Compilation to Parallel Programs from Constraints

Ajita John, J.C. Browne
Dept. of Computer Sciences
University of Texas at Austin
Austin, TX 78712

E-mail: {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 computation 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 language CODE, which represents parallel computation structures as generalized dependence graphs. Finally, parallel C programs are generated. To extract parallel programs of appropriate granularity, the following features are included: (i) modularity, (ii) operations over structured types as primitives, (iii) sequential C functions. A prototype of the compiler has been implemented. The domain of matrix computations is targeted for applications. Initial results are very encouraging.*

1. Introduction

Representing a 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 [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 for derivation of control structure. Constraint systems offer some unique advantages as a representation from which parallel programs are to be extracted. All types of parallelism (AND, OR, task, data) can be derived [11]. Either effective or complete programs can be derived from constraint systems on demand [11].

The focus of this research is to derive from constraint representations, parallel programs of execution efficiency competitive with procedural languages. The next two sections outline our approach and related work. This is followed by a description of the programming system and the compilation algorithm. We conclude the paper with performance results for one example program (and pointers to other examples) and directions for future work.

2. Approach

A constraint 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 constraint program. The dependence graph is mapped to the target language CODE [14], which expresses parallel structure over sequential units of computation declaratively as a generalized dependence graph. Finally, sequential and parallel C programs for shared memory machines like the CRAY J90, SparcCenter 2000, and Sequent and the distributed memory PVM [8] system can be generated. An MPI [5] backend is under development. The granularity of the derived dependence graphs depends upon the types directly represented as primitives in the constraint representation. Introduction of types and

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

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 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) narrowing of the semantic domain (iii) definition of sequential functions, and (iv) a rich type set. The main features of our approach are detailed in the rest of the section.

(i) 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 specification enumerates the different relationships that must be established or maintained by the executing code.

(ii) 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 for finding the real roots of a quadratic equation, $ax^2 + bx + c == 0$, which uses a module. $''U''$ 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)
```

Figure 1. Real roots of a quadratic equation

(iii) Compilation to a procedural language: Encapsulated within $A + B == C$ are three assignments: $A = C - B$, $B = C - A$, $C = A + B$ and a conditional: $A + B == C$. One of the three assignments can be extracted depending on which two of the three variables are inputs or *known* variables. If all three are inputs, the constraint can be transformed into a conditional to be checked for satisfaction. If less than two variables are inputs, the constraint is unresolved and no resulting program can be extracted. A derived dependence graph can establish the constraints by extracting computations to resolve some or all of the non-input (output) variables (see Section 5). Our translation exploits the parallelism in the constraints. The resulting program has single-assignment variables and can generate multiple solutions on alternate paths of the dependence graph.

(iv) 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. The matrix subtypes currently implemented are lower and upper triangular and dense matrices. We plan to extend to more specialized 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.

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

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 [7] 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 program-

ming 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, 9] 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.

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.

4. Language description

4.1. Types

The lowest level of the type system consists of integers, reals, characters, and arrays with the operators of addition, subtraction, multiplication, and 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. Other than dense matrices the system 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

Linear arithmetic expressions (including function calls) are allowed in a constraint. Only a restricted form of compilation (see Section 5.3) is done with non-linear expressions and functions whose inverses are not defined. This will be relaxed as we target more applications. *Indexed operators* of the following form are valid expressions: $< op > FOR\ (< index > < b1 > < b2 >)\ X$ $op \in$ {+, -, *, /}, *index* is an integer variable, $b1$ and $b2$ are range bounds for *index*, and X is an arithmetic expression. E.g. + FOR (i 1 5) $A[i]$ specifies the sum of the elements in array A from positions 1-5. *b1* and *b2* have to be statically bound.

The following (underlined) constructs are constraints:

Rule 1: (i) $X_1\ R\ X_2$, where $R \in \{ <, \leq, >, \geq, ==, \neq \}$, X_1, X_2 are arithmetic expressions

(ii) $M_1 == M_2$, where M_1, M_2 are expressions involving matrices and matrix operators

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

Rule 3: Calls to user-defined constraint modules

Rule 4: Constraints over indexed sets:

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

where *index* is an integer variable , $b1$ and $b2$ are range bounds for *index*, and { $A_1, A_2, \ldots, A_n$ } is a set (unordered) of constraints. *b1* and *b2* have to be statically bound. This will be relaxed in later versions of the compiler. E.g. 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 $B[3] == A[2]$

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

4.3. Programs

A program consists of: (i) Global variable declarations, Global Input variables. (ii) User-defined function signatures: signatures of C functions (linked during execution) (iii) Constraint Module definitions: module name, formal parameters and their types, local variable declarations, and a body, which is similar in syntax to the main body. (iv) Main body of the program: constraints constructed from Rules 1-4.

5. Compilation

The four phases of the compilation algorithm are described in this section. See [10] for more detail.

5.1. Phase 1

The textual source program is transformed into a source graph. 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 constraint attached to it. For each application of Rule 2, the graph is expanded as in Figures 2(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 2(c).

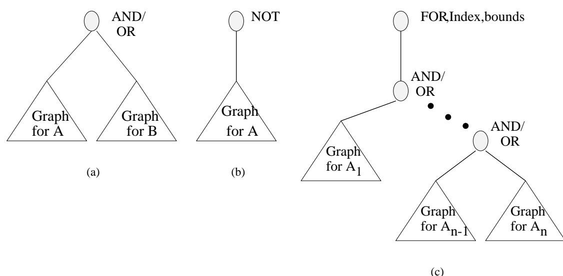

Figure 2. (a),(b): Rule 2 (c): Rule 4

5.2. Phase 2

A depth-first traversal of the graph, obtained from phase 1, constructs a tree. This construction simplifies the constraint specification as illustrated in Figure 3, where a, b, c, and d are simple constraints. Nodes are collapsed 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. The satisfaction of all the constraints (implicitly connected by AND operators) along a path from the root to a leaf in the resulting tree represents a satisfaction of the constraint system. Different paths (arising from OR operators) in the tree represent alternate ways of satisfying the constraint system giving rise to the possibility of multiple solutions. The detailed algorithm can be found in [10].

5.3. Phase 3

A dependence graph generation for the constraint and input set specification is attempted in this phase.

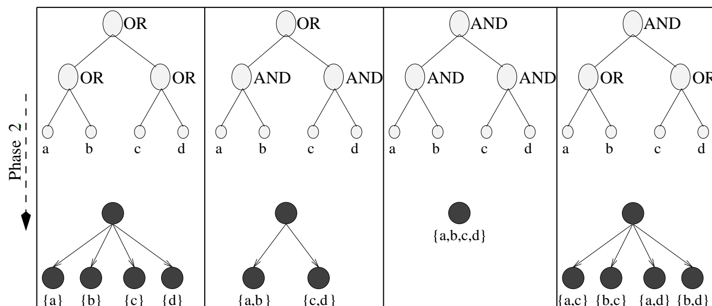

Figure 3. Phase 2

In the generated dependence graph, nodes are computational elements and arcs between nodes express data dependency. A node has the form: firing rules, computation, routing rules (see Figure 4). A path from the root of the graph to a leaf is a possible computation path. To generate a dependence graph, a depth-first traversal of the tree from phase 2 is done. The traversal starts with the information that variables in the input set (known set) are known and tries to generate paths that contain computations for variables in the output set. Each node in the tree has an attached set of simple constraints. 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 execute. To be so classified, a constraint must have no unknowns (determined from the composition of the known set) when the node is visited.
(ii) Computation: To fall into this category, a constraint must involve an equality and have a single unknown, which is added to the known set and retained in it for the subtree rooted at the current node. Thus, the compiler generates single-assignment variables.
(iii) Routing Rule: a condition that must hold for this node to send out data on its outgoing paths. To be a routing rule, all unknown variables in the constraint must have been resolved by the computations at this node.

When a constraint is classified as a computation, it is mapped to an equation. Depending on the types of the variables, specialized routines are invoked to solve for the unknown. Any constraint not falling into one of the 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

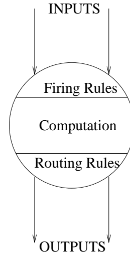

Figure 4. Dependence graph node

tree are abandoned the user is informed of the restrictiveness of the initial input set. Constraints involving inequalities must be resolved as firing/routing rules. Non-linear constraints must have the non-linear terms involving only known variables. This will be relaxed later. Details of phase 3 for constraint module calls can be found in [10].

5.3.1. 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 $x = b + 2$ can be done in parallel. Hence, 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. Evidently, the granularity of such a scheme depends on the complexity of the functions invoked within 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 $x = m * y + m * z$ and 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. 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.

Parallelism in AND indexed sets: Our system design handles indexed sets with specified patterns of access. To classify a constraint in an indexed set as a conditional or computation, it is evaluated for all index values. The restrictions on the indexed set structure to be compiled successfully in our system are: For all values of the index (i) a constraint has to have the same classification. (ii) if a constraint is classified as computation, the same unique general term in the constraint has to be the unknown.

Indexed sets are compiled to loops. To extract parallelism, the computations within the compiled loop structure are studied. 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.

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

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

5.4. Phase 4

The CODE [14] programming interface is drawing and annotating of a directed (dependence) 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 this AST, which is processed in the CODE system. The final output is an executable in the form of a sequential/parallel C program.

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, SparcCenter 2000, Sequent Symmetry machine and the PVM system. Due to space limitations, we have chosen to include one representative example in this paper. This is described in the next subsection. For other examples including the Block Triangular Solver, please see [10].

6.1. 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 $k \geq 2$. Thus, $N = Mn$ denotes the order of A. It is assumed that the vectors x and d are likewise partitioned, that is, $x = (x_1, x_2, \ldots, x_M)^t$, $d = (d_1, d_2, \ldots, d_M)^t$, $x_i = (x_{i1}, x_{i2}, \ldots, x_{in})^t$, and $d_i = (d_{i1}, d_{i2}, \ldots, d_{in})^t$, for $i = 1, 2, \ldots, M$. It is further assumed that the blocks B and C are symmetric and commute.

A version of the parallel algorithm (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 5. 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 5 are the terms 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)] }}
```

Figure 5. Constraint specification for BOER

The resulting dataflow graph is shown in Figure 6 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 program. The annotation "Replicated" on the arcs specify that the annotated arc and the destination node (shaded in Figure 6) are dynamically replicated for parallel execution. The two such annotated arcs correspond to the two nested indexed sets (for index i) in the program. 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 computations 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 7 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. The sequential version for k=7 and k=8 took 50.1 seconds and 91.9 seconds, respectively. Note the attainment of near-linear speedup for this (relatively small) number of processors.

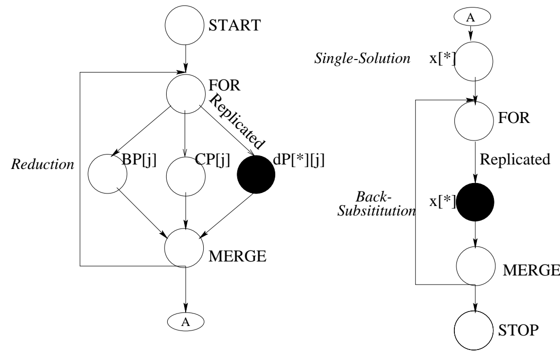

Figure 6. Dataflow graph for BOER

7. Summary and future research

In conclusion, we claim that constraint programs offer a rich, relatively untapped representation of computation from which parallelism can be readily extracted.

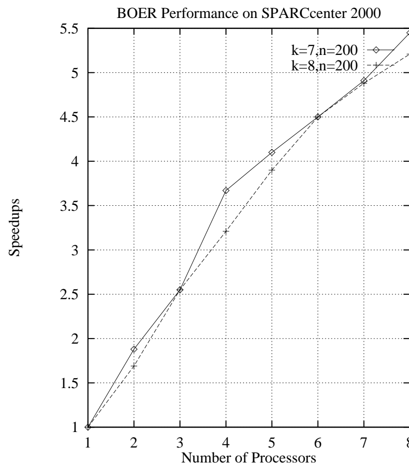

Figure 7. Performance results for BOER

Constraint systems with appropriate input sets 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. By giving the programmer control over compilation choices for the execution environment, we can 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 by Collins and Browne [4]. The hierarchical type model for matrices establishes a compilable semantics for computations over hierarchical matrices.

Additionally, the next steps in this research are: 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) Relax the restrictions on the AND indexed set construct. c) Implement completely the execution environment specification.

References

[1] D. Baldwin. Consul: A parallel constraint language. *IEEE Software*, 1989.

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

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

[4] T. Collins and J. 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] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

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

[7] B. N. Freeman-Benson. A module compiler for Thinglab II. In *Proc. 1989 ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1989.

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

[9] 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.

[10] A. John and J. Browne. Compilation of constraint systems to procedural parallel programs. In *Workshop on Languages and Compilers for Parallel Computers, LNCS*. Springer-Verlag, 1996, To Appear.

[11] A. John and J. Browne. Extraction of parallelism from constraint specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1501-1512, August 1996.

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

[13] M. Mehl, R. Scheidhauer, and C. 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, July 1992.

[15] H. Richardson. High Performance Fortran: History, overview and current developments. Technical Report TMC 261, Thinking Machines Corporation.

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