Compilation of Constraint Systems to Parallel Procedural Programs

by

Ajita John, B.Sc., B.E., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1997

Compilation of Constraint Systems to Parallel Procedural Programs

Publication No. _____

Ajita John, Ph.D.

The University of Texas at Austin, 1997

Supervisor: J. C. Browne

An attractive approach to specifying programs is to represent a computation as a set of constraints upon the state variables that define the solution and to choose an appropriate subset of the state variables as the input set. But, there has been little success in attaining efficient execution of parallel programs derived from constraint representations. 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 a control flow-free and dataflow-free specification of a computation, thereby offering the compiler freedom of choice in deriving control structures. All types of parallelism (AND, OR, task, data) 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 through different choices of the input set of variables.

This dissertation reports 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. 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 atomic functions.

A prototype 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. Performance results for example programs are very encouraging. The feasibility of extracting efficient and portable parallel programs from domain-specific constraint systems has been established.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Statement and Approach

The last decade has seen a rapid development in parallel hardware technology. Apart from supercomputers, parallelism has pervaded workstations, personal computers, and networks. Multiple processors inside a computer and across a network can be targeted by an application program for performance. Both scientific and commercial applications drive the need for exploiting parallelism in programs. Despite the advancement in parallel hardware technology, development in parallel software environments has lagged far behind. But, interest in parallel programming has sparked enthusiasm for alternative representations for expressing computations.

An ideal representation (parallel programming language) is one that would easily be applied to many problem domains and would be compilable for efficient execution in a variety of execution environments. A great deal of effort has gone into attempts to compile efficient parallel programs directly from existing sequential languages [EB91, HKT91]. Many extensions to add communication and synchronization to existing sequential languages have been proposed [CK92, And91]. New languages of many types have been proposed [CM89, And91]. But, there does not

yet appear to be any widely accepted approach to parallel programming.

In this research we suggest that constraint languages can potentially meet many of the requirements for a broadly useful representation for parallel programs. This dissertation defines and describes a constraint language for representing matrix-based numerical computations for parallel execution across a variety of architectures. This dissertation reports on the design and implementation of a compiler which produces efficient parallel procedural programs from the constraint system representations of computations. Finally, this dissertation reports successful parallel execution of non-trivial matrix computations expressed in the constraint specification language.

Our constraint specification language consists of a type system and a set of operators over the type system. A specification for a computation (a program) consists of a constraint system specified in the language, an initialization (an input set consisting of a subset of the names which appear in the constraint system), and a separate specification of the target execution environment. The type system includes hierarchical matrices as primitive types. (A hierarchical matrix is one whose elements may be matrices.) Before reading the next section, readers unfamiliar with constraint systems may wish to read Chapter 2, which defines and describes constraint systems in general and the constraint specification language for matrix computations defined in this project.

1.2 Constraint Systems as Representations of Computations for Parallel Execution

The design and evaluation of representations for parallel programs should be based on a requirements specification. Since it would be difficult to obtain consensus on the "requirements" for a parallel programming language we take the weaker posture

of posing a list of desirable properties for parallel programming languages. There follows a list of desirable properties for a parallel programming system together with an evaluation of constraint systems with respect to each property. The list is subjective and reflects our vision of parallel programming. We assume, for example, that most parallel programs will be written by discipline-area experts interested in solving problems in their discipline area. Other considerations may be important to different interest groups. For example, it will be important to programmers with large libraries of FORTRAN programs that they be able to use their existing programs in parallel execution environments.

Property 1.1 Naturalness of Expression

The representation should be natural to the application domain and should not require the scientist or engineer to reason in representations from other disciplines.

This property is desirable for all representations of computations regardless of the execution environment which is targeted.

Constraints are declarative relationships among entities in the application domain. Constraint specifications require no knowledge of programming. Constraint systems thus have the "Naturalness of Expression" property.

Property 1.2 Full Parallelization

The representation should not impede realization in the executable program of any of the parallelism, which is implicit in the computation, on any reasonable parallel execution environment and should impose no intrinsic barrier to scaling of the program to apply to arbitrarily large computations.

Constraint specifications do not specify control flow. The only restriction on the parallel computation structure which is derived by compilation are those implicit in the granularity of the typed entities over which the constraints are expressed. Therefore, constraint specifications have the "Full Parallelization" property.

Property 1.3 Specification of Execution Properties

The representation should allow the user to express desirable properties for the executable program in application terms. For instance, the representation should enable control over the granularity of operations in application terms.

The granularity of the entities in a constraint specification, matrix computations in our example domain, are readily parameterized. Therefore constraint systems have the "Specification of Execution Properties" property.

Property 1.4 Reuse of Components

The representation should enable easy use of commonly available components and libraries.

Constraint specifications actually require the use of components implementing operations over structured types since they do not specify the procedural algorithms for operations on structured types. The compiler must select an already existing implementation of the operation over the structured type. The domain chosen for this research, matrix computations, has many well-known libraries of components which can be incorporated into the compiled program by the compilation process. Furthermore, constraint specifications do not restrict the algorithms which are used to implement the elementary operations over the structured data types in the representation so that the compilation process is free to chose from among the available libraries. Thus, constraint specification possess the "Reuse of Components" property

Property 1.5 Adaptation of Program to Execution Environment*: The representation should allow the compiler to select algorithms and implementations which are appropriate for a given component of the computation on a given architecture.*

Constraint specifications do not impose any particular technique for the implementation of the operations in the system. This gives the compiler the freedom

to choose algorithms and implementations that are suitable for a particular architecture.

Property 1.6 Portability with Efficiency

The representation should not include assumptions concerning the execution environment so that the program can be compiled to execute with comparable efficiency across a spectrum of parallel execution environments.

Constraint specifications do not specify mechanisms for synchronization or communication so that the compilation process can choose mechanisms and implementations of synchronization and communication which are efficient on the target parallel execution environment without restriction. Properties of the execution environment are explicitly separated from the representation of the computation. Therefore constraint systems possess the "Portability with Efficiency" property.

We will revisit this evaluation at the end of Chapter 2 after giving an introduction to constraints. On the basis of the preceding analysis we believe that constraint systems are a very promising representation of computations where parallel execution is to be targeted. But, realizing this promise depends on implementing a compilation process which utilizes the opportunities offered by constraint specification representations. The development of this compilation process is a major conceptual and implementation challenge. This dissertation is the first attempt to meet this challenge.

Chapter 2

Background: Constraints

A program in an imperative programming language such as C++ or C is a step-by-step procedure to solve a problem. In contrast, programming using constraints is a declarative task requiring only specification of the desired relationships among the entities of the problem.

A constraint specifies a relationship between a set of variables. For example, $C == (F - 32) \times 5/9$ is a constraint relating temperatures in Centigrade and Fahrenheit. Note that the "==" denotes equality as opposed to assignment. A constraint specification enumerates the relationships that must be established or maintained by some constraint satisfaction mechanism if a solution to a set of variables is to be found. The constraint translation mechanism, which transforms the constraint specification to a program, determines the specific method used to satisfy the constraints.

In the von Neumann memory model the state of a computing system is specified by a store which is a vector V of n variables and a valuation assigning a value to each variable in V [Dij76]. An n-dimensional state space for the system is the product of all possible values for the variables in V, and a store is a point in this n-dimensional space. In imperative languages a specified algorithm takes the

state of the system from one store (satisfying some pre-condition) to another store (satisfying a post-condition). In contrast, the store in a constraint system is defined as a constraint which defines a set of points or valuations in the n-dimensional space for that system [Sar89]. *Evaluation* of a constraint system leads to determination of the specific points in the state space where all the constraints in the system are satisfied. A new constraint can be added to a constraint system if the resultant store is *consistent*, that is, it permits at least one valuation [Sar89]. The resultant store is defined by the intersection of the sets of valuations corresponding to the old store and the added constraint. Hence, it is not possible to change the value of any variable in a constraint system. However, it is possible to refine the set of values a variable can assume by pruning some earlier allowed values.

Another aspect of a constraint system is its ability to encapsulate different imperative programs in a single representation. The constraint $C == (F - 32) \times 5/9$ constitutes a constraint specification for a temperature conversion problem. Encapsulated within the constraint are two different imperative programs: $C = (F-32)\times 5/9$ and $F = 32+(9/5)\times C$. (Note the use of the assignment operator "=" instead of the equality operator "==".) Given F or C, the other can be computed by extracting the appropriate program. Constraint programming has been attractive in many application areas such as user interfaces [San94], modeling, and design [Med95, MM89, Ste93] due to its ability to encapsulate many different problems (view any of the variables as unknown) within a single constraint specification.

Logic languages [CM84] are specific instances of constraint languages in that the constraints in logic languages are expressed using predicate calculus. In general, the constraints in constraint languages are expressed in a logic defined for the application area, not necessarily predicate calculus.

2.1 Definitions

We now present some basic definitions upon which we will base the definition of our constraint specification language.

Definition 2.1 *A type system is a triple* $< \zeta, \nu, \theta >$, *where* ζ *is the set of* m *entities (types) in the system,* ν *is a set* $\{ \nu_i | 1 \leq i \leq m \}$ *with* ν_i *being the set (possibly infinite) of values the* i*th entity can assume, and* θ *is a set* $\{ \theta_i | 1 \leq i \leq m \}$ *with* θ_i *being the set of operators defined on instances of the* i*th entity. An element* Op *of* θ_i *is a mapping* $Op : \zeta \rightarrow \zeta$ *(In addition to defining the evaluation of applying an operator, the type of the evaluated value is also defined.).*

Definition 2.2 *A variable is an instance of a particular type.*

Definition 2.3 *A constant is a literal value.*

Definition 2.4 *An expression has an associated type and can be a variable, a constant, a function invocation, or an application of an operator on expressions involving a set of variables and constants.*

Linear expressions involve only linear functions and terms and non-linear expressions involve atleast one non-linear function or term.

Definition 2.5 *A constraint is a condition on the values of a set of variables expressed using some specified notation on expressions involving the set of variables.*

Constraints involving only linear expressions are linear constraints, and those involving at least one non-linear expression are non-linear constraints.

Definition 2.6 *A constraint system is a triple* $< \tau, \mathcal{V}, \phi >$, *where* τ *is a type system,* $\mathcal{V}$ *is a set of variables which are instances of the types in* τ, *and* ϕ *is a set of constraints on variables in* $\mathcal{V}$.

ϕ is referred to as the constraint specification for the system.

Definition 2.7 *The evaluation of a constraint system* $< \tau, \mathcal{V}, \phi >$ *determines a mapping* $\varepsilon : \mathcal{V} \rightarrow \xi$ *where* ξ *defines the set of allowed points under the set of constraints* ϕ *in the state space defined for the variables in* $\mathcal{V}$ *under the type system* τ.

2.2 Constraint Types

This section briefly discusses the common constraint types that arise in constraint systems. Our system handles multi-way (Section 2.2.2), linear and non-linear (with some restrictions) constraints. Other types of constraints can be easily included as future extensions.

2.2.1 Linear and Non-linear Constraints

Linear and non-linear constraints have been distinguished in Section 2.1. While many of the earlier constraint systems dealt with linear constraints, a number of current systems such as CAL [SA89] and CLP(BNR) [OB93], and architectures [MR95, Rue95, Ste93] handle non-linear constraints. Interesting areas of related research are linear and non-linear programming.

2.2.2 One-way and Multi-way Constraints

One-way constraints compute a function and assign the result to a variable. For example, $a == b + c$ (treated as a one-way constraint) evaluates $b + c$ and assigns it to a. Multi-way constraints allow any variables in a constraint to be altered to satisfy the constraint. In the preceding example ($a == b + c$ is is now treated as a multi-way constraint), if a changes then either b or c can be altered. The constraint satisfaction mechanism can treat any constraint as either one-way or

multi-way. Multi-way constraints are more powerful than one-way constraints but are less efficient because the satisfaction mechanism has to decide which variable to change as well as solve for that variable. The process of selecting the variable to be changed is referred to in AI as *planning* and is typically done at runtime. We execute this process during the compilation phase in our system.

2.2.3 Hierarchical Constraints

A set of constraints without a solution is *over-constrained*. On encountering an over-constrained set, the constraint-satisfaction mechanism can either abort its operation or attempt to satisfy a subset of the constraints. It can be aided in the latter process by a *constraint hierarchy* [BDFB$^+$87] which specifies an ordering on the use of the constraints according to their desired priorities.

2.2.4 Higher-Order Constraints

Higher-order constraints specify constraints on other constraints. An example is

$$\text{if } z \neq 0 \text{ then } z == x + y$$

This if/then constraint takes a predicate and a first-order constraint as arguments to make a second-order constraint. Such constraints can be treated as Boolean combinations of first-order constraints and then solved.

2.2.5 Meta-Constraints

Meta-constraints specify constraints on the constraint-satisfaction mechanism, i.e., constraints on how other constraints are to be satisfied. They may be used, for example, to specify the accuracy to be achieved by an iterative process to solve the constraints. They may also be used to specify the different conditions under which a variety of approaches to constraint satisfaction are to be used.

2.2.6 Temporal Constraints

Many real world problems involve constraints between time and other objects. An animation is a good example where the position of an object is a function of time. Time is an independent variable whose value is given by a "clock" outside the constraint system. Time may also be incorporated in a constraint system as a sequence of values for instances of types.

2.3 Constraint Graphs

A set of constraints can be represented as a *constraint graph* [Lel88] which is commonly used in many constraint systems as a representation for further processing during the constraint satisfaction phase. Figure 2.1 shows the constraint graph corresponding to the constraint $F == 32 + 1.8 \times C$.

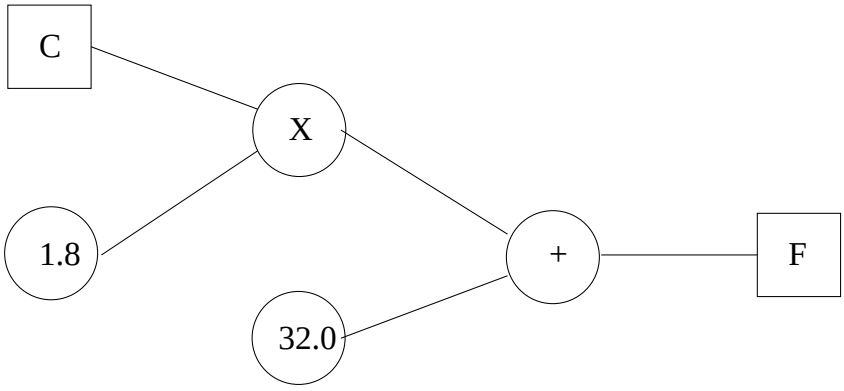


Figure 2.1: Constraint Graph for Temperature Conversion Program

Variables are represented as square nodes and operators as round nodes. The operands to an operator are connected on its left side and the result of applying the operator is connected on its right side. Hence, the "equal to" operator (==) is implicitly represented in the graph. Constants such as 32.0 are operators with no operands. We use a modified form of this representation in our system.

2.4 Constraint Satisfaction Techniques

In this section we review some of the techniques commonly used to evaluate a constraint system. The relationship between these techniques and ours will be given in Section 2.5.

2.4.1 Local Propagation

Local propagation [SS79], a simple and popular constraint-satisfaction mechanism, propagates known values along the arcs of the constraint graph. An operator or variable node can *fire* upon receiving sufficient information from the arcs connecting to it. It then calculates values for arcs that do not contain any and propagates these values out. Thus local propagation uses local information at each node. Figures 2.2 and 2.3 show the values propagated along the arcs when C is assigned a value of 30 and F is assigned a value of 100, respectively.

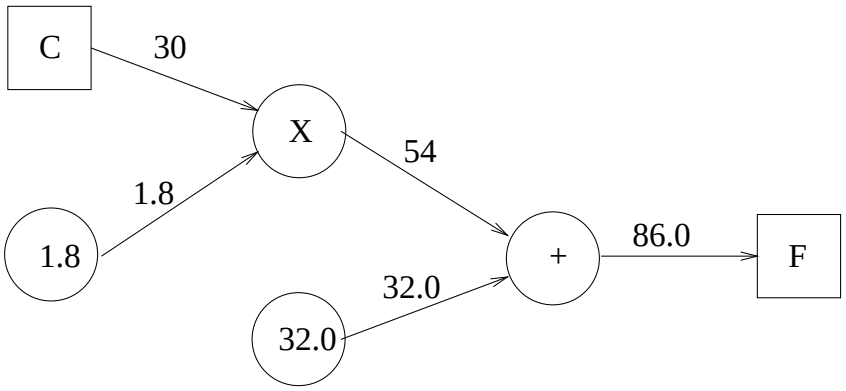


Figure 2.2: Computing the Fahrenheit value of 30 degrees Celsius using Local Propagation

Local-propagation techniques cannot solve constraint graphs with cycles [Lel88].

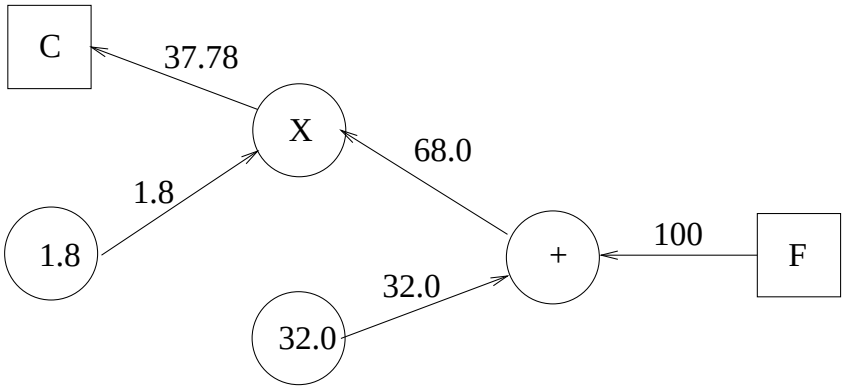


Figure 2.3: Computing the Celsius value of 100 degrees Fahrenheit using Local Propagation

2.4.2 Relaxation

Relaxation [Sut63] solves constraint graphs with cycles by making an initial guess at the values of unknown variables and estimating the error arising out of the guessed values. New guesses are made and the process is repeated until the error is sufficiently small. This technique can be used for overconstrained problems. However, it tends to be slow.

A combination of local propagation and relaxation can be used to solve for a large class of constraints, i.e., sets of constraints with no cycles and sets of constraints with cycles that converge using relaxation. We employ a variation of local propagation and relaxation at compile time to solve for this class of constraints.

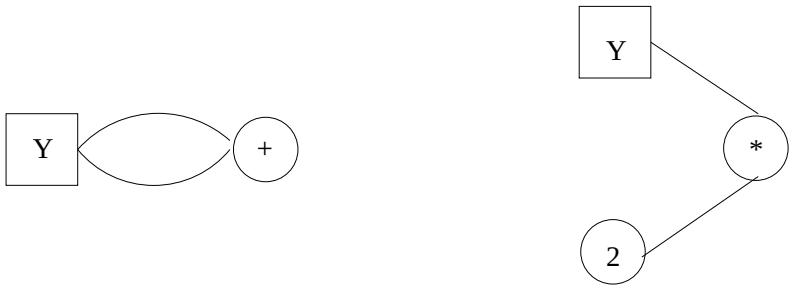
2.4.3 Propagating Degrees of Freedom

Propagating Degrees of Freedom [Lel88] is used when only parts of the constraint graph (containing cycles) need to be relaxed and the rest (without cycles) can be solved by local propagation. The branches connected to cycles are temporarily pruned from the constraint graph. Relaxation is performed on the variables in the cycles to determine their values which are then propagated out to the branches.

Pruning of branches involves searching for an object with few enough constraints (enough degrees of freedom) so that its value can be changed to satisfy the constraints and removing it along with all the applicable constraints. Typically, heuristic methods are used to find objects with enough degrees of freedom.

2.4.4 Graph Transformation

Graph Transformation [Lel88] uses rewrite rules to transform subgraphs of the constraint graph into other graphs which may be simpler to solve. For example, Figure 2.4(a) illustrates how an expression $Y + Y$ creates a cycle. The rewrite rule $X + X \Rightarrow 2 \times X$ eliminates the cycle as shown in Figure 2.4(b). While local propagation is restricted to checking a single node and the associated arcs, graph transformation can look at more of a constraint graph. However, it is still limited to looking at locally connected subgraphs. Most cycles formed by simultaneous equations cannot be solved by graph transformation.



(a) Y + Y (b) 2 * Y

Figure 2.4: (a) Cycle in a Constraint Graph (b) Using a Rewrite Rule to break the Cycle

2.4.5 Miscellaneous Techniques

Equation Solving techniques in symbolic-algebra systems [Mat83] are used to solve constraint programs with cycles. In practice, these are difficult to implement and are slow.

Other techniques that can be used for special forms of constraint systems are related to linear programming, truth maintenance systems [Doy77], theorem-proving methods such as resolution, and artificial intelligence techniques such as searching [NS63].

2.5 Our Approach

Most of the techniques described in Section 2.4 are applied at runtime because they are used in conjunction with local propagation which propagates values of variables. Consequently, execution tends to be slow and these techniques fall short of competing with computation expressed in procedural languages. Performance being one of the crucial issues in parallel systems, conventional constraint satisfaction techniques are unsuitable for forming the basis of a constraint satisfaction mechanism for parallel execution. This thesis focuses on compilation of constraints to parallel procedural code [JJ96].

We define a constraint program to be a constraint system (the triple $< \tau, \mathcal{V}, \phi >$ presented in Definition 2.6 in Section 2.1) and an input set $\mathcal{I}$, where $\mathcal{I} \subseteq \mathcal{V}$. A generalized dependence graph [Bro90] is compiled from the constraint program which computes values for the variables in $\mathcal{V} - \mathcal{I}$ and satisfies the constraints in ϕ. A generalized dependence graph is a parallel computation structure where nodes are atomic units of computation consisting of a mapping from inputs to outputs and a firing rule or guard which is a specification of the states from which execution of the unit of computation can begin [Bro90, WBS+91]. Arcs in the dependence graph

specify dependency relations which are the structuring elements used to compose units of computation into parallel computation structures. A simple example of a dependence graph is shown in Figure 4.15 where the nodes labeled *start* and *stop* initiate and terminate computation, respectively. The conditions on the two arcs from the start node ($N < 10$ and $M < 20$) specify firing rules determining the conditions under which the destination nodes can fire for execution. The computations $for \ldots x[i] = f(i)$ and $for \ldots y[i] = g(i)$ are performed if the corresponding nodes are executed.

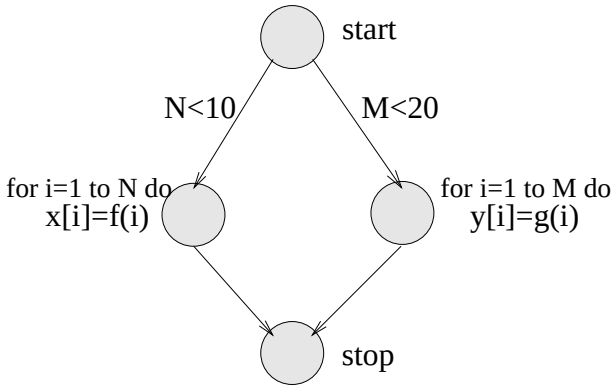


Figure 2.5: A Simple Dependence Graph

The compilation process for the constraint system $< \tau, \mathcal{V}, \phi >$ and input set $\mathcal{I}$ extracts conditionals (firing rules) and computations for the variables in $\mathcal{V} - \mathcal{I}$ from the constraints in ϕ and generates an ordering relation on them to construct a generalized dependence graph which is mapped to the target language CODE [NB92], which expresses parallel structure over sequential units of computation declaratively as a generalized dependence graph. Effectively, the applied technique creates a compiled version of local propagation. Relaxation is used for resolution of dependence graphs with cycles.

The software architecture or execution environment to which CODE is to compile is separately specified (SMP, DSM, NOW, etc). Sequential and parallel C

programs for shared memory machines such as the CRAY J90, SPARCcenter 2000, and the distributed memory PVM [GBD+94] system can be generated. An MPI [Fos95] backend for CODE is also available.

The granularity of the derived dependence graphs and the family of computations which can be expressed depend 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. Additionally, the operations over instances of structured types are often available as modules in libraries. The general requirements for a constraint representation which can be compiled to execute efficiently, include (i) modularity for reusable modules, (ii) definition of atomic functions, and (iii) a rich type set. The importance of modularity will be discussed in a later section while the need for (ii) and (iii) follows from the preceding discussion.

It is not at all surprising that a constraint specification of a computation can be compiled to a parallel program. It is obvious that a set of constraints defined over a set of typed instances of data structures and the choice of an appropriate subset of type instances as an input set defines a dependence graph. Indeed all compilation processes for text string representations of computations, procedural or declarative, whether targeting sequential or parallel execution, derive some form of dependence graph which is then mapped upon the target execution architecture. And there are several compilers for constraint systems to sequential programs [FB89, IWC+88]. *What is surprising is that the advantages for parallel compilation of a specification which is free of control structure has not been previously recognized.* The process of compiling a computation expressed as a constraint system to a parallel program is

actually the derivation of a parallel algorithm expressing the computation in terms of the type system of the constraint language.

We show how compilation of a computation expressed as a constraint system allows extraction of all the parallelism which is intrinsic to the computation. The type system of the constraint representation is critical to the effectiveness of the compilation process. The constraint representation used in the system (described herein) is based on a hierarchical type system [CB95] where matrix semantics are layered upon a hierarchical array type.

2.5.1 Types of Constraints Resolved through Our System

Our basic compilation algorithm can be applied to both linear and non-linear constraints without cycles. Cyclic constraints such as simultaneous systems of equations cannot be resolved by the basic compilation algorithm. The extended compilation process described in Chapter 5 generates programs which resolve cyclic systems through iterative solution algorithms.

The implemented compiler handles all types of linear and non-linear constraints where the initialization results in all non-linear terms being known at runtime. A detailed discussion is given in Chapter 4. This restriction could be alleviated by an extension to the compiler to incorporate higher order solvers for unknown non-linear terms into the compiled program.

All invoked functions must have defined inverses, otherwise compilation is only successful for cases where all parameters of the functions are known at runtime.

Constraint systems involving inequalities must be cast by the compilation process to conditional expressions where all of the variables are evaluatable at runtime.

2.6 Constraint Systems as Representations of Parallel Computations

2.6.1 Conformance to Desired Property Set

Let us now analyze the properties of constraint systems in terms of the desirable properties given in Section 1.2.

• Property 1.1 (Naturalness of Expression) (assuming an appropriate set of types are present in the representation) obtains because constraint specifications are mathematical relations familiar to all scientists and engineers.

• Property 1.2 (Full Parallelization) obtains because constraint systems do not directly specify any control flow model at all. Therefore any mode of parallel execution is equally realizable. The granularity of the operations is determined by the operations defined in the type system.

• Property 1.3 (Specification of Execution Properties) follows if the type system is sufficiently rich and expressive. The type system is critical to the representation and merits a separate discussion.

• Property 1.4 (Reuse of Components) follows because the constraint specification does not specify how a relation involving operators is to realize the operations defined in the type system leaving the compilation process free to define an implementation which can be realized with components from libraries of standard components.

• Property 1.5 (Adaptation of Program to Execution Environment) follows because properties 1.2 (Full Parallelization) and 1.4 (Reuse of Components) are properties of constraint representations as defined herein.

• Property 1.6 (Portability with Efficiency) follows because properties 1.2 (Full Parallelization) and 1.4 (Reuse of Components) are properties of constraint systems as defined herein.

2.6.2 The Role of the Type System

Realization of all of the desirable properties depends upon the type system over which the constraint relations may be specified.

• The type system must support compact and natural expression of the operations of the application domain if property 1.1 (Naturalness of Expression) is to be obtained.

• The operations on instances must generally be invertible to support transformation of constraints to equations defining computations.

• The granularity of instances of structured types must be parameterizable if property 1.3 (Specification of Execution Properties) is to be obtained.

• Parallel algorithms for implementing certain operations on structured types may be necessary.

The domain we have chosen for this demonstration of the feasibility of compiling efficient procedural programs from constraint specifications is numerical computations in general and computations over matrices in particular. This is a domain which, while compact enough for all of the requirements on the type system listed above to be met, is also the basis for a large fraction of the computations of engineering and science.

One principle innovation in this constraint language system is the introduction of a hierarchical matrix type [CB95] as a primitive type in the constraint

language. A hierarchical matrix type may include a specification of a structure for the matrix (say triangular), a composition rule for the block structure of the matrix and a specification for the structure of the composing blocks. The constraint relations must be expressed directly in the primitive types provided by the specification language or as invocations of modules. Details of the type system are provided in Chapter 3.

Hierarchical matrices are necessary to provide an adequate direct representation of matrix computations because

(a) Many interesting computations are defined in terms of block structured matrices where the matrix arising from the discretization of a partial differential equation is composed of blocks of sub-matrices, each block of which may have a specific structure.

(b) There are often efficient parallel algorithms for operations on matrices of special structure such as triangular or banded matrices.

We have not yet implemented the full feature set for hierarchical matrix types but only a feature set sufficient for a feasibility demonstration spanning a reasonable set of algorithms. We still find it necessary and useful to express some matrix computations in terms of operations at the scalar level as well. Both modes of representation will be illustrated.

2.6.3 Modular Structure

Both the need to implement new operations and practical software engineering requires that the constraint system have a modular structure. The compilation algorithm is in principle np-hard so that modularity may ultimately be necessary in order to enable compilation of very large programs.

Chapter 3

The Constraint Language

This chapter describes the components of our programming system. It explicates the type system, the rules for expressing constraints, and the structure of a complete program in the system. The chapter concludes with the constraint specifications for a few sample programs. The notations used are similar to those in the C programming language.

3.1 Type System

Our approach relies on a rich hierarchical type system where types at higher levels are constructed from those at lower levels in the hierarchy. The schematic for the layout of the type system is shown in Figure 3.1. The lowest level of the type hierarchy contains integers, reals, and characters. At the next level of the hierarchy are arrays to which we give semantic structure to construct the base matrix types, which define matrices of scalar elements. In addition to dense matrices, the base matrix type currently supports specialized matrix types such as lower and upper triangular enabling the flexibility to invoke specialized algorithms based on the structure of the matrix for the operations defined on the matrix subtypes. Other specialized

types can also be easily incorporated. At the highest level of the type system are hierarchical matrices, whose individual elements are matrices.

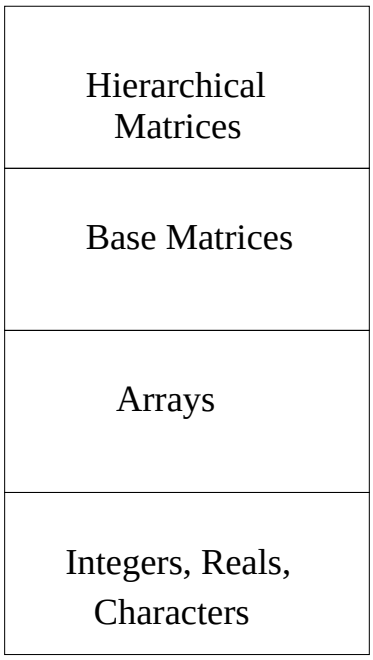


Figure 3.1: The Type System Layout

With respect to Definition 2.1 introduced in Section 2.1, the entities in the type system are integers, reals, characters, arrays, base matrices and hierarchical matrices. The operators of addition, subtraction, multiplication, and division are defined on integers and reals. The operator sets for characters and arrays are empty. The base matrix type has associated operators of addition, subtraction, scalar multiplication, matrix multiplication and inverse defined for matrices over integers and reals. The operator set for hierarchical matrices is empty since operations are only defined on the blocks which compose it.

3.2 Expressions

Expressions can be formed by applying defined operators on instances of types in the type system and through calls to library and user-defined functions. Functions must have defined inverses, otherwise only a limited form of compilation can be done. Examples of library functions are mathematical functions such as *sqrt* and *sqr*.

In addition to defined applications of operators, expressions of the following form using *indexed operators* are allowed.

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

An indexed operator applies a binary operator *op* to an expression X through a range of values $b1 \ldots b2$ for an integer variable *index*. The values of *b1* and *b2* have to be bounded at compile-time. An indexed operator allows for the compact representation of expressions and is useful in large systems. For example, the construct

$$+\ FOR\ (i\ 1\ 3)\ \{\ +\ FOR\ (j\ 1\ i)\ \{\ A[i][j]\ \}\}$$

expresses the sum of the lower partitions of a 3×3 matrix A:

$$\begin{aligned} &A[1][1]+ \\ &A[2][1] + A[2][2]+ \\ &A[3][1] + A[3][2] + A[3][3] \end{aligned}$$

The "+" operator refers to scalar addition or matrix addition depending on whether A is a base matrix or a hierarchical matrix, respectively.

3.3 Constraints

Rules which govern the specification of constraints are enumerated in this section. In designing these rules we have the motivation of capturing the entire set of con-

straints a programmer would wish to impose upon a system. Rule 1 allows for the expression of simple conditions, using relational operators, on expressions involving type instances. Rule 2 allows propositional connectives AND/OR/NOT to be applied on constraints to express conditions using compositions of constraints. Rule 3 is a generalization of Rule 2 through which large compositions of constraints using AND/OR operators can be compactly represented. Rule 4 introduces modularity to enable large bodies of constraints to be replaced by calls to reusable modules.

Rule 1:

(i) $X_1 \ \mathcal{R} \ X_2$, is a constraint,

where $\mathcal{R} \in \{ <, <=, >, >=, == , ! = \}$,

X_1, X_2 are expressions over instances of scalar types.

(ii) $M_1 == M_2$ is a constraint,

where M_1, M_2 are expressions involving matrices and matrix operators.

Rule 1(ii) allows a mix of scalars and matrices. Although we do not currently allow relations of the form $M_1 < M_2$, these could be easily defined to extend expressibility.

Rule 2:

(i) A AND/OR B

(ii) NOT A

are constraints, where A and B are constraints.

Rule 3: Constraints over *indexed sets* have the form:

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

An indexed set 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*. The values of $b1$ and $b2$ have to be bounded at compile-time. This condition will be relaxed in later versions of the compiler. Indexed sets allow for the compact representation of large constraint systems.

An application of Rule 3 is

$$\text{AND FOR (i 1 2) } \{ A[i] == A[i-1],\ B[i+1] == A[i] \}.$$

This concise construct represents the constraint

$$A[1] == A[0] \text{ AND } A[2] == A[1] \text{ AND } B[2] == A[1] \text{ AND } B[3] == A[2].$$

Another application of Rule 3 is

$$\text{OR FOR (i 1 2) } \{ A[i] == 0,\ B[i+1] == A[i] \}.$$

This construct succinctly captures the constraint

$$A[1] == 0 \text{ OR } A[2] == 0 \text{ OR } B[2] == A[1] \text{ OR } B[3] == A[2].$$

Rule 4: Calls to user-defined constraint modules are constraints. They have the form:

$$< ModuleName > (P_1, P_2, \ldots, P_n)$$

where $ModuleName$ is the name of a defined constraint module (Section 3.4 describes definition of constraint modules), which encapsulates constraints between its formal parameters, local variables, and global variables within its scope. $P_1, P_2, \ldots, P_n$ are the actual parameters for the constraint module call.

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. Both linear and non-linear constraints can be expressed using

these rules. Each rule has an analog in the procedural world - Rule 1 maps to simple conditionals and simple computations such as assignments, Rule 2 to sequencing and conditional statements, Rule 3 to loops and Rule 4 to procedures.

3.4 Program Structure

A program in our system has the following constituents.

(i) Program name.

(ii) Global variable declarations.

(iii) Global input variables: input set $\mathcal{I}$.

(iv) User-defined function signatures: signatures of C functions, which may be invoked in expressions. For example, the user-defined function *max* in the constraint $max(a, b) < 5$ may have the function signature *int max*(*int x*, *int y*). The actual function definitions are provided in a separate file which is linked with the compiled executable for the constraint program.

(v) Constraint module definitions: module name, formal parameters and their types, local variable declarations, and a *constraint module body* constructed from applications of Rules 1-4 in Section 3.3. Constraints within a module can involve local variables, formal parameters, and global variables. Name scoping and type matching are similar to those implemented for procedures in C programs.

(vi) *Main body* of the program: constraints on global variables expressed through applications of Rules 1-4 in Section 3.3.

3.5 Sample Programs

This section presents four example programs written using the language constructs presented in Section 3.3. While the first one is a toy example, the others have been successfully executed with good performance results.

3.5.1 The Quadratic Equation Solver

Figure 3.2 shows a constraint specification for the non-complex roots of a quadratic equation $ax^2 + bx + c == 0$. *sqr*, *sqrt*, and *abs* are library functions. The main body specifies the conditions on values of the roots $r1$ and $r2$ when $a == 0$ and when $a != 0$. The condition on the values of $r1$ and $r2$ when $a != 0$ is expressed by a call to a constraint module *DefinedRoots*. The definition for the module expresses the relationship between the parameters $a, b, c, r1, r2$ in the event that the discriminant (t) is greater than or equal to 0. The specification can be enhanced for imaginary roots. The input set could be $\{a, b, c\}$, $\{a, b, r1\}$, or $\{a, b, r2\}$. Other input sets will not lead to dependence graphs through the compilation process described in Chapter 4.

PROGRAM QUAD_ROOTS

VAR real a, b, c, r1, r2; /* **Global Variables** */
INPUTS a, b, c; /* **Input Variables** */

/* **Constraint Module** */
DefinedRoots(a: real; b:real; c:real; r1:real; r2:real)
real t; /* **Local Variable** */

/* **Constraint Module Body** */
$t == sqr(b) - 4 * a * c$ *AND* $t >= 0$ *AND*
$2 * a * r1 == (-b + sqrt(abs(t)))$ *AND* $2 * a * r2 == -(b + sqrt(abs(t)))$

/* **Main Body** */
$a == 0$ *AND* $b! = 0$ *AND* $r1 == r2$ *AND* $b * r1 + c == 0$
OR
$a! = 0$ *AND DefinedRoots(a, b, c, r1, r2)*

Figure 3.2: Constraint Specification for the Quadratic Equation Solver

3.5.2 The 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 matrix and vectors can be divided into blocks as shown in Figure 3.3. $S_0 \ldots S_3$ represent lower triangular sub-matrices along the diagonal of A and $M_{10}, M_{20}, \ldots M_{32}$ represent dense sub-matrices within A.

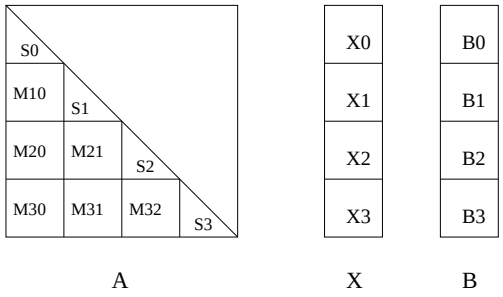


Figure 3.3: BTS: Partitioned Lower Triangular Matrix A, Vectors X and B

A constraint specification (excluding declarations) for a problem instance split into 4 blocks is shown in Figure 3.4. The input set can be chosen as $\{ S_0, \ldots, S_3, M_{10}, M_{20}, \ldots, M_{32}, B_0, \ldots B_3 \}$. The constraint specification closely imitates the mathematical representation of the partitioned version of the problem $AX == B$.

PROGRAM BTS_1

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

Figure 3.4: Constraint Specification for the BTS System

Using an indexed set of constraints and an indexed operator, an alternate compact program is shown in Figure 3.6 using partitions on A as shown in Figure 3.5. The input set can be chosen as $\{ A, B \}$ to yield the solution for X. Alternatively, $\{ A, X \}$ can be chosen as the input set to yield a solution for B. Other input sets will not yield solutions through the compilation process described in Chapter 4.

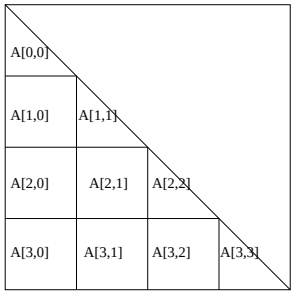


Figure 3.5: BTS: Partitioned Lower Triangular Matrix A

PROGRAM BTS_2

AND FOR (i 0 3) $\{$ *+ FOR (j 0 i)* $\{ A[i][j] * X[j] \} == B[i] \}$

Figure 3.6: Alternate Notation for the Constraint Specification for the BTS System

3.5.3 The Block Odd-Even Reduction Algorithm(BOER)

This is an example deliberately chosen by us to demonstrate that constructing the constraint specification by inspecting a given algorithm and processing it through the compiler extracts the original algorithm if an appropriate input set is chosen (shown later in the thesis). Consider a linear tridiagonal system $Ax == d$ where

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

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 ($B \times C == C \times B$).

A version of the parallel algorithm ([LD90]), shown in Figure 3.7, 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 (excluding declarations) for the problem is shown in Figure 3.8. The variable names BP, CP, and dP correspond to the indexed terms B, C, and d in [LD90] 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 $dP[i][0]$, $1 \leq i \leq M$. *pow* is a C function implementing the arithmetic power function. Note that the constraints have been constructed by mapping assignments (=) in the algorithm to equality (==) in the constraint specification and for loops to indexed sets. The *INITIALIZATION* phase corresponds to providing $BP[0]$, $CP[0]$ and $dP[i][0]$ as inputs. Also, the three constraints corresponding to

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

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

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

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

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

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

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

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

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

$$E(j) = \begin{bmatrix} B(j-1) & 0 & 0 & \dots & 0 & 0 \\ 0 & B(j-1) & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & B(j-1) & 0 \\ 0 & 0 & 0 & \dots & 0 & B(j-1) \end{bmatrix},$$

$$w(j) = \begin{bmatrix} x_{t-s} \\ x_{2t-s} \\ \vdots \\ x_{it-s} \\ \vdots \\ x_{2^{k-j}t-s} \end{bmatrix},$$

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

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

Figure 3.7: Parallel Algorithm for the Cyclic Block Tridiagonal System

```
PROGRAM BOER

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

AND

/* REDUCTION PHASE */
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

/* BACK-SUBSTITUTION PHASE */
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 3.8: Constraint Specification for the BOER System

the reduction, single-solution, and back-substitution phases have been reordered to demonstrate the independence of a constraint specification on the expressed order of the constraints.

3.5.4 The Laplace Equation

Consider the Laplace equation for a 4-point stencil on an $N \times N$ grid indexed by $(0 \ldots N-1)(0 \ldots N-1)$ as shown in Figure 3.9 for $N = 10$. The boundary elements (shaded) are inputs to the problem. Every element not on the boundary is the average of its four neighbors. Since there are $(N-2) \times (N-2)$ non-boundary elements, there are $(N-2) \times (N-2)$ constraints to satisfy.

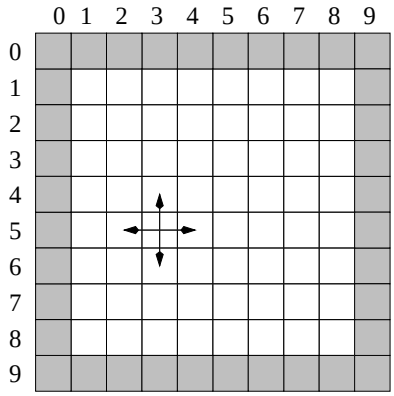


Figure 3.9: The Laplace Equation Grid

A constraint specification (excluding declarations) for the problem is shown in Figure 3.10. x is an array of dimensions ranging in $(0..N-1, 0..N-1)$. The simple constraint 4 * x[i][j] - x[i-1][j] - x[i+1][j] == x[i][j-1] + x[i][j+1] in the specification can be expressed in many equivalent representations including 4 * x[i][j] == x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1]. The specified set of constraints in Figure 3.10 forms a set of cyclic constraints. This program is an example of constraints over scalar elements of a structured type.

```
PROGRAM LAPLACE

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

Figure 3.10: Constraint Specification for the Laplace Equation System

Chapter 4

The Basic Compilation Algorithm

The constraint compiler transforms a textual program given in the format outlined in Section 3.4 to a sequential or parallel C program for a selected architecture such as a Sparc, Cray, PVM, or MPI configuration. This chapter discusses the basic compilation algorithm [JB96a] which handles constraint systems without cycles (see Chapter 2). We discuss an enhancement to the basic algorithm for constraint systems with cycles in Chapter 5. The compilation algorithm consists of the following phases.

Phase 1. The textually expressed constraint specification is transformed to an undirected graph representation as for example given by Leler [Lel88].

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

Phase 3. With a set of input variables $\mathcal{I}$, the directed graph is traversed in a depth-first manner to map the constraints in the constraint specification 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 [NB92].

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 the rest of this chapter. Phases 1-3 will be illustrated through the quadratic equation solver introduced in Figure 3.2 and whose constraint specification (without declarations) has been repeated in Figure 4.1 for convenience.

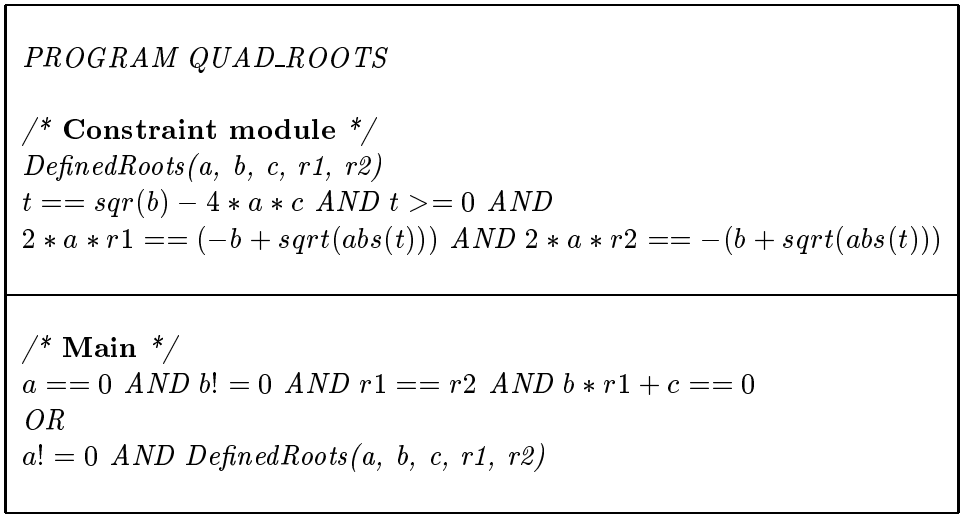


Figure 4.1: Constraint Specification for the Quadratic Equation Solver

4.1 Phase 1: Generation of Constraint Graphs

A parser transforms the textual source program to a source graph for the compiler. Starting from an empty graph, for each application of Rules 1-4 in Section 3.3 an undirected constraint graph can be constructed by adding appropriate nodes and edges to the existing graph. For each instance of a simple constraint (Rule 1) a node

is created with the constraint attached to it as shown in Figure 4.2(a). For each application of Rule 2 (A AND/OR B, NOT A) the graph is expanded as shown in Figures 4.2(b),(c). Figure 4.3(a) illustrates the expansion of the constraint graph for each application of Rule 3 (AND/OR FOR (<index> <b1> <b2>) { $A_1, A_2, \ldots, A_n$ }). For each application of Rule 4 ($< ModuleName > (P_1, P_2, \ldots, P_n)$) a node is created with the constraint module call and the actual parameters attached to it as shown in Figure 4.3(b).

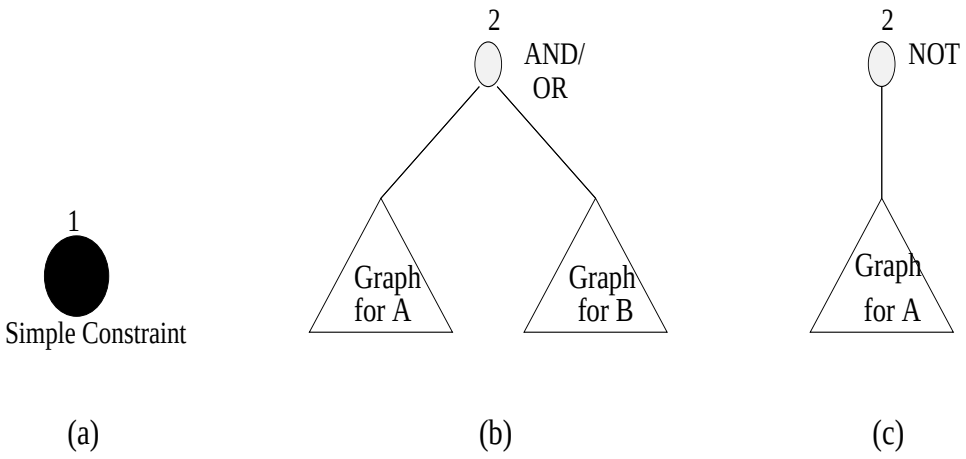


Figure 4.2: Constraint Graphs for (a) Rule 1 (b),(c) Rule 2

The different kinds of nodes in the constraint graph are (i) *simple constraint* nodes (1 in Figure 4.2(a)) (ii) *operator* nodes corresponding to AND/OR/NOT connectives (2 in Figures 4.2(b),(c)), (iii) *for* nodes corresponding to indexed sets (3 in Figures 4.3(a)), and (iv) *call* nodes corresponding to Constraint Module Calls (4 in Figure 4.3(b)). The index and its range information for an indexed set are attached to the corresponding *for* node.

A constraint graph is constructed for the main body and for each of the constraint module bodies giving rise to a set of constraint graphs. Each graph is constructed in a hierarchical fashion. *Simple constraint* and *call* nodes occur at lower levels, and *operator* and *for* nodes connect one or more subgraphs at higher levels.

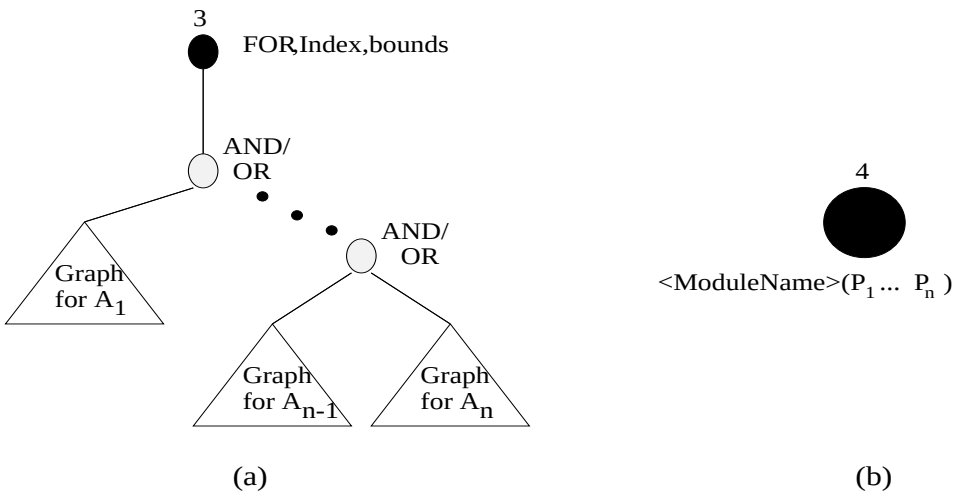


Figure 4.3: Constraint Graphs for (a) Rule 3 (b) Rule 4

There will be a single node 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 4.4.

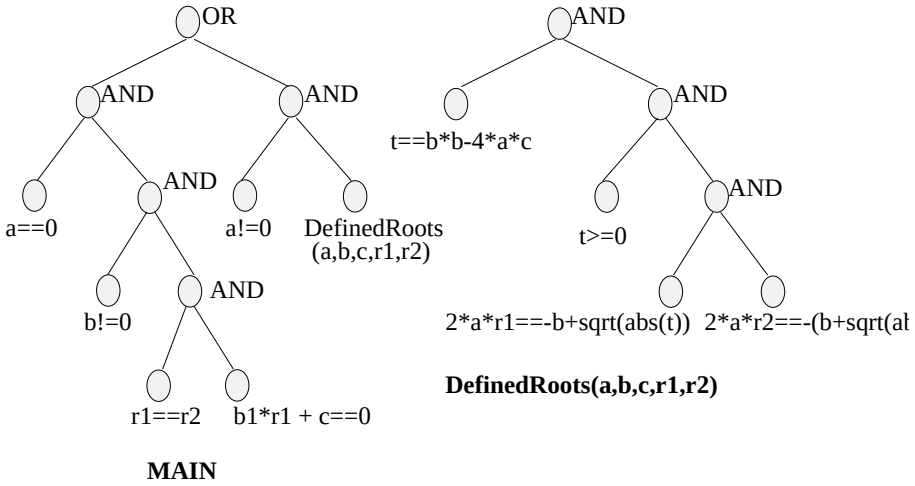


Figure 4.4: Constraint Graphs for the Quadratic Equation Solver

4.2 Phase 2: Translation of Constraint Graphs to Directed Graphs

A depth-first traversal of each graph in the set of constraint graphs obtained from the main body and the constraint module bodies constructs a set of directed graphs which are trees. The tree corresponding to the main body is referred to as the *main tree*. 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.

Figure 4.5 illustrates phase 2 for four base cases, where a, b, c, and d are simple constraints. There is a potential for combinatorial explosion in case 4 which corresponds to the applying the distributive law: (a OR b) AND (c OR d) $\equiv$ (a AND c) OR (a AND d) OR (b AND c) OR (b AND d).

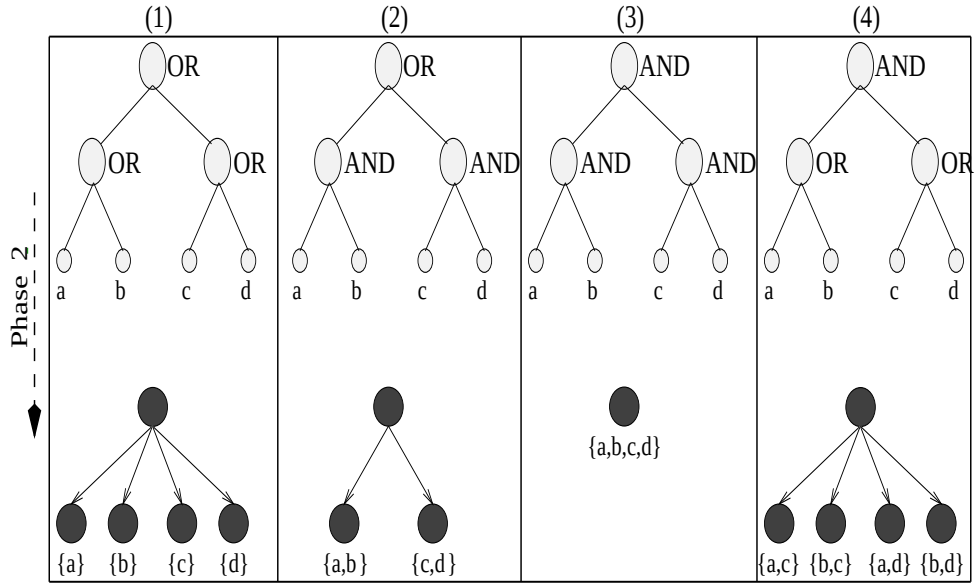


Figure 4.5: Phase 2 for four base cases

The resulting trees in this phase do not contain any AND/OR nodes. Instead, a node in a tree may contain a list of simple constraints, indexed sets, or constraint module calls. However, AND/OR nodes are implicitly represented in a tree since all constraints along a path are connected by the AND operator and constraints on different paths are connected by the OR operator. 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. Different paths, being implicitly connected by the OR operator, represent different ways of satisfying the constraint system.

The algorithm *dft* is a generalization of Figure 4.5. 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. dft is initially invoked with the call dft(v_1, v_1^*).

The case of the *operator* node NOT has been omitted from the description of *dft*. However, it is implemented in the system as follows. A NOT operator node operates on a single constraint subgraph. It 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 or another NOT node. If it reaches a simple constraint, the NOT node is removed by negating the simple constraint. If it reaches another NOT node, both NOT nodes are removed from the graph.

ALGORITHM dft (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 dft(u, v_c^*)

else create node u^* in G^* as child of v_c^*;

dft(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;

/*(u_{11} OR u_{12}) AND $u_2 \equiv (u_{11}$ AND $u_2)$ OR (u_{12} AND u_2)*/

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;

dft(v_c, v_c^*);

else for each unvisited neighbor u of v_c **do** dft(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 node at highest level of constraint graph in indexed set;

dft(v_i, $v_i{}^*$);

end;

The trees obtained for the quadratic equation solver through phase 2 are shown in Figure 4.6.

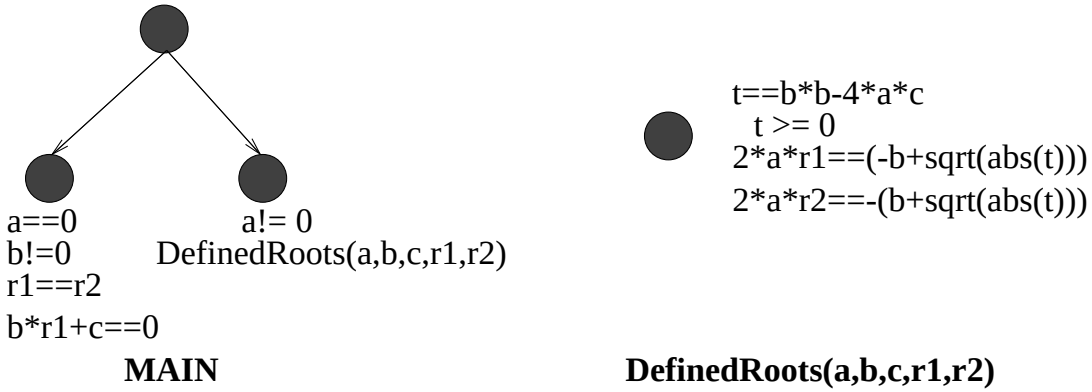


Figure 4.6: Trees from Phase 2 for the Quadratic Equation Solver

4.3 Phase 3: Generation of Dependence Graphs

Using the input set $\mathcal{I}$, a depth-first traversal of the main tree T_{main} 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 which has no arcs directed into it and whose inputs are in $\mathcal{I}$. Hence, the start node can be executed exactly once at the initiation of the computation. 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 4.7). 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.

At the initiation of phase 3, a dependence graph G is constructed which is similar in structure to T_{main}, i.e., there is a 1-1 mapping between nodes and arcs in

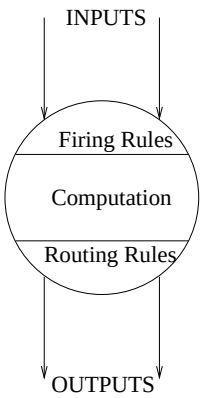


Figure 4.7: Generalized Dependence Graph Node

T_{main} and the nodes and arcs in G, respectively. The node in G corresponding to the root of T_{main} is designated as the start node. The structure of G may change later as detailed in Sections 4.3.2 and 4.3.3. The nodes in G are initially empty.

A *known* set is associated with each node in the dependence graph G. 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. A child node in the dependence graph inherits the known set of its parent when the node in T_{main} corresponding to the child node is visited during the depth-first traversal.

When a node in T_{main} is visited, constraints at that node may be *resolved* through processes detailed in Sections 4.3.1, 4.3.2, and 4.3.3 into computations or conditionals (firing/routing rules) of the corresponding node in G. Any constraint which cannot be resolved is retained in an unresolved set of constraints which is propagated down T_{main} to other nodes through the depth-first traversal in the hope that it may get resolved later. A number of passes may be made through each constraint at a node and the propagated unresolved set of constraints for resolution of these constraints. A new pass is initiated if at least one constraint was resolved in

the previous pass; otherwise the depth-first traversal proceeds to visit the next node. Treatment of constraints remaining unresolved at the leaves of T_{main} is described in Section 4.3.8.

4.3.1 Resolution of Simple Constraints

Each node v in the tree from phase 2 may have a set of simple constraints attached to it. Additionally, the depth-first traversal may have a list of unresolved constraints propagated down from v's parent. Each simple constraint at v or in the unresolved set of constraints 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 at v^* before the first pass through the list of constraints at v and the unresolved set of constraints.

(ii) Computation: To fall into this category a constraint must involve an equality and have a single unknown at v^*. The constraint is cast as a computation at v^* for the unknown which 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^*.

Constraints involving inequalities must be resolved as firing/routing rules. When a constraint is classified as a 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. If the unknown occurs in an actual parameter of a function, the inverse of the function may be applied to extract a computation for the unknown. Currently, our system solves equations in linear unknown terms. Thus non-linear constraints can be currently resolved if the unknown terms are linear. 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 written 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.

4.3.2 Resolution of Indexed Sets

An indexed set AND/OR FOR (<index> <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$ is resolved if every constraint A_i, $1 \leq i \leq n$, is resolved for all values of *index* in $b1 \ldots b2$. Resolved indexed sets are compiled to loops which iterate over values of *index* in $b1 \ldots b2$. If every constraint in a set $S_1 \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as a computation, every constraint in a set $S_2 \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as a firing/routing rule and every constraint in a set $S_3 \subseteq \{A_1, A_2, \ldots, A_n\}$ remains unresolved, the indexed set is split into the following three indexed sets.

(1) An indexed set AND/OR FOR (*index* <b1> <b2>) S_1 resolved as a computation

(2) An indexed set AND/OR FOR (*index* <b1> <b2>) S_2 resolved as a firing/routing rule

(3) An unresolved indexed set AND/OR FOR (*index* <b1> <b2>) S_3

Note that $S_1 \cup S_2 \cup S_3 == \{A_1, A_2, \ldots, A_n\}$ and

$S_i \cap S_j == \phi$ (null set) where $1 \leq i, j \leq 3$ and $i \neq j$.

The restrictions for a constraint $A_i, 1 \leq i \leq n$, in an indexed set structure to be compiled successfully in our system are as follows. For all values of *index* in $b1 \ldots b2$ (a) A_i has to have the same classification (computation/firing rule/routing rule), (b) if A_i is a simple constraint and is classified as a computation, a unique term in the constraint has to be the unknown (a term can be a simple variable x or an indexed term such as $X[< list\ of\ indices >]$, where X is a structured data type). An example of a construct that will be compiled successfully is

$$X[0] == 0 \text{ AND } (\text{ AND FOR (i 1 5) } \{\ X[i-1] == X[i] + Y[i]\ \}\ )$$

with Y known and X unknown.

It will be compiled to the computations

$X[0] = 0;$

for i=1 to 5 do

$X[i] = X[i-1] - Y[i];$

Note that the indexed set is compiled to a loop which computes the value of $X[i]$ in successive iterations.

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

$$\text{AND FOR (i 1 5) } \{\ X[1] == X[i] + Y[i]\ \}$$

with X unknown and Y known. This is because in the first iteration both the terms $X[1]$ and $X[i]$ are unknown whereas subsequent iterations have only $X[i]$ as an unknown (violates (b)).

Resolution of AND Indexed Sets

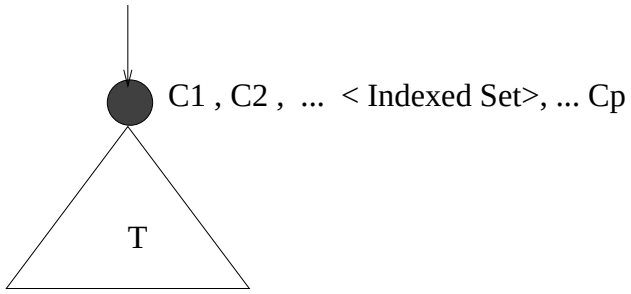


Figure 4.8: Indexed Set at a Node in a Tree from Phase 2

Let an AND indexed set AND FOR (i <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$ occur among constraints $C_1, C_2, \ldots, C_p$ at a node in a tree as shown in Figure 4.8. Evaluate constraints $A_1, A_2, \ldots, A_n$ for classification as firing/routing rules or computations for $i = b1 \ldots b2$. Let $k(1) \ldots k(n)$ be a reordering of the subscripts $1 \ldots n$. Let $\{A_{k(1)},$

$A_{k(2)}$, ... $A_{k(m1)}\}$ be the constraints which evaluate to firing rules for all $i = b1 \ldots b2$. Let $\{A_{k(m1+1)}$ ... $A_{k(m2)}\}$ be the constraints which evaluate to computation for all $i = b1 \ldots b2$. Let $\{A_{k(m2+1)}$ ... $A_{k(m3)}\}$ be the constraints which evaluate to routing rules for all $i = b1 \ldots b2$. Let $\{A_{k(m3+1)}$ ... $A_{k(n)}\}$ be the constraints which remain unresolved.

Similarly, evaluate constraints $C_1, C_2, \ldots, C_p$. Let $r(1) \ldots r(p)$ be a reordering of the subscripts $1 \ldots p$. Let $\{C_{r(1)}, C_{r(2)}$, ... $C_{r(l1)}\}$, $\{C_{r(l1+1)}$ ... $C_{r(l2)}\}$, and $\{C_{r(l2+1)}$ ... $C_{r(l3)}\}$ be the constraints which evaluate to firing rules, computations, and routing rules, respectively and $\{C_{r(l3+1)}$ ... $C_{r(p)}\}$ be the unresolved constraints. The generated dependence graph is shown in Figure 4.9. The unresolved constraints are propagated down T.

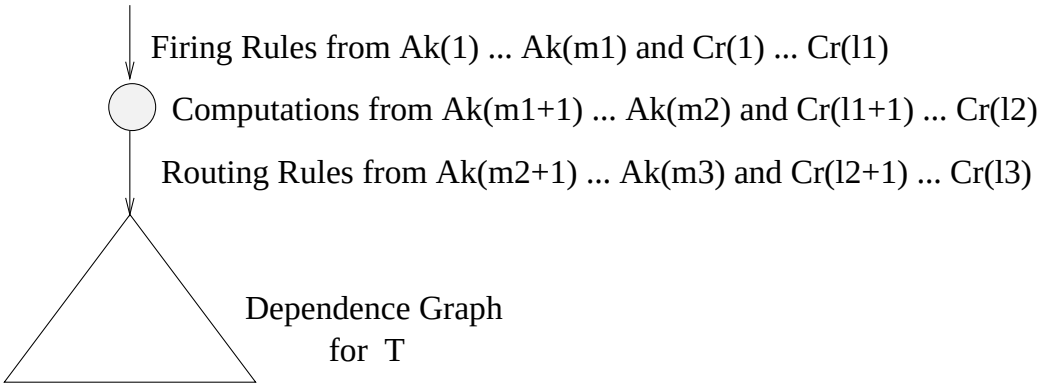


Figure 4.9: Generated Dependence Graph for an AND Indexed Set

The firing rule corresponding to $A_{k(1)}$, $A_{k(2)}$, ... $A_{k(m1)}$ is $A_{k(1)}$ AND $A_{k(2)}$ AND ... AND $A_{k(m1)}$ for all $i = b1 \ldots b2$. A similar construct is set up for the routing rule corresponding to $A_{k(m2+1)}$ ... $A_{k(m3)}$. The computations for $A_{k(m1+1)}$ ... $A_{k(m2)}$ are expressed as

for i = b1 to b2 do

Computation corresponding to $A_{k(m1+1)}$*;*

Computation corresponding to $A_{k(m1+2)}$*;*

$\vdots$

Computation corresponding to $A_{k(m2)}$*;*

Resolution of OR Indexed Sets

Let an OR indexed set OR FOR (i <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$ occur among constraints $C_1, C_2, \ldots, C_p$ at a node in a tree from phase 2 as shown in Figure 4.8. Evaluate constraints $A_1, A_2, \ldots, A_n$ for classification as firing/routing rules or computation for $i = b1 \ldots b2$. Let $k(1) \ldots k(n)$ be a reordering of the subscripts $1 \ldots n$. Let $\{A_{k(1)}, A_{k(2)}, \ldots A_{k(m1)}\}$ be the constraints which evaluate to firing rules for all $i = b1 \ldots b2$. Let $\{A_{k(m1+1)} \ldots A_{k(m2)}\}$ be the constraints which evaluate to computations for all $i = b1 \ldots b2$. Let $\{A_{k(m2+1)} \ldots A_{k(m3)}\}$ be the constraints which evaluate to routing rules for all $i = b1 \ldots b2$. Let $A_{k(m3+1)} \ldots A_{k(n)}\}$ be the constraints which remain unresolved.

Similarly, evaluate constraints $C_1, C_2, \ldots, C_p$. Let $r(1) \ldots r(p)$ be a reordering of the subscripts $1 \ldots p$. Let $\{C_{r(1)}, C_{r(2)}, \ldots C_{r(l1)}\}$, $\{C_{r(l1+1)} \ldots C_{r(l2)}\}$, and $\{C_{r(l2+1)} \ldots C_{r(l3)}\}$ be the constraints which evaluate to firing rules, computations, and routing rules, respectively and $\{C_{r(l3+1)} \ldots C_{r(p)}\}$ be the unresolved constraints. The generated dependence graph is shown in Figure 4.10. The unresolved constraints are propagated down T.

The "Call Node" invokes the dependence graph corresponding to T shown in Figure 4.8. $i : b1, b2$ shows that the associated arc and its destination node are replicated for values of i from $b1 \ldots b2$. The firing rule for $A_{k(1)}, A_{k(2)}, \ldots A_{k(m1)}$ is $A_{k(1)}$ *OR* $A_{k(2)}$ *OR ...* $A_{k(m)}$ *for any i=b1 ... b2.* A similar construct is set up for the routing rule for $A_{k(m2+1)} \ldots A_{k(m3)}$.

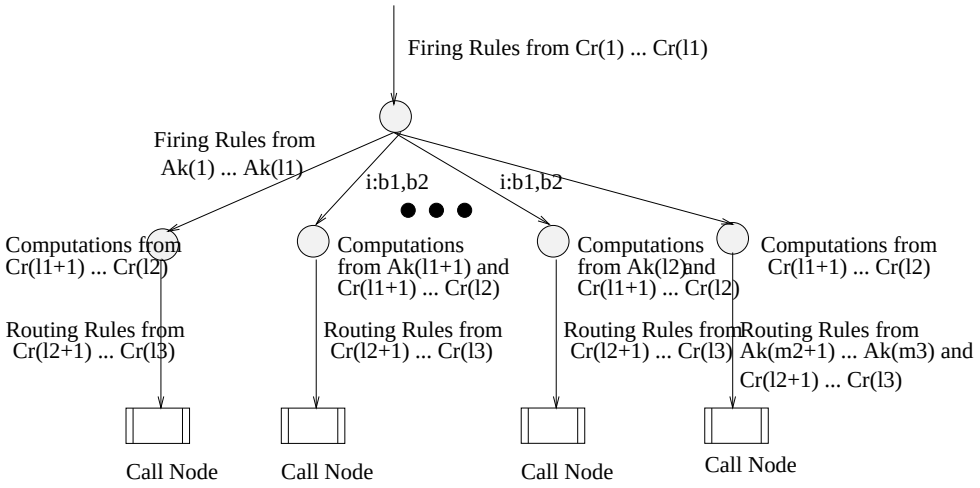


Figure 4.10: Generated Dependence Graph for an OR Indexed Set

4.3.3 Resolution of Constraint Module Calls

A constraint module call has the form $ModuleName(e_1, e_2, \ldots, e_n)$ where e_i, $1 \leq i \leq n$, is an actual parameter. Actual parameters may be expressions. 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 that node in G (dependence graph) which corresponds to the node in T (tree from phase 2) where the constraint module is invoked. If all the variables in $e_1, \ldots, e_n$ and all the global variables occuring in the constraint module body are in K and no local variable occurs in the constraint module body, the call to the constraint module is cast as a firing/routing rule which tests whether the body of the constraint module is satisfied or not.

If the constraint module call cannot be cast as a firing/routing rule, an attempt is made to generate a dependence graph from the constraint module definition. A new dependence graph G_{mod} is created which is similar in structure to the tree T_{mod} from phase 2 for the constraint module, i.e., there is a 1-1 mapping between nodes and arcs in G_{mod} and nodes and arcs in T_{mod}, respectively. T_{mod} is traversed with a new known set K_{module} which is initialized to $\{ f_i \mid \{\text{all variables in } e_i\} \subseteq$

K, $1 \leq i \leq n\} \cup \{\ x \mid x \in K$ and x is a global variable in the scope of the module $\}$. The unknowns are considered to be all formal parameters not in K_{module}, the local variables in the constraint module, and all the global variables not in K but in the scope of the module.

The resolution of constraints in the constraint module is similar to that for the main module with one difference. The dependence graph G_{mod} is retained with only the set of paths with the maximal output set for formal parameters and global variables. For example, let there be 5 paths numbered 1 through 5 with the following computed formal parameters and global variables, respectively. 1: $\{\ a, b\ \}$, 2: $\{\ a\ \}$, 3: $\{\ a, b, c\ \}$, 4: $\{\ a, b\ \}$, 5: $\{\ a, b, c\ \}$. Paths 3 and 5 have the maximal output set $\{\ a, b, c\ \}$ and are the only ones retained in the dependence graph; paths 1, 2, and 4 are deleted. If there is more than one distinct maximal set, any one maximal set is chosen at random. This technique of deleting paths not having the maximal output set is not implemented in the dependence graph generation of the main module where all paths need not have the same set of computed variables. The reason for imposing this condition in a constraint module 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 corresponding to a constraint module it is not possible to determine statically the actual parameters and global variables computed in the constraint module call, which have to be added to K.

If the dependence graph generation is successful, a new set of constraints 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 parameters corresponding to the set of computed formal parameters in the maximal output set. An attempt is made to resolve this set of constraints with $Z_1 \ldots Z_p$ in the known set K. If the constraints in this set are resolved as computation for all the unknowns in $e_{k1} \ldots e_{kp}$,

a call node which invokes the dependence graph for the constraint module call G_{mod} is generated as shown in Figure 4.11. A child node of the call node receives values computed for the formal parameters by the call node and binds them to $Z_1 \ldots Z_p$ and performs the computation generated from the new set of constraints.

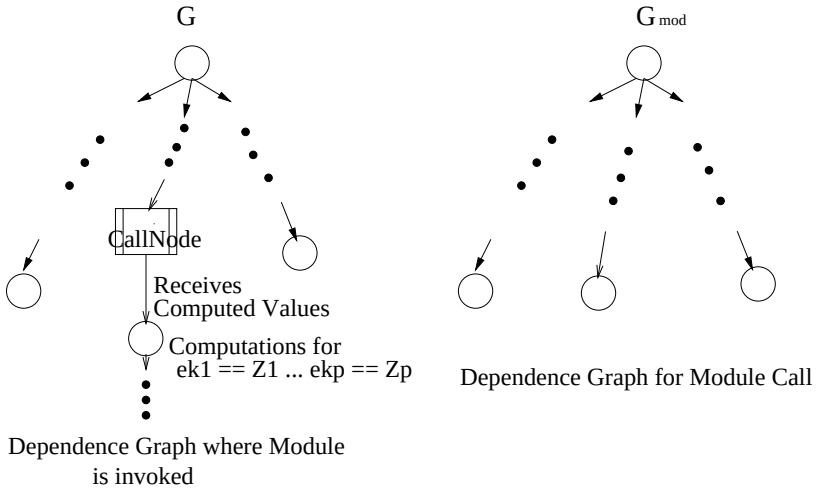


Figure 4.11: Dependence Graphs for a Constraint Module Call

If the dependence graph generation is not successful, the constraint module call is considered to be unresolved.

For a constraint module with n parameters there are 2^n possible input parameter sets and consequently, there are 2^n potential translations for a particular constraint module. Of course, not all translations might be successful. Constraint module invocations, with the same set of formal parameters and global variables as inputs, reuse the same dependence graph.

4.3.4 The Quadratic Equation Solver through Phase 3

The dependence graphs for the quadratic equation solver with the input set $\{ a, b, c \}$ are shown in Figure 4.12 where computations for $r1$ and $r2$ are extracted.

The dependence graphs for the quadratic equation solver with a different

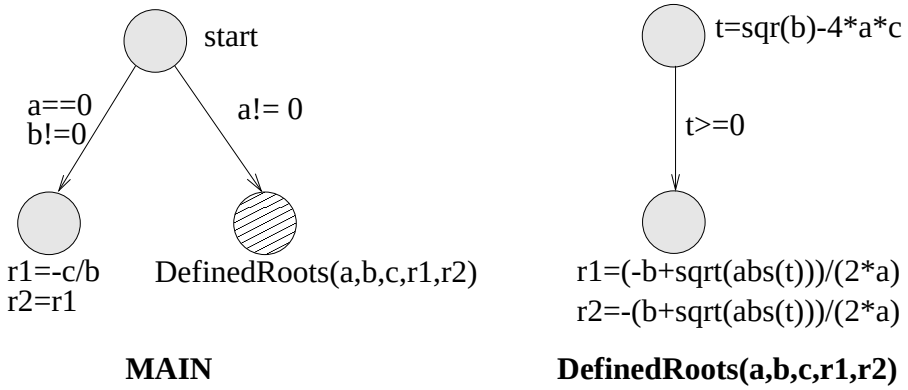


Figure 4.12: Dependence Graphs for the Quadratic Equation Solver with $\mathcal{I}$ = {a, b, c}

input set $\{\ a, b, r1\ \}$ are shown in Figure 4.13. 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.

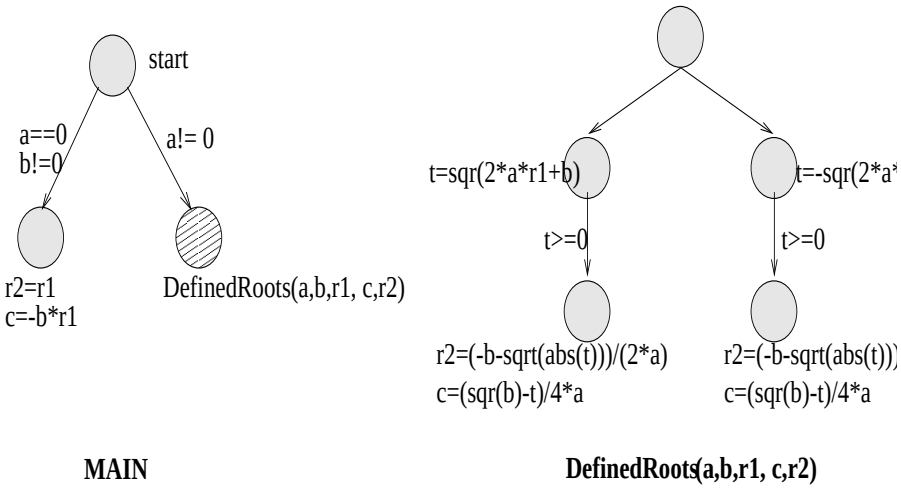


Figure 4.13: Dependence Graphs for the Quadratic Equation Solver with $\mathcal{I}$ = {a, b, r1}

Figures 4.12 and 4.13 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 dependence graphs where no constraints remain unresolved. For example, no dependence graph can be generated with the input set $\{ a, r1 \}$ because the simple constraint $b * r1 + c == 0$ and the constraint module call $DefinedRoots$ remain unresolved in the *main tree* (The module call $DefinedRoots$ remains unresolved because all the constraints in the tree for the module remain unresolved.). Note that phases 1 and 2 need not be repeated when a new input set is supplied for a constraint specification.

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.

4.3.5 Single Assignment Variable Programs

The compilation process generates dependence graphs with single assignment variables. This occurs because a child node in the dependence graph G inherits the known set of its parent as its initial known set and no deletions are made to the known set of a node. Hence, once a variable is added to the known set of a node it is retained in the known sets of all nodes in the subgraph rooted at that node. If a path in G contains nodes in the order ${v_1}^* \ldots {v_n}^*$, where ${v_1}^*$ is the start node, ${v_n}^*$ is the leaf and n is the length of the path, exactly one node ${v_i}^*$, $1 \leq i \leq n$, can contain a computation for a variable x. There will be no occurrence of x in any computation or firing/routing rule for nodes in ${v_1}^* \ldots {v_{i-1}}^*$ or in any firing rule for ${v_i}^*$. While single assignment variables are appropriate for parallel programs, they can lead to excessive use of memory in some circumstances. Chapter 5 details our

approach to introduction of mutable variables where they are necessary.

4.3.6 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 directed 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 4.3.7. Non-determinism arises if the compiler randomly chooses a path for execution in effective programs.

4.3.7 Extraction of parallelism

Our constraint representation maps to a dependence graph which is a parallel computation structure because all nodes that are enabled for execution may be executed in parallel. The constraint representation allows the targeting all types of parallelism (AND/OR, task and data parallelism) through a single representation. AND/OR parallelism refers to parallelism in computations extracted from terms connected by AND and OR operators, respectively. Task parallelism refers to parallelism in computations for different data. Data parallelism refers to parallelism in computations for different parts of a structured data item. In our system 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.

The different sources of parallelism and their respective types in the representation were detailed in [JB96b] and are enumerated as follows. While 1-4 are extracted by the current compiler, 5 has not yet been implemented.

1. **OR, Task**: OR parallelism corresponds to executing the different paths in the dependence graph in parallel. These paths have resulted from the extraction of computation from constraints connected by OR operators.

2. **AND, Task**: The computational statements that are assigned to a node have the potential for parallel execution. For instance, the assignments $r1 = (-b + r)/2 * a$ and $r2 = -(b + r)/2 * a$ in Figure 4.12 can be done in parallel. 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. A particular node may be split into several nodes to exploit the parallelism in the computations at the node. The granularity of such a scheme depends on the complexity of the functions and the operators invoked in the statements.

We illustrate AND-OR parallelism in 1 and 2 through a simple example. Consider the constraint specification in Figure 4.14 for a program involving variables { a,b,c,x,y }.

$$\boxed{\begin{array}{l} a<b \ AND \ b==x \ AND \ y==c \\ OR \\ a<c \ AND \ x==c \ AND \ b==y \end{array}}$$

Figure 4.14: Constraint Specification for a Simple Example

The dependence graph for the input set { a,b,c } and output set { x,y } for the specification in Figure 4.14 is shown in Figure 4.15. 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 4.3.6. AND parallelism is extracted from the computations for x and y.

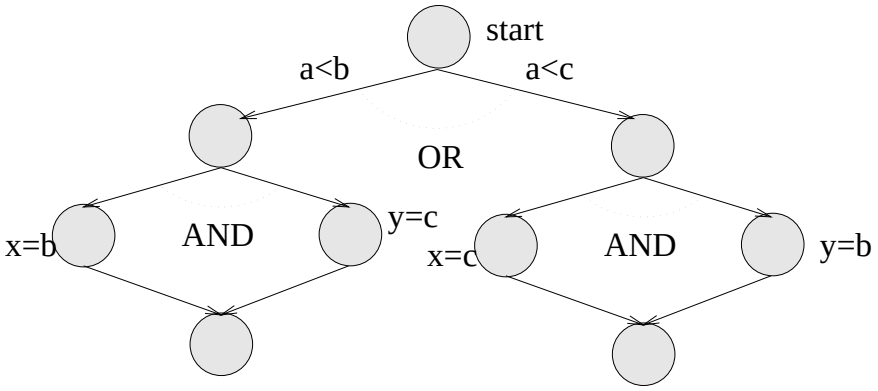


Figure 4.15: Dependence Graph showing AND-OR Parallelism

3. **Task**: 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 $N \times N$). In a later version of the compiler, provision will be made for user specification of parallelism for operations over structures.

4. **Data (Parallelism in AND indexed sets)**: The computations within the compiled loop structures corresponding to *AND* indexed sets have the potential for parallel execution. We first discuss the case of loops with a single computation. The discussion is then 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. The compiled parallel structure for such a loop is shown in Figure 4.16(a). The node performing the computation and the arc connecting the parent to it are replicated N times, where N is the range of the loop index. The results of the computation performed by the parallel nodes are merged (not shown in figure).

(ii) If the array 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. The compiled structure is again as shown in Figure 4.16(a).

(iii) If cases (i) and (ii) do not hold, the loop iterations are inter-dependent and are executed sequentially. The compiled structure for this case is shown in Figure 4.16(b). The node performing the computation is invoked repeatedly in succession.

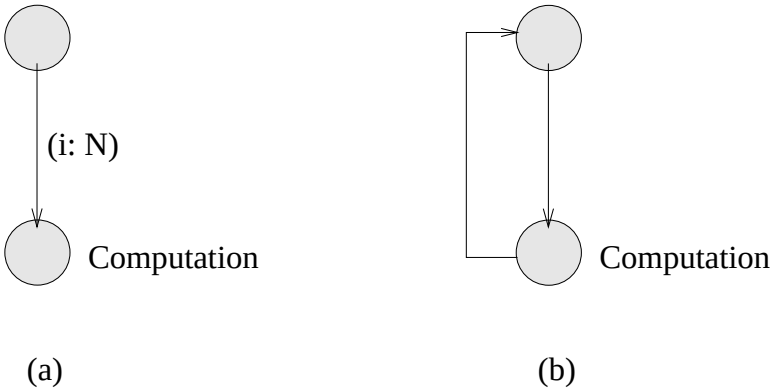


Figure 4.16: (a) Parallel Execution of Loop (b) Sequential Execution of Loop

A similar analysis is done for the loop structure compiled from an indexed set with more than one constraint. In such a case there is more than one com-

putation within the loop and interdependencies between different computations for all the iterations have to be checked in addition to dependencies between iterations of the same computation. If there are no dependencies between the iterations of a computation (cases (i) and (ii)) and no iteration of the computation is dependent on an iteration of another computation, then all iterations of the computation are executed in parallel; otherwise, the iterations of the computation are executed sequentially. In general, the loop structure will be a combination of parallel and sequential loop executions as shown in Figure 4.17.

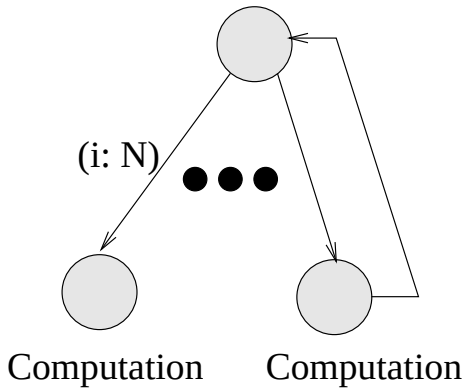


Figure 4.17: Generalized Compiled Loop Structure

5. **Data**: Finally, primitive operations in the base types like matrix-matrix multiply can be executed in parallel by invoking appropriate parallel algorithms.

4.3.8 Unresolved Constraints

Any path P from the root to a leaf in the tree T from phase 2 consists of nodes, each containing a set of constraints. P represents one way of satisfying the constraint system since constraints on different paths are implicitly connected by the OR operator. Every constraint on P must be resolved to either a computation or to a conditional (firing/routing rule) for P to satisfy the constraint system. The depth-first traversal described in Section 4.3 collects any unresolved constraint on

P at its leaf. An unresolved constraint can be of the following types.

(i) A simple constraint involving an equality and at least two unknowns.

(ii) A simple constraint involving a relational operator other than an equality and at least one unknown.

(iii) An unresolved call to a constraint module. This would imply that there is more than one unknown in the set of actual parameters, local variables, and global variables in the body of the constraint module. (Unknowns in an actual parameter imply that the corresponding formal parameter is unknown.)

(iv) An unresolved indexed set of constraints AND/OR FOR (<index> <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$ where each A_i, $1 \leq i \leq n$, is unresolved due to one of the following reasons.

(a) A_i may be an unresolved indexed set.

(b) If A_i is a simple constraint or a constraint module call, there is no unique unknown term for all values of i in $b1 \ldots b2$ (See Section 4.3.2).

(c) During the resolution process A_i is classified as a computation for some values of i in $b1 \ldots b2$ and as a conditional (firing/routing rule) for other values of i in $b1 \ldots b2$.

In case (c) we may be able to split the indexed set into several resolved indexed sets with different index bounds. Assume that

$S_1 \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as computations and conditionals in the ranges $B_{s_1(1)}, B_{s_1(2)}, \ldots, B_{s_1(p_1)}$,

$S_2 \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as computations and conditionals in the ranges $B_{s_2(1)}, B_{s_2(2)}, \ldots, B_{s_2(p_2)}$,

$\vdots$

and $S_q \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as computations and conditionals in the ranges $B_{s_q(1)}, B_{s_q(2)}, \ldots, B_{s_q(p_q)}$,

where B_i, $s_j(1) \leq i \leq s_j(p_j)$, $1 \leq j \leq q$, is a subrange in $b1 \ldots b2$,

$S_i \cap S_j ==$ the null set, ϕ, $1 \leq i, j \leq q$, $i \neq j$, and

$S_1 \cup S_2 \cup \ldots \cup S_q == \{A_1, A_2, \ldots, A_n\}$.

The indexed set can be split into the following resolved indexed sets.

AND/OR FOR $(i < B_{s_1(1)} >)$ S_1

AND/OR FOR $(i < B_{s_1(2)} >)$ S_1

$\vdots$

AND/OR FOR $(i < B_{s_1(p_1)} >)$ S_1

AND/OR FOR $(i < B_{s_2(1)} >)$ S_2

AND/OR FOR $(i < B_{s_2(2)} >)$ S_2

$\vdots$

AND/OR FOR $(i < B_{s_2(p_2)} >)$ S_2

$\vdots$

AND/OR FOR $(i < B_{s_q(p_q)} >)$ S_q

There are several options available for resolution of each type of unresolved constraint. We shall enumerate some of these.

1. Since there is a 1-1 mapping between nodes in T and the dependence graph G, there is a unique leaf in G corresponding to the leaf in P containing the unresolved constraints. In Figure 4.18 the two corresponding leaves in T and G are shaded. The shaded leaf in G can be deleted. This virtually removes P from T and corresponds to not attempting to satisfy the constraint system through the path P. If deletion of the leaf in G results in its parent becoming a leaf, the parent must be deleted too. This must be continued in a recursive fashion until the deletion of a leaf does not result in its parent becoming a leaf. Then a new path descending T is chosen and pursued to see if a usable G can be obtained. This approach can be applied to unresolved constraints of any type ((i)-(iv)). Of course, there is the danger of getting an empty dependence graph if all leaves in T contain unresolved constraints.

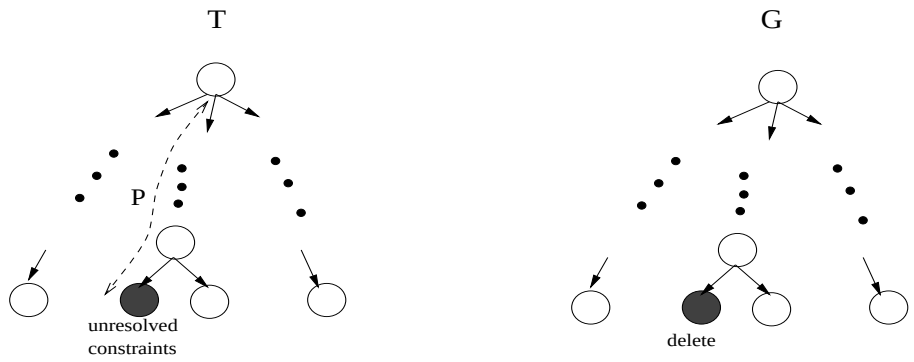


Figure 4.18: Deletion of a Path with Unresolved Constraints

2. The user may have incorrectly specified the initial input set by overlooking the inclusion of some variables or including the wrong variables and may be helped in the choice of a new input set through display of the unresolved constraints and the unknowns in them. The depth-first traversal in phase 3 can be performed again with the new input set. This can be done repeatedly until all constraints in the system are resolved. While selection of unknowns to be added to the initial input set may be easy for constraints of types (i), (ii) and (iii), it may be quite difficult to do for constraints of type (iv) since unknowns could typically be of the form $A[fn_1(i_1, \ldots, i_k)] \ldots [fn_l(i_1, \ldots, i_k)]$ where $i_1 \ldots i_k$ are indices for nested indexed sets containing the unresolved constraints, $fn_1 \ldots fn_l$ are arithmetic functions over the indices, and A is any structured data type. Some parts of A may be known and other parts of A may be unknown forcing the user to identify the regions that the term $A[fn_1(i_1, \ldots, i_k)] \ldots [fn_l(i_1, \ldots, i_k)]$ refers to and denote them as known.

3. Commercial solvers such as MATLAB can be invoked to solve the unresolved constraints by providing a wrapper around the invocation to the MATLAB solver in the form of a constraint module call. This technique can be most beneficial for the resolution of constraints of types (i) and (ii).

4. Iterative solutions can be attempted for unresolved constraints of types (i), (iii) and (iv) through several relaxation methods. This process is described in

detail in Chapter 5.

4.4 Phases 4 and 5: Specification of Execution Environment and Mapping to Code

Apart from the textual constraint program the programmer is encouraged to specify an execution environment specification which is used by the compiler to optimally select certain execution environment characteristics used by CODE [NB92] to generate programs. The execution environment specification merits a separate discussion and is described in Chapter 6.

Our target for executable for constraint programs is the CODE 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 4.7. 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 (see Appendix A for a description of CODE). 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.

The control flow for the entire compiler is shown in Figure 4.19.

4.5 Procedural Parallel Programs for the BTS and BOER Systems

In this section we show how all of the parallelism in the BTS (Figure 3.4) and BOER (Figure 3.8) examples can be extracted by the compiler.

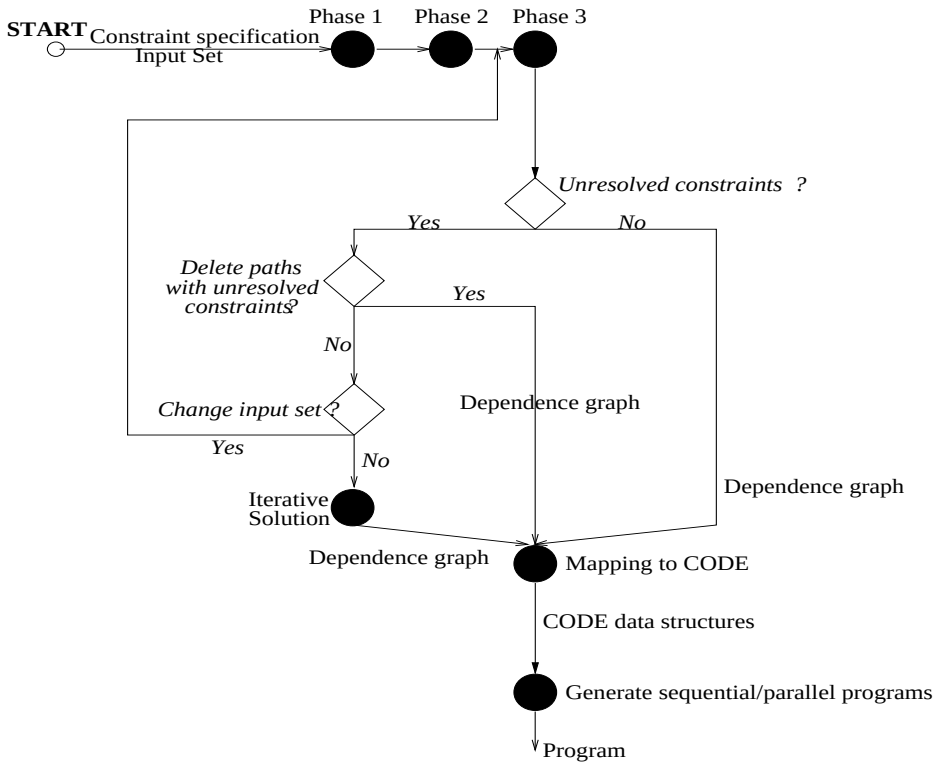


Figure 4.19: Control Flow for the Constraint Compiler

4.5.1 The BTS System

Consider the specification for the BTS system being compiled with the input set { $S_0, \ldots, S_3, M_{10}, M_{20}, \ldots, M_{32}, B_0, \ldots B_3$ }. The specification has been repeated in Figure 4.20 with certain terms ($\{X_0, X_1, X_2, X_3\}$) in bold-faced to indicate terms that are chosen as outputs by the compiler.

By applying technique 3 in Section 4.3.7 the compiler splits up the specifications to perform the multiplications in series such as $M_{10} * X_0$, $M_{20} * X_0$, and $M_{30} * X_0$ in parallel. Thus the vector multiplications for all Ms within a column may be done in parallel. Figure 4.21 shows the form of the extracted dataflow and exactly corresponds to the parallel algorithm in [DS86]. Data parallelism could be used on the block level operations and captured in our representation with an

$$
\begin{array}{l}
\textit{PROGRAM BTS\_1} \\
\\
(\ S_0 * \mathbf{X_0} == B_0\ \ AND \\
M_{10} * X_0 + S_1 * \mathbf{X_1} == B_1\ \ AND \\
M_{20} * X_0 + M_{21} * X_1 + S_2 * \mathbf{X_2} == B_2\ \ AND \\
M_{30} * X_0 + M_{31} * X_1 + M_{32} * X_2 + S_3 * \mathbf{X_3} == B_3\ )
\end{array}
$$

Figure 4.20: Constraint Specification for the BTS System with Computed Terms in bold

appropriate type structure, if desired.

4.5.2 The BOER System

The constraint specification for the BOER system has been repeated in Figure 4.22 with certain terms in bold-faced to indicate computed terms. Each indexed set is compiled to a loop iterating over values of the index. Each simple constraint is compiled to a computation for a term (bold in Figure 4.22).

Analysis of the computations extracted shows that, in the reduction phase, the computations for $BP[j], CP[j]$, and $dP[i * pow(2,j)][j]$ can be executed in parallel. However, different iterations of the loop enclosing these computations (for index j) cannot be done in parallel due to interdependencies between the three computations. The different iterations of the nested loop for index i enclosing the computation for $dP[i * pow(2,j)][j]$ can be performed in parallel. The nested loop for index i in the back-substitution phase enclosing the computation for $x[\ldots]$ can be performed in parallel. However, the iterations for the outer loop for index j enclosing the computation for $x[\ldots]$ cannot be parallelized. Our compiler detects all the dependencies for this analysis and correctly extracts all the existing parallelism in the specification.

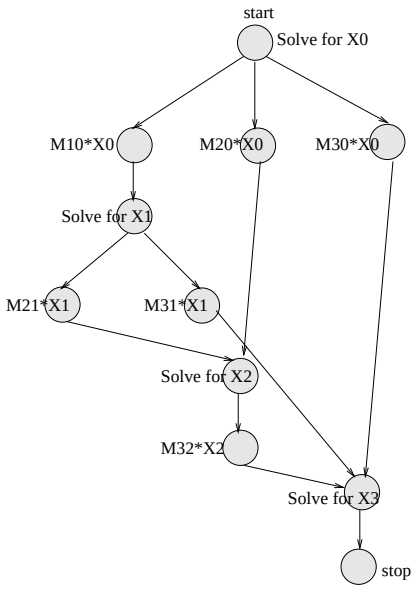


Figure 4.21: Dependence Graph for the BTS Program

The resulting dependence graph is shown in Figure 4.23 and exactly corresponds to the dataflow in the algorithm in [LD90]. 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 reduction and back-substitution phases in the constraint specification. The annotation "Replicated" on the arcs specify that the annotated arc and the destination node (shaded in Figure 4.23) 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 and are instances of data parallelism. The nodes annotated by *BP, CP, dP*, and x compute values for parts of the corresponding variable. The parallel execution of the computations for *BP, CP*, and *dP* is an instance of task parallelism. 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 extracting the parallelism within the expression $2 * CP[j-1] * CP[j-1] - BP[j-1] * BP[j-1]$ in the computation for $BP[j]$ by computing the products $2 * CP[j-1] * CP[j-1]$ and $BP[j-1] * BP[j-1]$ in parallel. By incorporating calls to BLAS routines (technique 5 in Section 4.3.7) which invoke parallel algorithms, incorporating data parallelism, for matrix-matrix multiply the compiler would have extracted all the existent parallelism in the example.

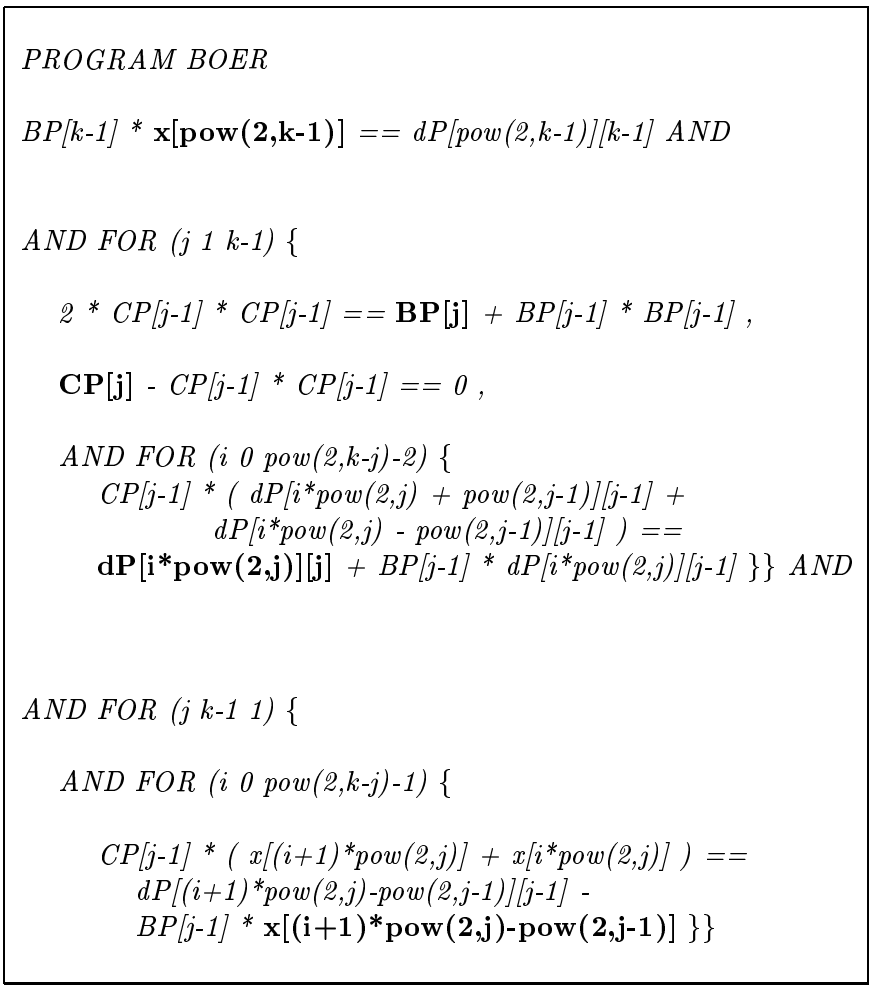


Figure 4.22: Constraint Specification of the BOER System with Computed Terms in bold

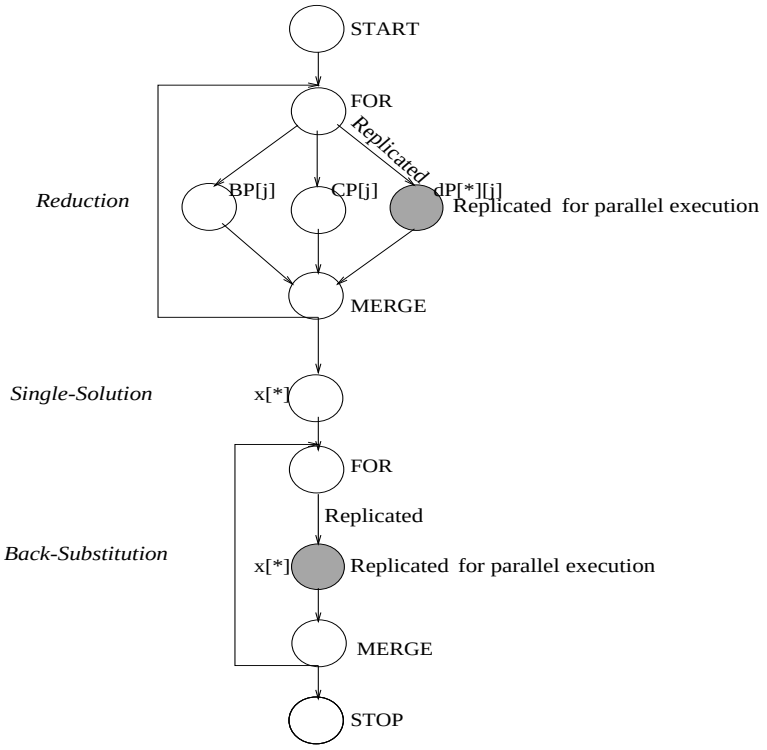


Figure 4.23: Dependence Graph for the BOER Program

Chapter 5

Iterative Solutions for Constraint Systems with Cycles

Chapter 4 detailed the basic compilation algorithm for translating a constraint specification along with an input set to a dependence graph. The basic compilation algorithm cannot resolve constraint specifications with input sets that give rise to dependencies with cycles.

To illustrate dependencies with cycles, consider the constraint program shown in Figure 5.1. Phase 2 of the compiler collects constraints connected by AND operators at the same node and since there is only a single AND operator in the specification in Figure 5.1, phase 2 will generate a single node with the two simple constraints: $a + b == x$ and $x + b == y$ (shown in Figure 5.2). When this node is traversed in phase 3 with the input set $\{ a, y \}$ both simple constraints remain unresolved because there are two unknowns b and x in each of them (simple constraints are resolved as conditionals if they have no unknowns and as computations if they involve an equality and only one unknown; otherwise they are unresolved). The term *cyclic* is used to refer to this situation because a cycle exists in the low-level constraint graph representation (introduced in Section 2.3) for this constraint

program as shown in Figure 5.3. Note that the arcs connected to the input variables a and y have directions on them to denote that the values for these variables are available. The non-input variables x and b are in a cycle and neither of the two "+" operator nodes can "fire" for computed values to be propagated along the arcs until either x or b is given a value. The constraints involved in such a situation are sometimes referred to as *cyclic constraints*. In fact, cyclic constraints give rise to *cyclic dependencies*.

PROGRAM CYCLIC_DEP1

VAR int a, b, x, y;
INPUTS a, y;

$a + b == x$ *AND* $x + b == y$

Figure 5.1: Constraint Specification and Input Set with a Cyclic Dependency

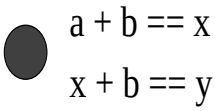


Figure 5.2: Tree from Phase 2 for Constraint Specification in Figure 5.1

This chapter discusses the augmentation [JB97] to the basic compiler for handling constraints with cyclic dependencies. We opt to use the technique of relaxation whereby iterative solutions to cyclic constraints are sought. Relaxation attempts to satisfy all the constraints in the system within a certain degree of accuracy by making an initial assignment of values to the unknowns, computing the value of one unknown in each constraint and then estimating the error in the current value. Further iterations of computing the value of the unknown variables are initiated if the errors are not sufficiently small. In each iteration, the values

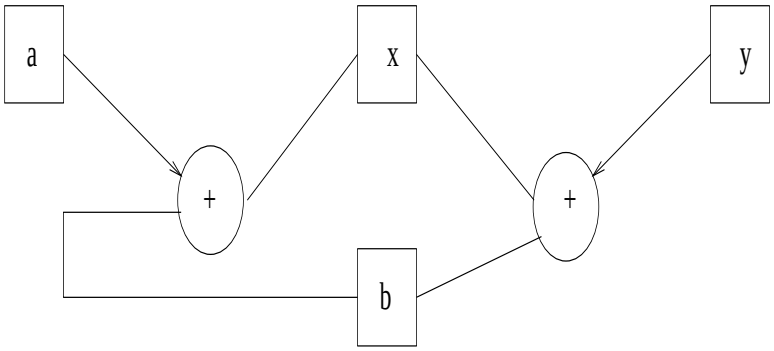


Figure 5.3: A Constraint Graph with a Cycle

computed in previous or current iterations are used to recompute the values of the unknowns in an attempt to achieve convergence where the difference in computed values in two consecutive iterations is reasonably small. The solutions extracted for the unknowns in the system are often approximate.

The class of numerical applications which can be solved through iterative methods is quite large. Many such applications are also quite amenable to parallelization. Relaxation is not, however, a universally satisfactory solution. Iterative methods may suffer from numerical stability problems. Systems using these methods might fail to terminate. Even for systems guaranteed to converge, these methods may be very slow.

A number of issues arise with respect to implementation of relaxation as an algorithm for resolution of cyclic dependencies: (i) Since there will be more than one unknown term in an unresolved constraint, how is the term to be computed selected from among all the unknown terms ? (ii) How does the compiler deal with the memory requirement for single assignment variables (Section 4.3.5) in iterative solutions ? (iii) How is the choice between the different kinds of relaxation methods (Jacobi, Gauss-Seidel etc.) made ? In the following sections in this chapter we trace the design of the compiler for iterative solutions to cyclic constraint systems.

5.1 Selection of Term to be Computed

Constraints that remain unresolved through the basic constraint compiler are collected at the leaves of the tree from phase 2 and will involve more than one unknown term (except in the case of simple constraints not involving an equality). The compiler chooses one of the unknown terms in an unresolved constraint as the term to be computed and either assigns default initial values to other unknown terms or accepts such values as inputs from the user. Unresolved constraints can be of three types: a simple constraint, a constraint module call, or an indexed set of constraints (See Section 4.3.8 for a detailed description of the causes for these constraints being unresolved). The following subsections detail the selection of the computed term for the three types of unresolved constraints.

5.1.1 Unresolved Simple Constraints

Relaxation can be attempted only for simple constraints involving an equality since other types of simple constraints must be resolved as firing/routing rules. An unresolved simple constraint involving an equality has more than one unknown variable and any such variable is randomly chosen as the term to be computed. For example, consider the unresolved constraints in Figure 5.2. There are two unknowns b and x in both the constraints. b can be chosen as the term to be computed in the first constraint $a + b == x$. Subsequently, second constraint $x + b == y$ has just one unknown x, which is chosen as the term to be computed.

5.1.2 Unresolved Constraint Module Calls

An unresolved constraint module call has more than one unknown in its set of actual parameters, local variables and global variables in the body of the constraint module. An unknown in an actual parameter implies than the corresponding formal parameter is unknown. A constraint module call could be unresolved for either of

the two following reasons.

(a) The tree from phase 2 for the constraint module call has unresolved constraints at the leaf of at least one path. This situation is shown in Figure 5.4(a) where the unresolved constraint C contains unknown variables $\{ f_1 \ldots f_p, l_1, \ldots l_q, g_1 \ldots g_r \}$, where f_i, $1 \leq i \leq p$, is a formal parameter for the constraint module, l_i, $1 \leq i \leq q$, is a local variable for the constraint module, and g_i, $1 \leq i \leq r$, is a global variable in the body of the constraint module. Depending on the structure of C (simple constraint/constraint module call/indexed set) an unknown variable will be chosen for computation and other unknown variables will be given initial values.

(b) Some subset of the set of constraints $e_{k1} == Z_1$, $e_{k2} == Z_2$, ..., $e_{kp} == Z_p$ (See Section 4.3.3 for a description of the terms and notation) to be resolved as computations for the child node of the call node in the dependence graph which invokes the constraint module remain unresolved (see Figure 5.4(b)). Again, depending on the structure of each unresolved constraint a computed variable is chosen and other unknown variables are initialized.

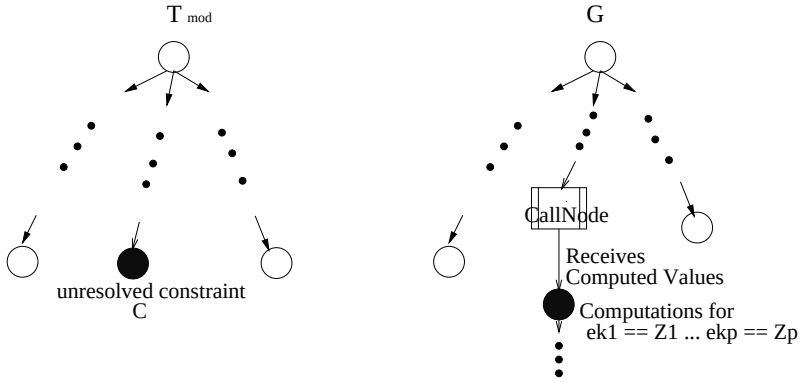


Figure 5.4: An Unresolved Constraint Module Call

5.1.3 Unresolved Indexed Sets

Relaxation cannot be used when an indexed set AND/OR FOR (i <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$ is unresolved for the following reason (See (c) in Section 4.18).

• During the resolution process a constraint A_i, $1 \leq i \leq n$, is resolved as a computation for some values of i in $b1 \ldots b2$ and as a conditional (firing/routing rule) for other values of i in $b1 \ldots b2$.

We showed in Section 4.3.8 how the compiler can generate a closed form solution in the preceding situation. If this case does not arise, the failure to resolve an indexed set of constraints can be recursively traced to "culprit" (unresolved) simple constraints and constraint module calls nested in it (A_i, $1 \leq i \leq n$).

Consider any unresolved (simple constraint/constraint module call) constraint C nested within an unresolved indexed set. A term in C is typically of the form $A[fn_1(i_1, \ldots, i_k)] \ldots [fn_l(i_1, \ldots, i_k)]$ where $i_1 \ldots i_k$ are indices for nested indexed sets containing C, $fn_1 \ldots fn_l$ are functions over the indices, and A is any structured data type. Some parts of A may be known and other parts may be unknown depending on the initial input set and the preceding computations in the current path in the dependence graph. The compiler evaluates each term in C to determine the term that accesses the largest unknown region in the structured data type. To illustrate this, consider an example constraint specification involving a $1 \times N$ array x in Figure 5.5. The end elements of A (shaded in Figure 5.6) are the inputs to the system. The values for the index i in the indexed set are in the range $2 \ldots N-1$. The constraint $x[i-1] == x[i] - x[i+1]$ remains unresolved because there is no unique unknown term for all values of i in $2 \ldots N-1$ (Reason (b) in Section 4.18). The term $x[i-1]$ accesses the region between indices $1 \ldots N-2$ in the array x, the term $x[i]$ accesses the region between indices $2 \ldots N-1$ in the array x, and the term $x[i+1]$ accesses the region between indices $3 \ldots N$ in the array x (see Figure 5.6). Hence, the term $x[i]$ accesses the largest unknown region in x, i.e.,

$x[2], x[3], \ldots, x[N-1]$ and is selected as the term to be computed in the iteration process while other terms have to be given initial values for the first iteration.

```
PROGRAM CYCLIC_DEP2

VAR x;
INPUTS x[1], x[N];

AND FOR (i 2 N-1) {
    x[i-1] == x[i] - x[i+1] }
```

Figure 5.5: Example of an Unresolved Constraint Specification

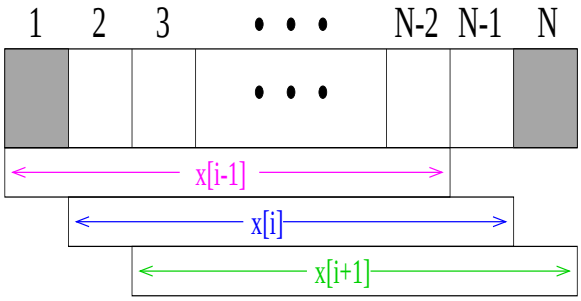


Figure 5.6: Regions of Access by Terms in Figure 5.5

The motivation behind using the heuristic of selecting the term accessing the largest unknown region as the computed term is due to the following reasons.

• Since the selected (computed) term accesses the largest unknown region, the largest number of values will be computed in each iteration of the relaxation process.

• Since the other terms access smaller unknown regions, fewer initializations will have to be done.

If the selected term does not access the entire unknown region in the data, the iterative process will not converge because certain locations in the data will not be computed. The compiler can abort the process after a fixed number of iterations,

which can be a parameter in the system. Also, if the selected term accesses a location that is an initial input to the system, convergence may not be reached because that location will be overwritten in the first iteration.

5.2 Mapping single assignment variables to mutable variables

To satisfy a constraint within some degree of accuracy, the values for the selected unknown terms have to be computed over some number of iterations t. Since the basic compilation process generates single assignment variables, iterative computations would require t memory locations for each computed term. Such a memory requirement can be quite prohibitive when the values of large data structures are being computed iteratively.

To overcome the large memory requirement for computing iterative solutions with single assignment variables, a procedure for local introduction of mutable variables is required. For each variable x being computed iteratively, the compiler may keep two locations: x and old_x. Any computed value is stored in the location x. Accessed values may come from either x or old_x, depending on the relaxation scheme being used. This will be detailed in Section 5.3. At the end of each iteration, a check is done to see if the difference between values in x and old_x is greater than the specified degree of accuracy for solution of the constraints. If it is, x is copied to old_x and further iterations are initiated. The parallel functional language SISAL employs a variant of this technique [Szy91]. In our system, the user may choose to supply a value for the degree of accuracy or accept the default value assigned by the system.

Using only single assignment variables, any computed variable with N memory locations would require $t \times N$ memory locations for t iterations. By transforming

single assignment variables to be mutable variables, the memory requirement is reduced to $2 \times N$.

5.3 Relaxation Methods

Relaxation methods such as Jacobi and Gauss-Seidel [FJL+88] can be used for iterative solutions to constraints. The Jacobi method is a stationary, iterative, method typically used for solving a partial differential equation on a numerical grid. The update of each grid point depends only on the values at neighboring grid points (defined by a *stencil*) from the previous iteration. In the Gauss-Seidel method the most recent grid values are used in performing updates. To generalize these two techniques to an iterative system, the Jacobi method can be implemented by using values from the previous iteration and the Gauss-Seidel method can be implemented by using the most recent values (some possibly from the current iteration). The Jacobi method yields more parallelism since all computations in a current iteration are independent. However, convergence is typically slower than the Gauss-Seidel method.

The user should be able to choose the method of relaxation to be used by the constraint compiler. As mentioned in Section 5.2, two locations for each computed variable x are kept: x and old_x. If the chosen method of relaxation is Jacobi, the compiler restricts all accessed values of the variable x to be retrieved from the location old_x, which stores the values of variable x computed in the previous iteration. If the chosen method of relaxation is Gauss-Seidel, the compiler restricts all accessed values of variable x to be retrieved from location x which stores the most recently computed value. The compiler currently implements only the Jacobi relaxation technique.

5.4 The Laplace Equation Example

Consider the Laplace equation for a 4-point stencil on an $N \times N$ grid indexed by $(0 \ldots N-1)(0 \ldots N-1)$ as shown in Figure 3.9. A constraint specification for the problem was presented in Figure 3.10.

The Laplace equation specification with the input set (boundary elements) constitutes a cyclic dependency. Applying the technique described in Section 5.1, x[i] will be chosen as the term to be computed since it accesses the largest unknown region, i.e., all interior elements in the grid x. The two indexed sets in the specification are compiled to loops and the simple constraint $4 * x[i][j] - x[i-1][j] - x[i+1][j] == x[i][j-1] + x[i][j+1]$ is compiled to a computation for $x[i] : x[i][j] = (x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1])/4$.

If the Jacobi method of relaxation is chosen by the user, the constraint specification can be compiled to the procedural code shown in Figure 5.7. If the Gauss-Seidel method of relaxation is chosen by the user, the constraint specification can be compiled to the procedural code shown in Figure 5.8. The user may supply initial values for the interior (non-shaded) points of the grid or choose to accept the default initial values assigned by the compiler. Variable x is initialized to the initial values and the input boundary values. Variable *old_x* is initialized such that at least one point differs in value from its corresponding point in x by more than the degree of accuracy so that the first iteration can be initiated. The function *check_accuracy(x, old_x)* returns 1 if the difference between any value in x and *old_x* is greater than the degree of accuracy; otherwise it returns 0. The function *copy_values(old_x,x)* copies values from locations in x to corresponding locations in *old_x*.

5.4.1 The Dependence Graph for the Laplace Equation

Compilation of cyclic dependencies for an iterative solution has been implemented in the constraint compiler for the Jacobi method of relaxation. The Gauss-Seidel

```
while (check_accuracy(x,old_x)) {
    copy_values(old_x,x);
    for (i 2 N-2) {
        for (j 2 N-2) {
            x[i][j] = (old_x[i-1][j] + old_x[i+1][j] + old_x[i][j-1] + old_x[i][j+1])/4
} } }
```

Figure 5.7: Jacobi Relaxation for the Laplace Equation

```
while (check_accuracy(x,old_x)) {
    copy_values(old_x,x);
    for (i 2 N-2) {
        for (j 2 N-2) {
            x[i][j] = (x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1])/4
} } }
```

Figure 5.8: Gauss-Seidel Relaxation for the Laplace Equation

method has not yet been implemented.

In the Jacobi method of relaxation, both loops (for i and j) surrounding the computation can be executed in parallel. A naive parallelization of the loops will lead to $(N-2)^2$ computation nodes, each executing an instance of the computation $x[i][j] = (old\_x[i-1][j] + old\_x[i+1][j] + old\_x[i][j-1] + old\_x[i][j+1])/4$. This is highly undesirable since the computations are too fine-grained. To overcome this, the compiler detects instances of computation extracted from constraints specified at the scalar level. Simple data partitioning techniques are applied to partition the data involved in the computation over a specified number of computation nodes P. The data partitioning techniques will be detailed in a later chapter. In the Laplace equation, the grid x is partitioned in a row-wise manner across P nodes in the extracted dependence graph. Each partitioned slice in a computation node contains locations that the node computes through each iteration and any overlapping regions

with other computation nodes that it accesses. For computations specified at the scalar level, as in this case, the region of overlap between computation nodes is determined by examining the terms in the computation. In the Laplace equation, the accessed terms are $x[i-1][j], x[i+1][j], x[i][j-1]$, and $x[i][j+1]$. The indices for the accessed terms specify a maximum displacement of 1 in the four directions of north, south, east, and west. Since x has been partitioned in a row-wise manner, the overlap is 1 row in the north and south directions. The row-wise partitioning of a 10×10 matrix across 4 nodes numbered $0 \ldots 3$ is illustrated in Figure 5.9. Each node i, $0 \leq i \leq 3$, gets rows in the range $2 * i \ldots 2 * (i+1) + 1$.

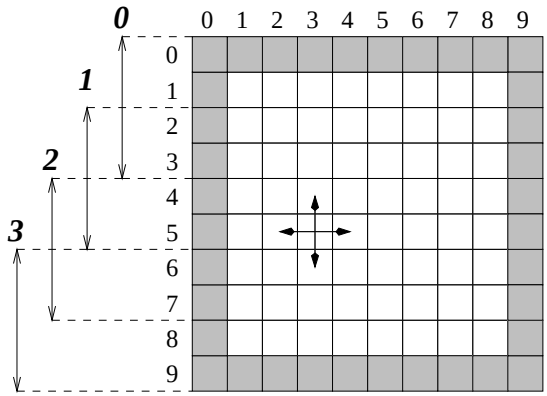


Figure 5.9: Data Partitioning for the Laplace Equation

In Figure 5.10 we show the dependence graph extracted by the compiler for a Laplace equation system executing on P nodes. The super node S initiates new iterations. The computation nodes numbered $0 \ldots P-1$ each have a slice of approximate size $\frac{N}{P} \times P + 2$ (overlap between slices) of the matrix x. In each iteration the code in Figure 5.7 is executed by each computation node on its local slice. At the end of each iteration overlapping regions are exchanged between computation nodes and the super node is informed by each computation node whether the degree of accuracy has been reached for the values in the local slice. Computation is terminated when all the nodes achieve convergence on individual slices.

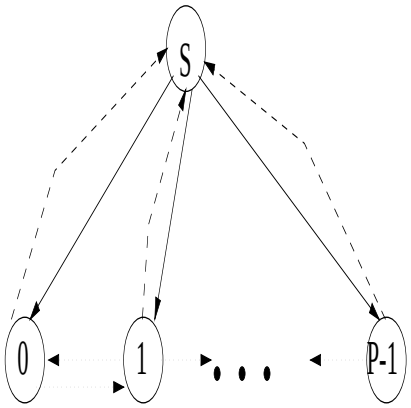


Figure 5.10: Dependence Graph for the Laplace Equation

Chapter 6

Execution Environment Specification

The advantage of using a program specification that is independent of the execution machine is portability - the ability to create executables for different architectures without changing the program specification. The constraint program specifications in our system are translated to an intermediate architecture-independent dependence graph which can be mapped to many different parallel machines. However, there are many architectural mechanisms which can be exploited by an executable program if it is directed to do so. This usually leads to an improvement in performance. Without violating the "sanctity" of our architecture-independent program specification, we propose an execution environment specification, separate from the constraint program, that allows the user to provide useful hints to the compiler about the underlying execution machine. The compiler can use these hints to produce programs that may be more optimized for performance.

This chapter discusses the design of the execution environment specification for our compiler. Several features are discussed in individual sections. While some of them have been implemented in our system, there are several others which could

be added in the future.

6.1 Shared Variables

In shared memory architectures such as the Sparc and Cray J90, a vast improvement in performance can be obtained if some variables are declared as shared because it avoids the copying of large data across computation nodes. This is demonstrated through the BTS example in a later chapter where performance results for a version of the program not using shared variables and another using shared variables are presented. The program using shared variables shows a dramatic improvement in performance over the one not using shared variables.

The user has to be cautious when declaring shared variables in a program containing constraints connected by OR operators. OR operators translate to multiple paths in the dependence graph and hence, give rise to the potential for multiple solutions. In a program not using shared variables, each path can compute a solution independent of other paths. However, a path in the dependence graph for a program using shared variables may overwrite the value computed for a variable in another path. To illustrate this, consider the dependence graph in Figure 6.1 where both paths emanating from the start node will be executed if $c > 0$. If a and b are shared variables instead of being local to each node, only one solution for a and b will be finally retained and it could be one of $\{a = 10, b = 20\}$, $\{a = 10, b = 0\}$, $\{a = 5, b = 20\}$, and $\{a = 5, b = 0\}$ depending on the interleaving of computations in a parallel environment. However, since nodes in CODE lock shared variables when execution is started and the locks are released only after the entire computation is completed, only one of the two solutions - $\{a = 10, b = 20\}$ and $\{a = 5, b = 0\}$- is possible in our system.

Hence, the user should not declare variables as shared if there is the potential for multiple solutions for them, which can be determined from the constraint

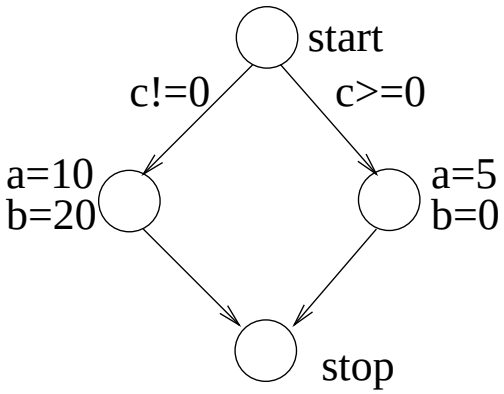


Figure 6.1: A Dependence Graph with Multiple Solutions

specification by inspection.

6.2 Number of Available Processors

This piece of information can be used to determine the number of nodes to be created when spawning off a computation to be executed in parallel. For example, the N iterations in a loop structure can be partitioned across P processors such that each processor gets approximately $\frac{N}{P}$ iterations to execute or the data computed within a loop structure can be partitioned equally across P processors. Since CODE allows the dynamic creation of nodes, the number of processors can determine the number of computation nodes at runtime.

6.3 Data Partitioning with Overlap Sections

It has been amply demonstrated by many parallel programming experiments that data partitioning techniques play a significant role in improving performance. While, currently, we have only implemented simple mechanisms for data partitioning, we show in this section that other sophisticated mechanisms can be specified too.

In many applications such as the Laplace equation, the computations are

specified at a very fine granularity, say, at the scalar level. When the compiler detects that the operations involved in the computations are over scalar types or over small-sized data (the threshold size is fixed by a parameter to the system), it partitions the variables involved over a number of nodes. This is especially important if the computation is nested within loops because the computation is executed repeatedly and the overhead in executing scalar operations repeatedly can severely degrade performance.

The form of the partition depends on the data accesses in the computation. For any matrix, if the accesses are only in the north and south directions the data is partitioned column-wise. If the accesses are only in the east and west directions the data is partitioned row-wise. If there are accesses in mixed directions, say north and east, the data is partitioned such that there is minimum overlap between the partitioned slices. This scheme minimizes the overhead in the synchronizations necessary when data is shared across computation nodes. Each node gets approximately $\frac{N \times M}{P} + overlap$, where the matrix being partitioned is of size $N \times M$ and P is the number of nodes. The amount of overlap between partitioned slices must be determined by the user or by the compiler by inspecting the terms in the computation. The mechanism of partitioning data involved in scalar computations has been used for the Laplace equation.

The user may specify the partitioning mechanism, instead of allowing the compiler to select it, by indicating the actual regions in the data type to be distributed across the nodes. (The user must specify the actual overlap between the partitions to determine the regions to be synchronized.)

6.4 Option of not Parallelizing a Module

A constraint module may have very fine-grained operations in the constraints for the constraint module body. Parallelizing such a module may lead to degradation in

performance due to the overheads involved. For this reason, a user can denote that the dependence graph for a particular module call should be mapped to a sequential procedure rather than a parallel one. This feature has not yet been implemented in our compiler. However, since CODE allows the generation of sequential programs, this would be simple to incorporate.

6.5 Selecting Operations to be Executed in Parallel

Operations over structured data types are primitives in the type system. But parallel execution can be selected for these primitive operations. The complexity of some of these operations may be larger than others. An example is the matrix-matrix multiplication operation. In the interests of performance, it would be beneficial to extract such operations out of a computation to execute in parallel. For example, if there is a computation $(d_1 \triangle d_2) \bigtriangledown (d_3 \triangle d_2)$, where $\triangle$ and $\bigtriangledown$ are primitive operations, to be executed and the operation $\triangle$ is very computation-intensive, the specification can be split into two computations to be executed in parallel: $(d_1 \triangle d_2)$ and $(d_3 \triangle d_2)$. The results can then be merged and operator $\bigtriangledown$ can be applied on them. We use this technique in the BTS example where multiple matrix-matrix multiply operators in a computation are executed in parallel.

The execution environment specification provides a platform for the user to indicate that some operations be selected for extraction from a computation for subsequent parallel execution.

6.6 Choices among Parallel Algorithms to execute some of the Operations

A variety of choices exist among parallel algorithms to execute operations on data instances under a type system. The user should be able to select one among a

number of implemented algorithms in the system to execute an operation. We have not yet implemented this feature in our system.

Chapter 7

Performance Results

A prototype of the constraint compiler has been implemented in C++ using object-oriented techniques. A number of examples have also been programmed and executed on the Cray J90, SPARCcenter 2000, Enterprise 5000, Sequent Symmetry machine [Ost89], and the PVM system. The sections in this chapter present the performance results obtained for some of the examples programmed in our system. Overall, the results have been extremely satisfactory.

The execution times reported in this chapter are wall clock times. Whenever possible, timings have been taken for executions during either dedicated CPU access or when the loads on the machines were low.

7.0.1 The Block Triangular Solver (BTS)

The extracted dependence graph corresponds to the parallel algorithm in [DS86]. The parallelism yields an asymptotic (in the number of blocks) speedup of $N^2/(3N-2)$, where N is the number of blocks. Asymptotic speedup assumes zero communication and synchronization times.

Figure 7.1 gives the speedups for a 1200 $\times$ 1200 matrix on a 14-processor shared memory Sequent machine. A hand-coded parallel program was written by