specifications. This includes the exploitation of AND-OR parallelism in the specification, extraction of control and data parallelism, and the factors governing granularity of operations.

5.1 AND-OR Parallelism

Our translation process collects constraints connected by AND operators at the same node in the dependence graph [12]. Some of these constraints may be classified as computations depending on the input set. These computations have the potential for parallelism. Since the compiler generates a single-assignment system, the lone write to a variable occurs before any reads to it. This helps in determining data dependence and independent computations can be assigned to several nodes for parallel execution. The parallelism extracted from these computations will be referred to as AND-parallelism. In the dependence graph for *DefinedRoots* in Figure 2 the computations for $r1$ and $r2$ can be done in parallel. Similarly the computations for $r2$ and c in the dependence graph for *DefinedRoots* in Figure 2 can be done in parallel. Both of these are instances of AND-parallelism.

Constraints connected by OR operators are collected on diverging paths in the dependence graph [12]. Some of these constraints get classified as computations and some as conditionals depending on the input set. The divergent computational paths that arise as a result of constraints connected by OR operators have the potential to be executed in parallel and give rise to OR-parallelism.

We illustrate AND-OR parallelism through a simple example. Consider the constraint specification in Figure 4 for a program involving variables { a,b,c,x,y }. The dependence graph for the input set { a,b,c } and output set { x,y } is shown in Figure 5. Since a, b, c are inputs, $a < b$ and $a < c$ are classified as conditionals. The constraints involving equalities ($b == x$, $y == c$, $x == c$, and $b == y$) are classified as computations for the single unknown in them. OR parallelism comes into play in the parallel execution of the two paths branching out from the start node in the event that $a < b$ and $a < c$. This also implies that this program can be compiled to be either complete or effective, as discussed in Section 4.2. AND parallelism is extracted from the computations for x and y.

```
a<b AND b==x AND y==c
OR
a<c AND x==c AND b==y
```

Figure 4: Constraint Specification for a Simple Example

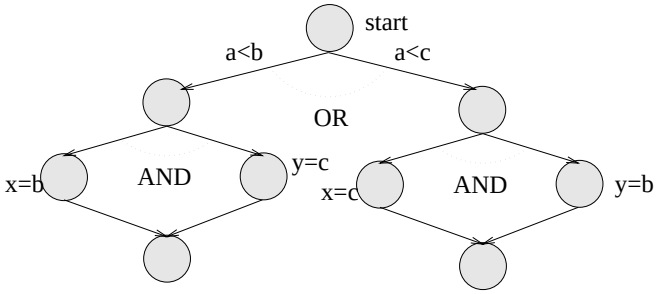

Figure 5: Dataflow Graph showing AND-OR Parallelism

5.2 Control and Data Parallelism

Both control and data parallelism are extracted by our translation process. Control parallelism arises from nodes being assigned computations derived from the translation process. Data parallelism arises from the hierarchical representation of our type system. For example, matrices can be represented as blocks of sub-matrices and constraints over sub-matrices are translated to data-parallel conditionals/computations. We illustrate the extraction of control and data parallelism through an example in the next subsection.

5.2.1 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 $k \geq 2$. It is assumed that the vectors x and d are likewise partitioned. It is further assumed that the blocks B and C are symmetric and commute. A version of the parallel algorithm (taken from [14]) programmed in our system is shown in Figure 6. It is not necessary to understand all the details in the algorithm to follow the translation to a dependence graph. There are three phases in the system: (i) Reduction, which involves an *AND FOR* over three constraints, the last of which is another *AND FOR*, (ii) Single-Solution, which involves the solution of an $Ax = b$ type equation, (iii) Back-Substitution, which has two nested *AND FOR*s.

The resulting dataflow graph is shown in Figure 7. The nodes annotated by *BP, CP, dP*, and x compute the value of the corresponding variable. A *FOR* node initiates the different iterations of a loop. The two such nodes in the figure correspond to the two outer *AND FOR*s for index j in the program. The annotation "Replicated" on the arcs specify that the annotated arc and the destination node (shaded) are dynamically replicated for parallel execution. The two such annotated arcs correspond to the two nested *AND FOR*s (for index i) in the program. The form of parallelism extracted here is data. Different parts of the matrices, dP and x are computed in parallel in the two replications, respectively. Control parallelism occurs in the parallel execution of the nodes computing BP, CP, and dP, respectively. The nodes annotated by "Merge" collect computed results from parallel executions. Note 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 control parallelism within the expression $2 * CP[j-1] * CP[j-1] - BP[j-1] * BP[j-1]$ in the computation for $BP[j]$.

```
/* REDUCTION PHASE */
AND FOR (j 1 k-1) {
      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 0 pow(2,k-j)-2) {
            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

/* SINGLE-SOLUTION STEP */
BP[k-1] * x[pow(2,k-1)] == dP[pow(2,(k-1))][k-1] AND

/* BACK-SUBSTITUTION PHASE */
AND FOR (j k-1 1) {
      AND FOR (i 0 pow(2,k-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)] ) }}
```

Figure 6: Specification of the BOER Algorithm as a Constraint System

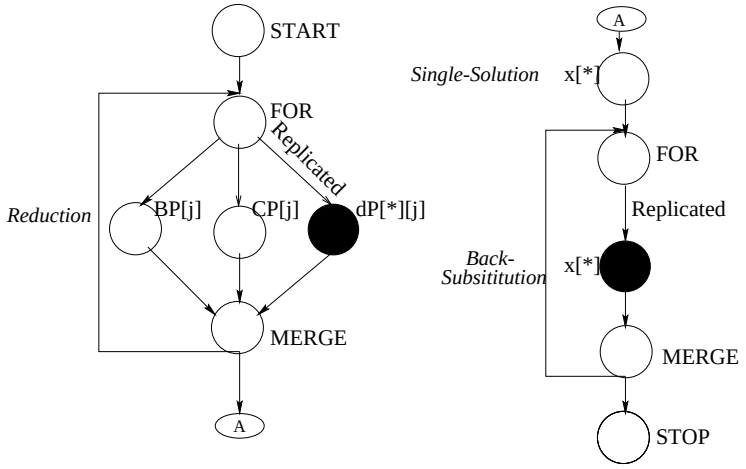

Figure 7: Dataflow Graph for BOER Program

5.3 Control over Granularity

The granularity of the derived dependence graphs depends upon the types represented as primitives in the constraint representation. The introduction of structured types and operations on structured types as primitives in the constraint representation gives natural units of computation at a granularity appropriate for task level parallelism Granularity is further controlled by the following features in our system: (i) modularity for reusable modules, and (ii) definition of sequential functions. We illustrate control over granularity through structured types by means of an example in the next subsection.

5.3.1 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 [8] involves dividing the matrix into blocks. A constraint program for a problem instance split into 4 blocks is shown in Figure 8. $S_0 \ldots S_3$ represent lower triangular sub-matrices that are solved sequentially, and $M_{10}, M_{20}, \ldots M_{32}$ represent sub-matrices that must be multiplied by the vector from above and the result subtracted from the vector from the left. The vector multiplications for all Ms within a column may be done in parallel. This parallelism yields an asymptotic ideal speedup of $P^2/(3P-2)$, where P is the number of processors. 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 operator and an indexed set of constraints, 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. If the specification is given in terms of indexed sets of constraints over sub-blocks of the matrix, then the granularity of the computation can be controlled by specification of the size of the sub-blocks.

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

Figure 8: Specification of the BTS Algorithm as a Constraint System

6 Execution Environment Specification

So far we have demonstrated that constraint programs can be compiled to parallel programs of appropriate granularity. However, to achieve high performance it is vital to make use of architecture-specific mechanisms when generating executables for different architectures. We plan to do this through a specification, separate from the constraint specification, for execution environment directives. In this specification, features for the following will be included:

(i) The Architecture Model: For a shared memory system there can be dramatic improvement in performance if certain variables are declared as shared. This avoids copying of large data arrays. The compiler automatically implements synchronization and locking mechanisms for consistency of data. However, data should not be shared across computational paths arising from OR connectives as this could lead to incorrect results and would not deliver multiple solutions, if any. Data could be shared across AND computations.

For distributed memory and distributed shared memory, compiler directives could be given to preserve data locality in computation.

(ii) Communication Properties: The programmer could give specifications for latency or bandwith.

(iii) Granularity Specification: The programmer could specify estimates of the amount of computational work associated with each unit of computation and each unit of communication, which (s)he thinks will yield effective computation. This can be used by the compiler to determine appropriate granularity for the execution of computations.

7 Performance Results

In this section we present the performance results for some of the examples introduced earlier in the paper. These results present initial evidence that constraint specifications can be compiled to competitively efficient executables.

7.1 Block Odd-Even Reduction Algorithm

The authors in [14] 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 because the computation for $BP[j]$ and $CP[j]$ involves 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 9 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.

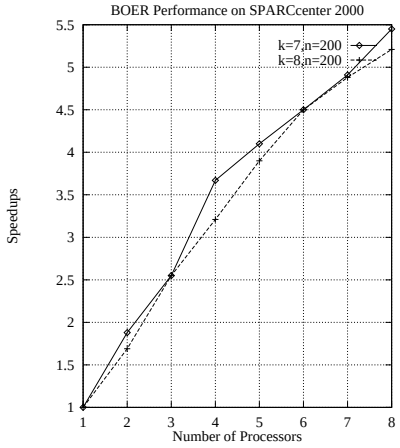

Figure 9: Performance Results for BOER Program

7.2 Block Triangular Solver

Figure 10(a) gives the speedups for a 1200 $\times$ 1200 matrix on a shared memory Sequent machine. It is seen that the performance of the code generated from the constraint system is comparable to the hand-coded program's performance. The difference in speedup 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 10(b) gives the speedups for a 8800 $\times$ 8800 matrix (number of blocks=11, block size=800) on an 8-processor SPARCcenter 2000. 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 asymptotic ideal, until about 7 processors, after which it drops below the asymptotic ideal. The hand-coded parallel program for the Sparc is under development.

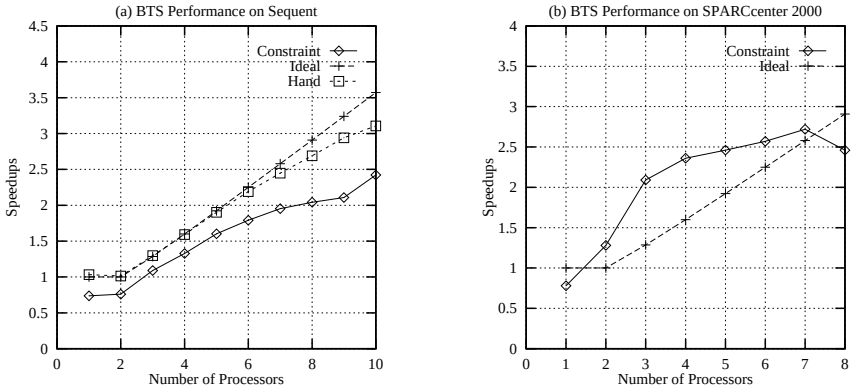

Figure 10: Performance Results for BTS on: (a) Sequent (b) SPARCcenter 2000

8 "NEW" Construct for Iterative Solutions

The programs compiled from constraint systems may pose a heavy memory requirement for iterative programs, which occur frequently in parallel numerical algorithms. Direct compilation uses a new memory location for every iterative update to a single-assignment variable. To overcome this problem we propose a construct called *new*, very similar to the corresponding complementary construct *old* in the functional language Sisal [4]. *new* <*variable*> denotes a new instantiation of the variable and thus simulates a destructive update. We plan to enhance the compiler with the capability of detecting instances which require the use of *new*. We foresee this occurring frequently in indexed sets of constraints with an index that does not occur in the body. For example, consider the constraint specification in Figure 11 for the Laplace Equation for an $N \times N$ board using a 4-point stencil.

```
AND FOR (t 1 100) {
    AND FOR (i 2 N-1) {
        AND FOR (j 2 N-1) {
            4*x[i][j] == x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1]
        }}}
```

Figure 11: Constraint Specification for the Laplace Equation

The variable t does not occur in the body. This is interpreted by the compiler as an indication of an iterative solution. If the input to the problem are the boundary values, the compiler determines that the first term, namely $x[i][j]$, is the only term which remains as unknown over all iterations. Consequently, it chooses this term as the candidate for *new*. It also recognizes that initial values need to be chosen for all points excluding boundary points. By default, these are chosen to be 0. The programmer is then presented with an enhanced version of the code shown in Figure 12, where an *INIT* construct for initializing the required section of the array has been introduced and the term $x[i][j]$ has been enhanced to *new* $x[i][j]$. The programmer is allowed to change any of this. Finally, the term enhanced with *new* is chosen to be the computed term in all iterations of the loop. Each iteration overwrites the current value of $x[i][j]$.

```
INIT {for (i 2 N-1) { for (j 2 N-1) { x[i][j] = 0 } } }
AND FOR (t 1 100) {
    AND FOR (i 2 N-1) {
        AND FOR (j 2 N-1) {
            4 * (new x[i][j]) == x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1]
        }}}
```

Figure 12: Compiler-enhanced version of the Laplace Equation

If no unique term is unknown in all iterations of a loop, any arbitrary term is chosen as the candidate for *new*. By making the compiler interactive we hope to overcome most of the problems that may in encountered in the implementation of indexed sets of constraints as loops.

9 Summary and Future Research

In conclusion, we claim that constraint systems offer a representation of computations, from which all types of parallelism can be readily extracted. Constraint systems with appropriate initialization specifications can be mapped to a generalized dependence graph. Coarse-grain

parallelism can be extracted through modularity, operations over structured types, and specification of arithmetic functions. Constraint systems offer other advantages such as extraction of multiple programs from a single representation and the generation of either complete or effective programs. 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 progress to date in automatic extraction of parallelism from computations expressed as constraint systems. We plan to continue the work with a focus on matrix computations. 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 [2]. 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 by Collins and Browne [7]. The hierarchical type model for matrices establishes a compilable semantics for computations over hierarchical matrices. Additionally, further steps in this research are to implement completely the execution environment specification, to define the semantics of recursion in constraint module calls, and to relax the requirement that all of the bounds in indexed sets of constraints be statically found.

References

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

[2] Utpal Banerjee. *Loop Parallelization*, A Book Series on Loop Transformations for Restructuring Compilers, Kluwer, 1994.

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

[4] D.C. Cann. The optimizing Sisal Compiler: Version 12.0. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April 1992.

[5] K.M. Chandy, S. Taylor. *An Introduction to Parallel Programming*, Jones and Bartlett, 1992.

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

[7] T.S. Collins, J.C. Browne. MaTrix++; An Object-Oriented Environment for Parallel High-Performance Matrix Computations. *Proc. of the 1995 Hawaii Intl. Conf. on Systems and Software.*

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

[9] I. Foster, S. Taylor. *Strand: New Concepts in Parallel Programming* Prentice Hall, 1990.

[10] I. Foster. Designing and Building Parallel Programs. Addison-Wesley, 1995.

[11] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. PVM: Parallel Virtual Machine:A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994.

[12] Ajita John, J. C. Browne. Compilation of Constraint Systems to Procedural Parallel Programs. *Ninth Workshop on Languages and Compilers for Parallel Computers*, California, August, 1996.

[13] M. V. Hermenegildo. *An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel.* PhD thesis, University of Texas at Austin, 1986.

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

[15] M. Mehl, R. Scheidhauer, C. Schulte. An Abstract Machine for Oz. *Programming Languages, Implementations, Logics and Programs, Seventh Intl. Symp.*, Springer-Verlag, LNCS 982, The Netherlands, September 1995.

[16] P. Newton, J. C. Browne. The Code 2.0 Graphical Parallel Programming Environment. *Proc. of the 1992 Intl. Conf. on Supercomputing.* Washington, DC, July 1992.

[17] H. Richardson. HPF: History, Overview and Current developments. Thinking Machines Corporation, TMC 261, URL: http://www.crpc.rice.edu/HPFF/publications.html

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