Order Number 9224791

A parallelization of the constraint logic programming language 2LP

Atay, Coşkun, Ph.D.

City University of New York, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

A Parallelization of the Constraint Logic Programming Language 2LP

by

Coşkun Atay

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1992

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

April 30, 1992
Date

Ken McAloon
Professor Kenneth McAloon
Chair of Examining Committe

April 30, 1992
Date

Prof. Stanley Habib
Professor Stanley Habib
Executive Officer

Professor Carol Tretkoff
Professor David Arnow
Dr. Kim Marriott

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

A Parallelization of the Constraint Logic Programming Language 2LP

by

Coşkun Atay

Advisor : Professor Kenneth McAloon

2LP is a modeling and control language for applications in Operations Research and Artificial Intelligence developed at the Logic Based Systems Laboratory at Brooklyn College of CUNY. It is a small, compact language with C-like syntax that has been designed to run on various parallel architectures and it is the first parallel implementation of a Constraint Logic Programming (CLP) language. Its engine is based on the revised simplex method coupled with logic programming techniques. The parallel implementation preserves the sequential semantics for a very large class of programs and is especially effective in dealing with mixed integer optimization and combinatorial problems.

In order to achieve greater scalability and portability, we employ an OR-parallel model of logic programming based on recomputation using an *itinerary* which leads a processor to a point in the computation tree. This model makes the 2LP system portable across parallel platforms of different kinds. We have developed and tested different work distribution strategies which take advantage of the itinerary technique of distributing work in a parallel environment. Currently versions are running on a network of Sun workstations using C-Linda *(tuple space model)*, the Intel iPSC Hypercube *(message passing model)*, and the BBN Butterfly *(shared memory model)*. On all three architectures, the parallel

code provides significant improvement in performance over the sequential code, and the sequential code itself outperforms commercial codes on some examples.

Acknowledgements

I would like to thank my dissertation committee members, Professors Kenneth McAloon, Carol Tretkoff, David Arnow and Dr. Kim Marriott. I thank Professor Stanley Habib of the CUNY Graduate Center and Professor Aaron Tenenbaum of Brooklyn College for their assistance. My thanks first goes to my advisor, Professor Kenneth McAloon, and to Professor Carol Tretkoff. Many of the ideas presented in this thesis originated with and benefited from them. I gratefully acknowledge the moral support and valuable advice they have given to me. I also thank Professors David Arnow, James Cox and Attila Maté for many valuable discussions. Thanks are also due to Mr. Joseph Driscoll for his excellent administrative work. I acknowledge financial support from NSF grants and New York State Graduate Fellowships.

The work in this thesis was done as part of the 2LP Project at the Logic Based Systems Laboratory at Brooklyn College of CUNY. I would like to thank my colleagues at the lab, Jo, Seth, Assen, David, and Jerry for many helpful discussions and the collegial working environment. The test runs of 2LP which constitute a major part of the thesis wouldn't have been possible without the network support from Professor David Arnow and Mr. Jialing Jiang. I appreciate very much the long hours and weekends they spent for the network.

Finally, I would like to express my special thanks to my parents, my sisters, Filiz and Hatice, the Dillon family and Elizabeth for their constant encouragement and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Introduction

Logic programming refers to the use of formulas of first order predicate logic as statements of a programming language [Kowalski]. The use of logic as a programming language was first suggested by Kowalski and Colmerauer. The first and most influential logic programming language in the field has been Prolog [Colmerauer]. In logic programming there is a clean separation of semantics and control. This separation makes it possible to experiment more freely with different implementation techniques. It also facilitates parallelization of programs transparent to the user; that is, the same sequential code can be run in parallel without any modifications. In traditional von Neumann languages, however, it is difficult to exploit parallelism in such generality because of the basic state transition semantics of these languages.

1.1 Extending Prolog with Constraints

The power of Prolog-like logic programming rests on three mechanisms: (1) relational form, (2) unification, and (3) non-deterministic computation. Prolog carries out computations in the Herbrand universe (the set of program constants and terms obtained by applying program functions to the constants) on uninterpreted terms, symbols that are not attributed

any meaning. Therefore when modeling a problem, one must map the problem from its intended domain to the Herbrand universe. This can cause loss of (1) naturalness in the problem expression, and (2) efficiency of its solution. However, Logic Programming can be extended beyond unification and the Herbrand Universe [Jaffar Lassez 86]. In fact, unification is itself a constraint solving technique used to solve equations which are just one kind of constraint and the Herbrand Universe is just one particular domain of computation. Therefore one can introduce a richer computation domain than the Herbrand universe in order to handle more expressive terms than the uninterpreted Herbrand terms. This means extending unification in Prolog in order to take into account the intended interpretations given to some functional and relational symbols. Another extension is the introduction of more general constraint solving techniques because unification is used to solve equations which are just one kind of constraint. Prolog itself does not allow constraints; they are handled in a passive way through the "generate-and-test" paradigm. Various Constraint Logic Programming (CLP) languages differ basically on the choice of computation domain and their corresponding constraint solving techniques.

1.2 Constraint Logic Programming

The CLP Scheme is a framework for the formal foundations of a class of programming languages [Jaffar Lassez]. It includes both constraint solving and logic programming paradigms. Thus CLP is an extension of logic programming aimed at replacing the pattern matching mechanism of unification by a more general operation called constraint satisfaction. The general idea behind this is the use of some mathematical tools like simplex to deal with numerical constraints, and consistency checking and constraint propagation techniques to handle symbolic constraints.

Given a computation domain, a constraint expresses a relationship between some objects

of this domain. Constraints allow concise and natural representation of problems. They state relations implicitly rather than coding them explicitly in Prolog terms. The implicit representation characteristic of constraints increases the expressive power of logic programs [Jaffar Lassez].

1.3 Parallelism in Logic Programming

In classical logic programming, the execution of a program starts with an initial goal and through a series of resolutions the interpreter computes values of variables from the program clauses and the existing goals. Each computation step is a derivation of a new goal from an existing goal and a program clause. This execution can be viewed as a search tree, where multiple branches of a node corresponds to different clauses for resolving with the goal at that node. OR parallelism refers to parallel execution of the different clauses of a procedure, while AND parallelism is the parallel execution of the goals in the body of a clause. Thus their sources are nondeterminism in the choice of goals and nondeterminism in the choice of clauses respectively. From the search tree point of view, OR parallelism corresponds to parallel search of different paths, while AND parallelism corresponds to parallel execution of steps within a path. In OR parallelism any one of the paths that succeeds is a solution, however in AND parallelism all steps must succeed for a solution.

AND parallelism is usually more difficult to exploit than OR parallelism because of possible dependencies between AND-parallel goals. OR parallelism offers very good potential for coarse-grained parallelism for a wide range of applications. It exists in any search problem where there is a "generate-and-test" case. Its main problem is how to handle different bindings of a variable on different branches of the search tree.

1.4 WAM: An Abstract Prolog Interpreter

The WAM (Warren Abstract Machine) [Warren 83] is an abstract Prolog instruction set complete with its storage model for efficient execution of Prolog programs. Since many (sequential and parallel) logic programming languages are based on the WAM, a short description of its storage model will be useful before proceeding any further.

The main data areas are the *code area* containing the instructions and data, the (local) *stack*, the *heap* (global stack), the *trail*, and a push-down list (pdl) used for unification. The stacks expand with procedure calls and shrink on backtracking. The heap contains the structures and lists created by unification and procedure call. The trail contains references to variables that have been bound by unification and must be unbound on backtracking. The stack contains *environment* and *choice point* frames. An environment holds local variables and bookkeeping information. A choice point contains the information necessary (pointers to the alternative clauses and register values) to restore an earlier state on backtracking. It is created when entering a procedure only if the procedure has more than one clause (a nondeterministic procedure). A *continuation* chain links environments and a *backtrack* chain links choice points. The top frame in each is called the *current* frame. There are *state registers* to used to manage the storage areas and there is a set of *argument registers* for passing parameters during procedure calls and calculating temporary results. In addition, *tail recursion optimization* removes information from the local stack when executing the last procedure call in a determinate procedure, and the *cut* operator excises backtracking information from both the local stack and the trail.

For OR-parallel execution of Prolog programs, Warren proposes another model, the SRI model [Warren 87-2]. The SRI model is an extension of the WAM. Each worker has a *binding array* for immediate access to bindings it is working with, and the trail is modified

to contain address-value pairs instead of just addresses. When switching tasks, the workers modify their binding arrays incrementally, using the difference between the binding lists of their old node and the new node. The binding array makes the unification steps (creation and access to bindings) constant-time operations but not the task switching.

Chapter 2

CLP Systems

2.1 CLP($\mathcal{R}$) : (Jaffar, Michaylov '87)

CLP($\mathcal{R}$) [Jaffar Michaylov] is designed to be an instance of the CLP Scheme, a family of rule-based constraint programming languages defined by [Jaffar Lassez 87]. The domain of computation $\mathcal{R}$ is the algebraic structure consisting of uninterpreted functors over the real numbers. Consider the following CLP($\mathcal{R}$) program:

```
ohm(V, I, R) :-  V = I * R.
```

Here the functor is `ohm`, and is defined with a constraint over variables which can have real values. In CLP($\mathcal{R}$) constraints are treated uniformly; they are used in the input, execution and output of the program. The abstract machine of CLP($\mathcal{R}$), the CLAM (Constraint Logic Abstract Machine), is an extension of the WAM which has a simplex tableau based linear constraint solver and a delay mechanism for non-linear constraints. The CLP($\mathcal{R}$) system is organized in three main parts:

- *inference engine:* executes derivation steps, and maintains variable bindings;
- *interface:* evaluates complex arithmetic expressions and transforms constraints to a canonical (simplified) form via detection of implicit equalities, redundancy removal, etc.;

- *constraint solver:* solves constraints that are too complicated to be handled directly in the engine and interface; maintains a delay/wakeup mechanism for non-linear constraints; it is incremental in the sense that adding a constraint does not require recomputing old constraints (like simplex).

2.2 PROLOG III : (Colmerauer '87)

Prolog III [Colmerauer 87] is an extension of Prolog with the following features at the unification level:

1. refined manipulation of trees, including infinite trees, together with a specific treatment of lists,
2. complete treatment of Boolean algebra,
3. treatment of the operations of addition, subtraction, multiplication by a constant, and of the relations $<, \leq, >, \geq$,
4. general processing of the relation $\neq$.

At the heart of these extensions is the replacement of the unification concept by the concept of constraint solving in a specific domain. The domain chosen by the Prolog III is the set of (infinite) trees whose nodes can be labeled by identifiers, characters, boolean values, real numbers and lists.

The kernel of the Prolog III interpreter consists of a two-stack machine which explores the search space of the abstract machine via backtracking. The first stack contains the structures representing the states. The second stack keeps track of all the modifications made on the first stack and for this reason address-value pairs are used to restore the previous states upon backtracking.

2.3 CHIP : (ECRC group '88)

CHIP (Constraint Handling In Prolog) [Dincbas *et al.*] is a Prolog-like logic programming language extended by symbolic and numerical constraint solving techniques. The new techniques and domains introduced are consistency techniques for finite domains, equation solving in Boolean algebra and a symbolic simplex-like algorithm for rationals. The system uses sophisticated lookahead techniques and graph-theoretic constraint satisfaction techniques [Hentenryck].

2.4 CAL : (Sakai, Aiba '89)

This system was developed at the ICOT Lab and is the only system for constraint programming over the Complex Numbers. The constraint solving engine is based on the Buchberger Gröbner Basis Algorithm [Sakai Aiba].

2.5 2LP : (McAloon, Tretkoff '89)

2LP (Logic Programming and Linear Programming) [McAloon Tretkoff] is a Constraint Logic Programming language with a C-like syntax. It is designed to solve Mixed Integer Programming (MIP) problems and AI problems using Operations Research and Logic Programming techniques. Technically, the domain for this CLP language is the ordered group of Rational Numbers as a **Z** module [Cox McAloon Tretkoff]. Since the logic of 2LP is relational, the problems can be formulated easily and naturally, and linear programming is a subset of 2LP. The declarative power of logic programming makes it easier to formulate constrained optimization problems such as MIP problems where the logic can be used instead of 0-1 variables. (The reader is referred to Appendix A for an in depth coverage of Linear and Mixed Integer Programming in 2LP.)

2LP introduces the following novel ideas:

- *Witness Programming:* allows the programmer to communicate with the incumbent or current solution. At termination, the witness point is the output of the computation. This replaces the Prolog or CLP($\mathcal{R}$) "answer substitution."

- *Compact Constraint Logic Programming:* is the family of CLP languages obtained by varying the logic and the domain that can be custom designed for intended domains of applications. Its advantages are feasible witness programming, efficient compilation and easy parallelization.

- *S-CLAM:* (Subcompact Constraint Logic Abstract Machine for 2LP) is the first abstract machine for a compact constraint logic programming language; it supports witness programming; shadows the WAM architecture in its simplicity; and affords a tight integration between the logic and the state of the simplex-based constraint solver.

In this thesis we will introduce a new element which is the key tool in implementing a portable parallel version of 2LP:

- *Itinerary-based Parallel Scheduling Strategies:* are a set of work distribution strategies. In order to achieve greater scalability and portability, a model of scheduling based on *itineraries* has been developed and tested. An itinerary is a road map that leads a processor to a node in the computation tree of a problem.

Because of its resemblance to C [Kernighan Ritchie], the syntax of 2LP is easy to describe. It has variables and arrays of types `int` and `double`. For the purpose of constrained optimization, a new type `continuous` has been introduced to represent the decision variables of a problem. This new type has the distinction of being defined as an

interval. At the beginning its interval is $[0, +\infty)$ and at each step toward the solution as the constraints are added, the interval gets smaller; at the optimum it is reduced to a point which represents the solution value of the variable. Consider the following declarations:

```
double      d[10];
int         i, j, i_array[20];
continuous  x[2], y;
```

The declarations define variables and arrays of type `double`, `int` and `continuous`. The `continuous` variables define a polyhedron of dimension 3. As the program progresses, constraints on the `continuous` variables are generated and will define an evolving polyhedron in 3 dimensional space. At the beginning of the program, the polyhedron is the positive orthant (Figure 2.1(a)). If the constraint `x[0] + x[1] + y <= 1;` is generated, then the polyhedron becomes the triangular solid of Figure 2.1(b). Later, adding the constraint `x[0] == x[1];` to those that define the polyhedron of Figure 2.1(b) restricts the polyhedron to the two dimensional polytope in Figure 2.1(c).

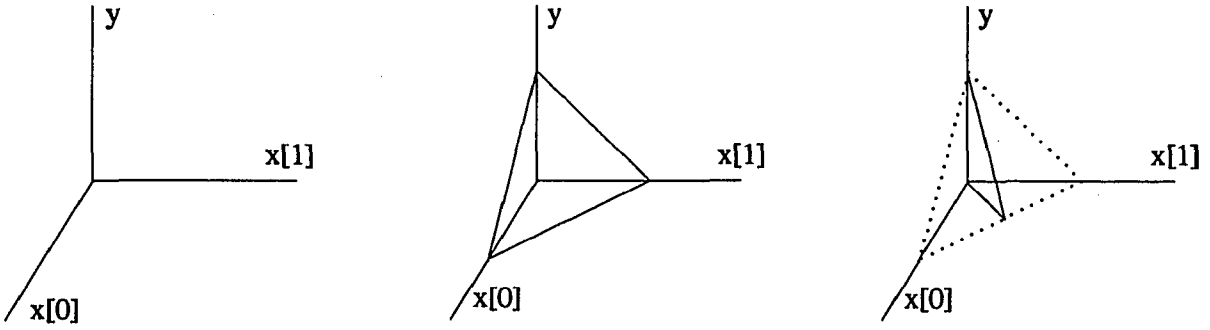

Figure 2.1: (a) The positive orthant, (b) the triangular solid defined by the constraints, and (c) its projection into a triangle in 3-space

While the fundamental operation on variables of type `double` and `int` is assignment (=), the `continuous` variables are constrained by linear equalities (==) and inequalities (<=, -

>=). The language also has and-loops similar to the for-loops of C to facilitate representing group of related constraints, and sigma notation for shorthand representation of summation as in algebra. The following are examples of these declarative constructs of 2LP. The constraint

$$\sum_{i=1}^{N} a_i x_i \leq K$$

can be written as

```
sigma(int i=1;i<=N;i++) a[i] * x[i] <= K;
```

while the group of constraints

$$\sum_{i=1}^{N} a_{ij} x_{ij} \leq K, \quad j = 1, \ldots, M$$

can be written as

```
and(int j=1;j<=M;j++)
    sigma(int i=1;i<=N;i++) a[i][j] * x[i][j] <= K;
```

The 2LP abstract machine is called the S-CLAM for Subcompact Constraint Logic Abstract Machine. The S-CLAM is analogous to the WAM [Warren 83] of Prolog and the CLAM of CLP($\mathcal{R}$) [Jaffar Michaylov].

Since 2LP eliminates structured terms, in the S-CLAM, a simple form of unification of classical logic programming is combined with constraint solving using the revised simplex algorithm. The heap is replaced by a matrix of constraints. The trail stack keeps track of changes in the upper and lower bounds on the continuous variables. The procedure stack contains the frames as in the case of the WAM with a slight difference to accommodate for the constraints. In 2LP, the nondeterminism of logic programming can be easily transformed into OR-parallelism. Because 2LP does not have structured

terms, dereferencing is trivial. There are no binding arrays in the heap. With only a small amount of data transfer, potentially long running processes can be forked from the top-most choice point. In the constraint optimization problems, parallel processes are not speculative because they are required in order to insure optimality or near-optimality. Also, these threads communicate the current best solution to each other in order to prune the search space.

Chapter 3

Current OR-Parallel Systems

The Muse [Ali Karlsson] approach to an OR-parallel Prolog system is based on having multiple sequential engines which are minimal extensions to the WAM. Each has its own local address space, and some shared memory space. The major operation in the Muse model is copying engine states for task switching. A working processor copies its state and information describing unprocessed alternatives from its local memory to the shared memory, so an idle processor can have access to it. Also, because the Muse workers process the shared nodes using the built-in backtracking mechanism of Prolog, an idle worker begins work at the bottommost (most recent) available choice point, although the previous choice points are also passed to the idle worker in the shareable frame.

The Aurora [Lusk *et al.*] system is OR-parallel implementation of full Prolog for shared memory multiprocessors. All of its three schedulers (Manchester, Argonne and Wavefront) release work from the bottommost (most recent) node on a branch. The schedulers attempt to maintain one, live, shareable node on their current branch, irrespective of whether any other worker is idle. Also, dividing the search tree into public and private parts requires some locking mechanisms for the nodes in the public part which in turn introduces overhead when a worker switches tasks.

In 2LP, we have none of these restrictions on work distribution. Our work distribution

does not involve copying any part of engine states from a working process to an idle process. A 2LP worker gives away work only when there is an idle worker waiting. The itinerary technique makes task switching much simpler and makes it natural for a processor to obtain work at the topmost (earliest) choice point or anywhere else for that matter. The advantage of the topmost choice point is that a small amount of data needs to be transferred to generate a potentially long-running process. A potential drawback with the itinerary technique is the time spent on recomputing; however the empirical results have shown that this is negligible, and also using shared itinerary (Section 4.1) we reduce this recomputation time dramatically.

The REDUCE-OR process model [Kalé] is a method for parallel execution of Prolog programs. It uses REDUCE-OR trees instead of AND-OR trees to take better advantage of AND and OR parallelism. The process model is derived from the tree representation by an efficient bookkeeping mechanism for Prolog substitutions. The model is efficient but it does not fit in with the itinerary mechanism of 2LP. Also AND parallelism is discarded in our current setting, and we mainly deal with constraints as opposed to unification of terms as in Prolog.

Parallel depth-first and parallel iterative-deeping A* (IDA*) [Korf] [Kumar Ramesh Rao] [Kumar Rao] do not apply to our model, because these methods are for straightforward AI tree search problems in which workers look for the goal nodes and when one is found the search is completed. In 2LP, we do optimization and parallel runs are not speculative in optimization problems where all branches of the tree must be accounted for in order to insure optimality or near-optimality. That is, in such problems the goal is analogous to *find_all* in Prolog and the separate threads must be accounted for explicitly or implicitly in any case. Moreover, these threads communicate information on the current best solution and collaborate to prune search space.

3.1 The Muse Model of OR-Parallel Prolog

Muse (Multi-Sequential Prolog Engines) has been developed and implemented at the Swedish Institute of Computer Science, SICS. Its goal is OR-Parallel execution of Prolog programs by having several sequential Prolog engines running in parallel, each with its local address space, and some shared memory space. The sequential SICStus Prolog engine has been adapted for this OR-parallel implementation.

The Muse execution model depends on the following assumptions:

- A number of processors (workers) with identical local address space, and some global shared address space.
- Each processor is a sequential Prolog engine with its own local stacks: choicepoint stack, environment stack, term stack, and trail.

The Prolog program which is either stored in the shared space or in each worker's space is executed as follows:

1. One worker P processes the top level query creating all data structures in its own stacks. The other workers are idle until P creates local nodes (choicepoints).
2. One of the idle workers Q interrupts P requesting work.
3. P allows Q to get a piece of work by sharing its local nodes with Q as follows: (a) For each local node, P creates a shareable frame in the shared space with information describing the unprocessed alternatives. (b) Each local node acquires a pointer to the corresponding shareable frame. (c) P copies its state to Q.
4. P and Q work together to finish processing the shared unprocessed alternatives. They process the shared nodes from the bottom-most to the root node using backtracking.

5. Each worker processes its task exactly as a sequential engine

6. When a worker P creates local nodes and there is an idle worker Q, P shares its local nodes with Q.

7. Execution terminates when all workers become idle.

- **Incremental Copying**

Incremental copying is designed to overcome the overhead of copying a worker state in step 3(c) where Q tries to get the same state as P. The idea is to make Q keep the part of its state which is consistent with P's state, and P copies only the difference. It is implemented as follows:

1. Assume that Q is idle and P has local nodes.

2. Q selects P for sharing its local nodes.

3. Q backtracks to the youngest shared node with P.

4. P copies to Q only the differing parts of its state (top segment of P's trail stack)

The incremental copying idea is related to the itinerary based technique that we have developed using "decomputation-recomputation" as described below in Chapter 4; however, in our model the recomputation is done by the newly activated worker, work is usually given away from the top-most branch point in the search tree, and, of course, the itinerary technique does not require a shared-memory model of parallel computation.

- **Memory Organization**

The Muse model has been implemented as a minimal extension of the WAM. Each sequential engine is a worker with its own four stacks. They share a segment of memory administered as a general heap. The heap is used for storing:

- frames associated with shared nodes that store choice points,
- global tables (symbol table, predicate table),
- asserted rules,
- terms generated during parallel execution of setof and bagof, and
- global registers.

There is one shared frame for every shared node. The frames are allocated by a worker who wants to share the corresponding node, and deallocated by the last worker backtracking from the node. They are referenced from the corresponding choicepoint frames. Each frame contains:

- a pointer to the next alternative of the node,
- a bit map indicating workers at and below the node, and
- a lock for synchronizing accesses to the node.

- **Modifications to the WAM**

As an extension to the WAM, one extra field is added to each choicepoint frame. For a shared node, this field contains a pointer to the next child choicepoint frame in order to find quickly the child node on cut/commit operations. For a non-shared node, it contains the number of remaining alternatives of the nonshared nodes older than this node, in order to measure the local load of each worker for guiding the scheduler. (An idle worker will be sent to the busiest worker.)

On every procedure call, a worker checks for interrupt signals from other workers. Possible interrupts can be a request to make local nodes shareable, or a signal to abort the current task due to cut/commit operation.

- **Characteristics of the Execution Model**

The basic characteristics of the execution model can be listed as:

Efficiency:

- High degree of locality of references (WAM stacks are not shared among workers; Q copies top stack segments from P's cache),

Simplicity:

- Easy to adapt any sequential Prolog system with very low extra overhead.
- Keeps all advantages of the sequential Prolog execution (efficient compilation, indexing, unification with constant access time, stack based storage management, garbage collection, etc.)
- Suitable for any multiprocessor system supporting: (1) local and global address spaces, and (2) copying of memory blocks from one local space to another. These functions are supported by the DYNIX (Sequent Symmetry) and MACH (Butterfly) operating systems.

- **Scheduling**

Nodes of the search tree correspond to WAM choicepoints. The search tree is divided into shared and local parts. A shared node is accessible to workers below that node. Local nodes are only accessible to the worker that created them. Workers are in scheduler mode in the shared part and they are in engine mode in the local part.

The main functions of the scheduler are (1) maintaining the sequential semantics of Prolog, and (2) matching idle workers with the available work with minimal overhead. For the scheduler, the sources of overhead are copying a part of the worker's state, making local nodes shareable, and grabbing a piece of work from a shared node.

Several scheduling strategies to minimize the overhead have been developed in the Muse framework. The scheduler attempts to share a chunk of nodes between workers on every sharing in order to maximize the amount of shared work. When a worker runs out of work from its branch it will try to share work with the nearest worker which has maximum load. Workers which can not find any work will try to distribute themselves over the tree. Thus an idle worker is responsible for selecting the best busy worker for sharing its load. This allows busy workers to concentrate on their task.

The main scheduling algorithm can thus be described as follows. When a worker finishes a task, it attempts to get the nearest piece of available work on the current branch. If none exists, it attempts to select a busy worker with excess local work for sharing. If none exists, it becomes idle and stays at a suitable position on the tree.

- **Overheads of Parallelism**

In the Muse model, the basic overheads of OR-parallel execution are the following:

Idling: time spent in looking for a worker with excess local work when there is no available work in the shared nodes.

Sharing: time spent in making local nodes shared with other workers.

Grabbing Work: time spent in grabbing available work from shared nodes.

Copying: time spent in copying.

Waiting: time spent in synchronization with other workers on sharing and copying activities.

Backtracking: time spent in moving up within the shared region.

Other: time spent in other activities like spin lock, signaling other workers for sharing work, etc.

3.2 The Aurora OR-Parallel Prolog System

Aurora is a prototype OR-parallel implementation of the full Prolog for shared-memory multiprocessors; it is implemented on Sequent and Encore machines. SICStus Prolog has been chosen as the underlying sequential implementation. Aurora is based on the SRI model in which a group of workers cooperate to explore a Prolog search tree starting at the root. The tree is defined implicitly by the program, constructed explicitly during the exploration and discarded eventually where constructing and discarding branches corresponds to resolution and backtracking in Prolog.

The SRI model extends the WAM in the following ways. Any choice point can be a candidate for OR-parallel execution. Nodes of search tree correspond to the WAM choicepoints with extra fields to enable workers to move around the tree and to support scheduling generally. The binding array is divided into a local part and a global part.

- **Memory Management**

To support OR-parallelism, WAM stacks are generalized to "cactus stacks". To achieve this, each worker is allocated a segment of virtual memory, divided into four physical stacks: node stack, environment stack, term stack and trail. Each worker allocates objects in its own physical stack. These objects may also be linked to objects in other workers' stacks. The main difference from the WAM is task switching: the worker may need to preserve data at the base of stacks for the benefit of other workers.

- **Cut, Commit and Suspension**

Aurora supports "cut" and "cavalier commit" where cut selects the first branch and prunes

branches to the right, whereas cavalier commit selects any one branch and prunes branches to the left and right. Cut is implemented by requiring the processing of the branches to the right of the cut to suspend until the cut is processed. Commit does not require any suspension but can not guarantee to prevent side effects.

- **Public and Private Nodes**

The search tree is divided into 2 parts: the public part which is accessible to all workers and the private part which is accessible to the worker creating it. The purpose of this is twofold:

1. A worker in the private part of the tree behaves as a standard sequential engine without being concerned about locking or extra data for scheduling.

2. This division provides a mechanism for controlling the granularity of the OR-parallelism.

When a worker enters the public part, it becomes a scheduler by moving around the public part and coordinating with other workers When a worker enters the private part, it becomes an engine by executing work as fast as possible. Periodically, the engine pauses to perform various scheduling functions, mainly to make its topmost private node public if necessary. The frequency with which nodes are allowed to be made public provides the granularity control. A busy worker always has a pointer to the topmost private node which has a lock, sibling, and parent pointers.

- **Scheduling**

The function of scheduler is to match idle workers with available work. The sources of overhead are installation of bindings, locking to control access to shared parts of the tree and bookkeeping to make work publicly accessible. The scheduler prefers "good" work

which can be loosely defined as large grain size computations. The complications that may arise during scheduling are: (1) large-grain work far away in the tree, and smaller-grain or speculative work nearby, (2) idle workers without available work (stay or guess/position to possible future work sites), (3) unstable terrain (tree is constantly changing), and (4) cut, commit, and suspension.

Workers adopt a depth-first left-to-right search strategy as in Prolog. A worker who finishes his task moves over the tree to take up another task. Workers try to maximize working time, and minimize scheduling time. The search tree is represented by data structures similar to those of WAM. Workers on the same branch share data on that branch. Each worker has a private binding array for conditional (shared) bindings. Conditional bindings are also recorded chronologically in a shareable binding list called "trail." Unconditional bindings are implemented as in the WAM. The binding array and the trail introduce little overhead on a worker while it is working, significant overhead when it switches tasks.

- **Schedulers**

Since the engine and scheduler are identifiable components, the clean interface between engine and scheduler allows different schedulers to be tested easily. Their similarities are:

- All handle cut, commit and side effect predicates.
- All release work only from the bottommost node on a branch.
- All attempt to maintain one, live, shareable node on their current branch.
- None treats speculative work, and
- All consider that all work is equal.

- **Manchester Scheduler**

It matches workers to available work as well as possible by giving work to the closest idle worker. If no work exists, the worker sits idle at its node. Each node has a worker map indicating which workers are at or below the node. There are two global arrays indexed on worker number: one indicates work each worker has available for sharing and its migration cost (number of trail entries from root down to that node), the other indicates status of each worker and its migration cost if it is idle.

If a worker is looking for work, by examining the bit map in its current node it knows which work array entries need be considered and chooses the one with the lowest migration cost. If the subtree contains no work then scanning up the branch it considers larger trees at each step. The execution of cut and commit also uses bit maps to locate and notify the workers in the branches to be pruned.

Other refinements are:

1. *Shadowing:* Idle workers distribute themselves over the tree evenly, each shadowing an active worker in the hope that it will release some work later on.

2. *Delayed re-release:* When a piece of work is acquired by a worker from a node, release of further alternatives from the same node is disabled for a moment in order to avoid congestion of workers.

3. *Lazy release:* Nodes are made public only when there are idle workers in order to avoid creating public nodes that will never be shared.

4. *Straightening:* A node is removed when a worker fails back to it and there is no other branch to try.

- **The Argonne Scheduler**

There is no global dispatcher; this allows workers to make local decisions with a very little use of global data. To find work, any worker in the public part, makes a decision about whether to choose an alternative at its current node or to move along an arc of the tree to a nearby node and repeat the decision process. Since workers are allowed to make their own decision, the strategy is easily modifiable. A bit in each node indicates whether or not an unexplored alternative exists at this node or below. The scheduler tries to maintain at least one active public node on each branch.

- **Wavefront Scheduler**

This scheduler makes use of the fact that the most important part of the tree is at the boundary between public and private regions; new work is found only at the youngest public nodes. The idea is to link together these active nodes allowing more direct access to work. This linked chain is called wavefront . The wavefront also contains the information necessary for updating binding arrays. An idle worker periodically looks for work along the wavefront. When the work is found, the worker (1) reserves the work, (2) possibly performs a wavefront expansion, and (3) removes itself from its previous wavefront position.

3.3 The REDUCE-OR Process Model

This model, developed by [Kalé] for the parallel execution of logic programs, depends on the REDUCE-OR tree representation of search space instead of the traditional AND-OR tree representation because of difficulties associated with the AND-OR trees during parallel execution. One such problem is detecting binding inconsistencies that might occur during parallel execution when workers working under the same node try to unify a variable. The REDUCE-OR tree representation constrains the tree model so that each node represents a completely described (independent) subproblem. There is a *partial solution set* PSS

associated with every node to ensure independence. The PSS contains substitutions that are solutions (most general unifiers) to the literal or query that labels the node and computed from the PSSs of the children nodes. Thus a logic program represented this way renders itself to parallel computation more easily.

A REDUCE-OR tree has two types of nodes: REDUCE nodes and OR nodes. REDUCE nodes correspond to queries while the OR nodes correspond to literals. The tree is rooted at a REDUCE node which is the main query. A REDUCE node can have OR children and an OR node can have REDUCE children. Each OR child of a REDUCE node is an instance of a literal in the parent query. More formally, an OR node is represented by $O(\sigma, G,$ PSS), where σ is substitution, G is a literal, and PSS is a set of substitutions that satisfy G. A REDUCE node is represented by $R(H, \{G_1, \ldots, G_n\},$ PSS), where H and $\{G_1, \ldots, G_n\}$ are the instantiated head and body of some clause of the program, and PSS is a set substitutions that satisfy $\{G_1, \ldots, G_n\}$ and hence H. The nodes in REDUCE-OR trees inherit first two components of the representation from parents or children, and the third one, PSS, is synthesized from children.

The REDUCE-OR process model is derived from the REDUCE-OR trees by providing a process interpretation of the tree. Computation of a query Q with respect to a logic program is a process of growing a REDUCE-OR tree rooted at a REDUCE node labeled with Q. A process is associated with each node of the tree. This can be either a REDUCE process or an OR process. The process model is complete (finds any particular solution) and extracts full OR parallelism by solving the interaction between AND and OR parallelism and efficient bookkeeping mechanisms.

- **OR Process**

An OR process is given a single goal literal G and a context pointer by its parent REDUCE

process. It finds from the program all the clauses whose heads unify with G, and creates substitutions. Then it sends them to the parent REDUCE process if it is a "fact". Otherwise (it is a "rule"), it creates a REDUCE process for each matching rule and then terminates.

- **REDUCE Process**

A REDUCE process for a query Q maintains relations for each literal and node associated with Q. These relations are maintained in the form of a *Data Join Graph*, DJG. A DJG specifies which literals can be solved in parallel and which must wait for data from others. When started, a REDUCE process is given the process id of its grandparent process to whom it returns answers, a parent goal number which represents the context, and a substitution π that expresses the variables in the parent in terms of its own variables. It identifies all the literals that have no predecessors in the DJG of Q (literals that haven't been worked on yet), and starts an OR process to solve each such literal. Then it waits for answers (solutions to the literals) from these processes. As it receives the answers, it tries to construct complete solutions to all literals of Q, and sends them to the (grandparent) REDUCE process.

This model is still in experimental mode and in comparison with other models is premature at this point.

3.4 Parallel Depth-First Iterative-Deepening Search

It is known that breadth-first search suffers from memory limitations and depth-first search suffers from time limitations. The depth-first iterative-deepening (DFID) [Korf] is designed to recover the these limitations by combining their best features. The algorithm performs a depth-first search to depth one; then discarding the nodes generated in the first search, starts over and does a depth-first search to level two; and continuing this process until

a goal node is reached. Thus it is a depth-first search in a breadth-first fashion. The first solution found by DFID is guaranteed to be optimal in solution length. From the complexity theoretic point of view, it has the space complexity of depth-first search, $O(d)$, and the time complexity of the breadth-first search, $O(b^d)$, where b is the node branching factor and d is the depth of the solution node. The DFID is asymptotically optimal among brute-force tree searches in terms of time, space and length of solution.

In the same way that DFID defeated the space complexity of breadth-first search, iterative-deepening A* (IDA*) [Korf] drastically reduces the memory requirements of A* without sacrificing optimality of the solutions found. IDA* consists of repeated cost-bounded (A* style) depth-first search, i.e., it cuts off a branch when its total cost ($f = g + h$) exceeds a given threshold, over the search space. For the first iteration, this threshold is the cost of the initial state; for each new iteration the threshold used is the minimum of all node costs that exceeded the previous threshold in the preceding threshold. The algorithm continues until a goal node is found. If the cost function f is admissible (never overestimates the actual cost), then IDA* is guaranteed to find an optimal solution; that is, optimal in terms of solution cost, time and space, over the class of admissible best-first searches on a tree.

IDA* is parallelized [Kumar Ramesh Rao] by sharing the work done in each iteration among a number of processors. In parallel IDA* (PIDA*), each processor searches a disjoint part of the cost-bounded search space in a depth-first fashion. When a process has finished searching its part of the search space, it tries to get an unsearched part from another processor. The processors uses the ring termination detection algorithm of [Dijkstra] to stop search when a solution is found by one. Since each processor does a depth-first search, it represents its own search space by a local stack of node-children pairs. To minimize the overhead involved in transferring work from one processor to another, each worker keeps a

shared pointer (that can be updated only under mutual exclusion) to the middle of its stack, so no other worker can touch the top portion but can "steal" the bottom portion without notifying or bothering it.

Depending upon when a solution is detected by a processor, PIDA* can expand fewer or more nodes than IDA* in the last iteration. To guard against the "deceleration anomaly" where more nodes are opened by the parallel search than in the sequential, in PIDA*, at least one processor at any time is working on a node n such that everything to the left of n in the tree has been searched.

Chapter 4

The 2LP Model of Parallel Search

In the context of 2LP programming, for the intended class of applications, constraint programming and backtracking are inextricably linked. Hence AND-parallelism as in Strand and other concurrent logic programming languages is ruled out. However, in 2LP, separation of the logic from the mathematical model cleanly defines two axes for parallelization of computations: (1) the parallelization of the numerical processing in the constraint solver (has not been pursued yet), and (2) the natural OR-parallelization of the logic as in Prolog. Thus, 2LP presents a natural candidate to directly transform the non-determinism of logic programming into OR-parallelism.

In order to achieve greater scalability and portability we employ a model of OR-parallelism based on *decomputation* and *recomputation* using an *itinerary* which leads a processor to a point in the search tree. This strategy makes the 2LP system portable across parallel platforms of different kinds. Currently versions are running on a network of Sun workstations using C-Linda *(tuple space model)*, the Intel iPSC Hypercube *(message passing model)*, and the BBN Butterfly *(shared memory model)*. On all three architectures, the parallel code provides significant improvement in performance over the sequential code and benchmarks and analysis are provided in Section 7.6. Moreover, on many examples the sequential code outperforms commercial integer programming codes.

4.1 The Itinerary Technique

The 2LP interpreter carries out a depth-first search with chronological backtracking. At the backtrack, the alternatives are tried in the order they appear. This process generates a tree whose nodes correspond to choice points. Figure 4.1 contains an example 2LP program and its computation tree. The arcs of the tree are labeled by the number of the rule to which that branch corresponds. In this program `$p` has 2 rules, `$q` has 3 rules, `$r` has 1 rule, and `$s` has 2 rules. The procedure `$r` is not a choice point and does not generate a node in the tree. Note that, for its simplicity of drawing, we adopted the convention of representing trees upside down, that is, the root is at the top and as the tree grows (downward) the most recent nodes will be at the bottom. Also by the term "workers", we mean processes each running on a different processor.

```
2lp_main()
{
    $p();
    $q();
    $r();
    $s();
}

$p() {...}
$p() {...}

$q() {...}
$q() {...}
$q() {...}

$r() {...}

$s() {...}
$s() {...}
```

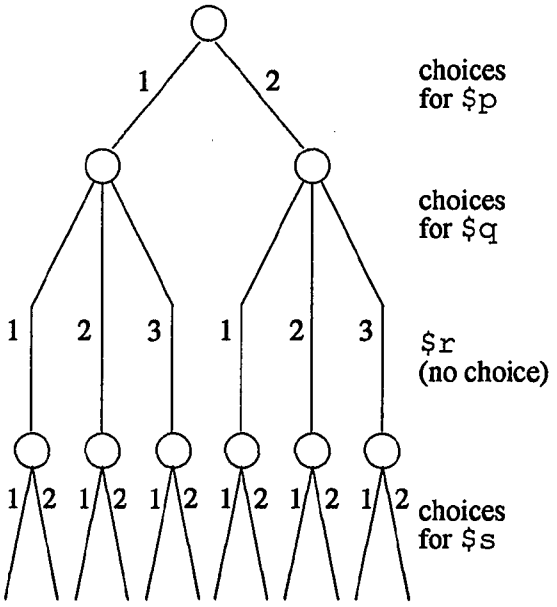

Figure 4.1: A 2LP program and its computation tree

Each node in the computation tree has a unique itinerary that identifies it (Figure 4.2). The itinerary of a node is a list of integers that represents the route to be taken from the

root of the tree to the node. Each integer corresponds to a choice point. The length of the itinerary therefore is equal to the depth of the node.

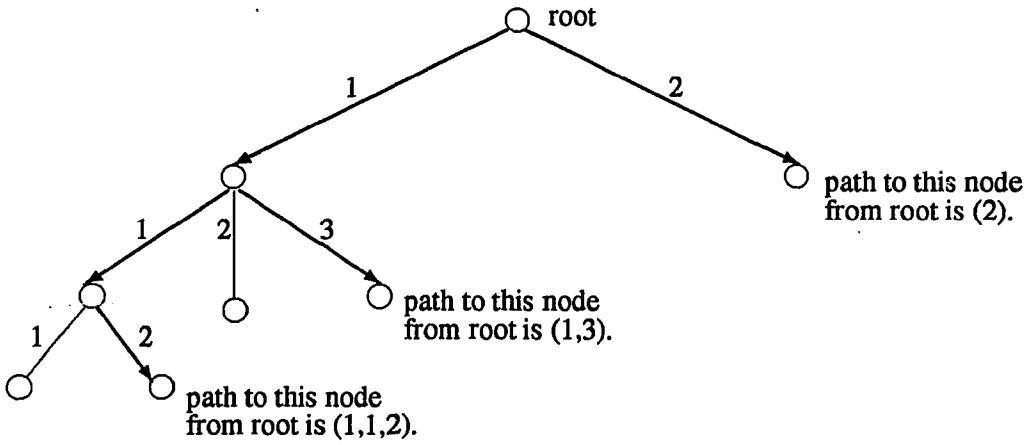

Figure 4.2: Example itineraries in a computation tree

At the beginning, one worker starts with an itinerary of length zero which designates the root of the tree (the main goal) and immediately distributes tasks to the other workers (Section 5.1). After receiving assignments, the workers are independent and each worker starts searching its subtree in a depth-first manner, building its stacks and creating choice points as it expands the nodes on its way. When giving away work, a worker communicates the itinerary of an unexplored node, and a *cutoff level* to which the receiving worker can not backtrack; backtracking to this node is a *fail* for this subtask.

A key aspect of Logic Programming is that the backtracking and choice point machinery are designed to recover state. The itinerary scheduler of parallel 2LP is able to exploit this feature systematically. As a worker develops its search tree and gives off tasks to other workers, it updates its cutoff level and has its stack maintain a record of the choice points encountered on the itinerary leading to its current node. Upon completion of the current task, a worker has preserved the itinerary up to the cutoff level of the fail, the stack including a record of choice points of its computation up to this level, and the

polyhedron corresponding to this level. What is important is that the simplified structure of the S-CLAM stack makes this relatively compact and that the machinery for maintaining the polyhedron does not depend on the level in the procedure stack.

4.2 Shared Itineraries

When given a new itinerary and new initial cutoff level, a worker process can then backtrack to the most recent node common both to its previous itinerary and the new itinerary and then branch off in the new direction. This requires a decomputation and then a recomputation following the itinerary up to the point where new computation starts (Figure 4.3). As it can be seen, taking advantage of the shared itinerary to reduce the recomputation time can be effective anywhere in the computation tree, especially at the deeper levels where full recomputation (starting from the root) is more costly.

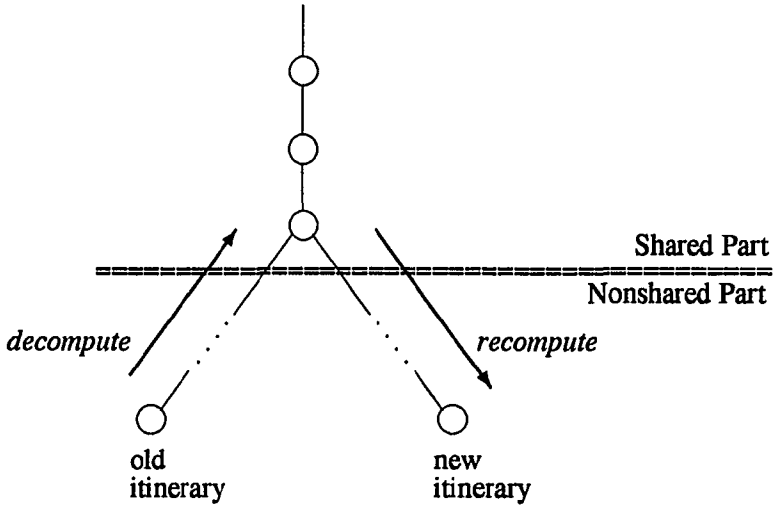

Figure 4.3: Using shared itinerary when switching tasks

Empirical results have shown that the parallel search exhibits a *locality of reference*. By the nature of work distribution strategies described in Chapter 5, the workers give away work in their vicinity of search. This results in clustering of workers in certain parts of

the computation tree. Since the "fails" occur at deeper levels, they exchange work in a certain locality in the computation tree which increases the chance of having a long shared itinerary. It can be imagined easily that the shared part of an itinerary will increase as the locality of search goes deeper. Table 7.5 show that we save more than half of the recomputation on the average, and most of the savings come from itineraries which have length of 7 or more.

At this point let us describe in some more detail the logical engine and mathematical machinery that supports the "decomputation-recomputation" technique in the itinerary method [Atay McAloon Tretkoff].

The 2LP architecture is an abstract machine, colloquially called the S-CLAM for Subcompact Constraint Logic Abstract Machine [McAloon Tretkoff]. The S-CLAM is, *toute proportion gardée*, analogous to the WAM (Warren Abstract Machine) of classical Logic Programming [Warren] and the CLAM (Constraint Logic Abstract Machine) of CLP($\mathcal{R}$) [Jaffar Michaylov]. The 2LP language eliminates structured terms and so the S-CLAM architecture affords a tight integration of the logic with the state of the constraint solver. The procedure stack consists of frames that are similar to the classical ones. However, all frames in the S-CLAM procedure stack are environment frames and no special class of choice point frames is needed. There is no distinction made between argument registers A1,...,An, permanent variables Y1,...,Ym, and temporary variables X1,...,Xk, and unlike the WAM and CLAM, the registers A1,...,An are not overwritten, except under backtracking, deallocation or last call optimization. (Last call optimization is critical in the S-CLAM since in 2LP logical OR-loops and AND-loops are coded by recursions.) In addition to the frame variables, the frame has a field with a CP pointer back to the parent frame with the next instruction to execute upon return to that frame. If the new frame is a choice point, its address is pushed onto the choice point stack; upon

backtracking, control goes to the topmost choice point and its next rule is fired from the same variable settings as the previous rule. This simplified arrangement is possible because no structured terms or constants appear in the heads of rules and no local variables appear in the body of rules. In tandem with the procedure stack the S-CLAM also maintains the polyhedron defined by the current constraints. The polyhedron is maintained by a matrix of constraints **M**, trail stacks which keep track of changes in the upper and lower bounds on variables, and a witness point which geometrically is a vertex on the polyhedron. For search and heuristics, this witness point is available to the program and several built-in functions are provided to access it. By way of example, the built-in identity function `id(x[i])` returns the current value of the ith coordinate of the witness point. These functions can be supplemented by user defined functions or functions from the C library. The convention is that when one of these functions is applied to a continuous variable `x[i]` the argument is in fact the value of the ith coordinate of the witness point. Similarly, if a continuous variable `x[i]` appears on the right hand side of an assignment statement, the value of the witness point is used.

From a mathematical point of view, unless backtracking occurs, the constraints define successively smaller polyhedra in $\mathbf{R}^n$; the constraint matrix **M** thus behaves monotonically as do the upper and lower bounds on the continuous variables. But a salient difference with the classical Logic Programming picture is that the representation of the witness point on the polyhedron's boundary is non-monotonic. The key issue that emerges is the incrementality of the Linear Programming kernel under the "pushing" of constraints. That is, the system must be able to test the consistency of a new constraint without testing the consistency of the entire constraint set from scratch. There is also the dual situation of "popping" of constraints when backtracking.

While a computation is progressing without backtracking, part of the representation

of the witness point is given by the inverse of the submatrix **A** of the constraint matrix **M** determined by the basis columns in the simplex algorithm. We have developed an enhancement of the LU decomposition method of [Fletcher Matthews] which is incremental under the adjunction of new constraints. This method is also locally incremental under change of basis columns in the simplex. This decomposition technique is used with a modified version of the revised simplex algorithm which minimizes use of artificial variables to produce a highly incremental version of Phase I of Linear Programming. Another critical point is that the witness point is maintained without stacking witness points from different generations of choice points. Thus the storage and data structures required for this stay virtually constant over the run of the program.

Incrementality of the mathematical solver as the program forward chains is but one side of the incrementality issue. The state of the constraints and the witness point on the polyhedron at the previous choice point must be restored when the system backtracks. To avoid stacking of witness points, bases, factorizations and other simplex machinery, An incremental "decomputation" method of backtracking to a previous polyhedron is also needed. We have adapted the simplex method to this end. The key remarks are the following: (1) constraints can be removed in any order, and not necessarily in stack order; (2) for each constraint that needs "popping" or variable whose bound needs to be reset, at most one simplex pivot is required; and (3) for the entire backtrack, at most one additional "refactorization" of the matrix **A** is required, no matter how many constraints are "popped."

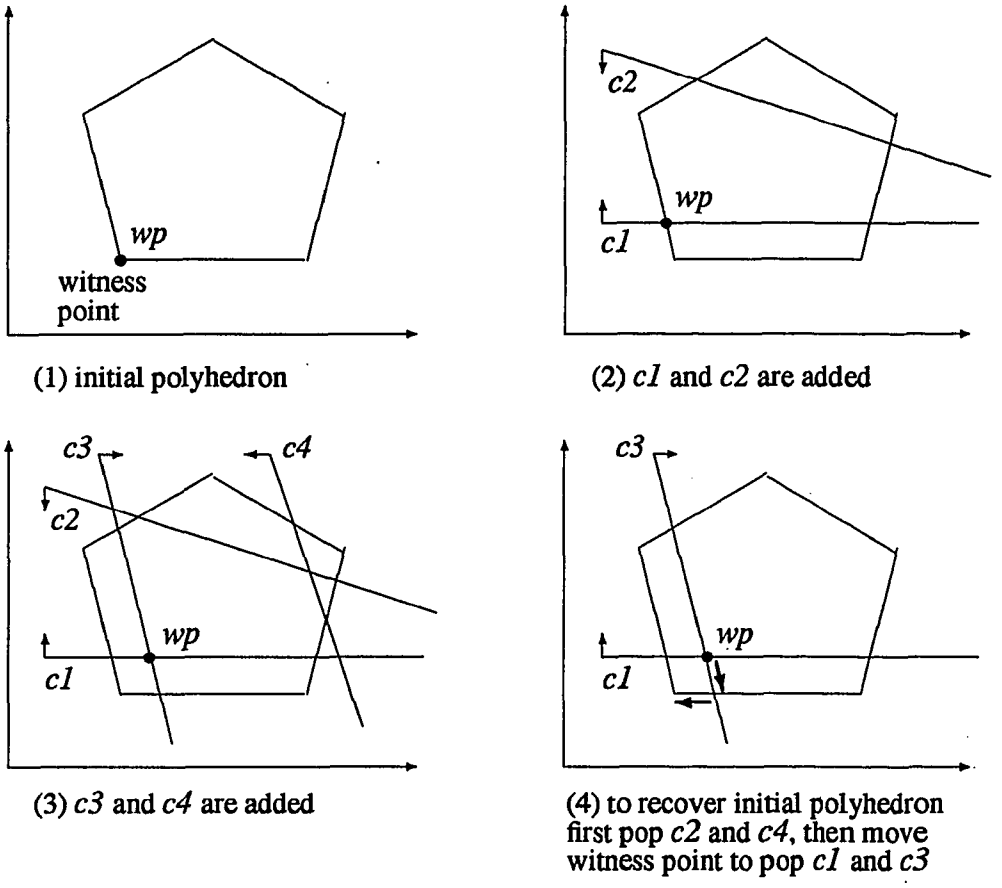

Figure 4.4: Pushing and popping constraints

4.3 Some Definitions about Search

4.3.1 Expanding, Branching and Fathoming

With each node of the search tree is associated a (possibly empty) set of constraints. With that node is associated a procedure call which may have one or more rules. A node is *expanded* when all of its associated rules have been placed on the stack of one or more workers. A node is *fathomed* if no further expansion from that node is possible.

A node that is not fathomed and whose corresponding rule set has not been expanded is called a *live* node. *Branching* means choosing a live node for expanding. If the current node is fathomed, one backtracks along the current path until a node having at least one

live successor is encountered. If there are no live nodes up to the cutoff level the search is completed.

4.3.2 The States of the Nodes During Search

At any time during the search, a node can be in one the following states:

A. Unreached: A node which has not yet been reached by any worker.

1. **Live:** A node which has not been reached but still has a chance to be reached.
2. **Pruned:** A node which has not been reached but has been discarded either because one of its ancestors has been found infeasible or because the current bound on the best solution (z*) is not good enough.

B. Reached: A node which has been reached and expanded.

3. **Active:** A node which has been reached, expanded, and is still under investigation.
4. **Fathomed:** A node which has been reached, expanded, and searched completely.
   a. **Discarded:** Part of an infeasible itinerary.
   b. **Successful:** Part of a feasible itinerary.

4.4 The Description of Parallel Search

Before going into details of sharing work, it has to be made clear how a worker sees the search tree it owns and the data structures it uses for representing this tree. We will now describe the machinery of search used by workers.

A worker keeps its current itinerary in the integer array `iter[]`. It also maintains two integers, the `length` of the itinerary and the `cutoff_level`. When it has failed back to its cutoff level, a worker has finished its subtask, and is in a position to receive a fresh

task. At this point, the length of its itinerary is equal to the cutoff level and its stack is preserved up to that level. To begin a new task, the worker is given a new itinerary and a new cutoff level. These are the only inputs given to the worker. Before it starts on its new task, it updates its stack and polyhedron to reflect the new itinerary it has received. As stated above, this requires a decomputation process to recover the state of the stack and the polyhedron at the first node shared by the old and new itineraries and then a recomputation to bring the the stack and the polyhedron to the end point of the new itinerary. Empirical results from profiling have shown that the cost of decomputation and recomputation is nearly negligible compared to the total run time. We will return to this point below.

To give an intuitive insight rather than the details at the program level, it is helpful to introduce the key data structures of search:

`iter[]`: the array that records the worker's current itinerary.

`cutoff_level`: the level from the root to which backtracking is not permitted. For the first worker initially it is zero; for others initially it is the `cutoff_level` assigned by the process which is sharing work. It is updated when part of the work is given away in order to make sure that the work given away is taken out from the sender's subtree.

`choicepoint_stack[]`: the array to keep the choice points for every node expanded. It is not created for the nodes which have only one choice, e.g., the rule `$r` of the example given in Figure 4.1. When sharing work with a fellow worker, if necessary, it is updated by deleting those parts that have been given away.

`cps_index`: the index of the current level of the `choicepoint_stack` at which the worker is working currently.

`top_of_compass`: starts at -1 and incremented by 1 at every creation of a choice point; so

`iter[0],...,iter[top_of_compass]` is the current branch the worker is working on.

`sextant[]` : the array that records the `top_of_compass` when a choice point is created. This will not change throughout the life of this node. Thus `sextant[cps_index]` is the `top_of_compass` at the creation level of `cps_index`.

Upon backtracking to fail, before `cps_index` can be decremented, `top_of_compass` is set to `sextant[cps_index]`, its value at the creation of that choice point. Then `iter[top_of_compass]` is set to the next branch number.

The cutoff level is the length of the part of the itinerary in which all choices are predetermined for the worker who receives the itinerary. By varying the cutoff level, different work distribution strategies can be developed and tested for better load balancing (Figure 4.6).

This scheme also makes it possible to control the number of branches given at any level again for better load balancing. An example of this is the following:

- itinerary=(2), cutoff=1: only the second branch from the root (Figure 4.5).
- itinerary=(2), cutoff=0: all branches to the right starting with branch 2.

The itinerary technique makes task switching much simpler than in the Aurora and Muse models of parallel Prolog and the S-CLAM architecture makes it natural for a processor to obtain work at the bottommost (earliest) choice point or elsewhere in the search tree depending on the load balancing considerations.

When transferring a task, the worker which gives away work is a *sender* and the one which picks up the task is a *receiver*.

The following is a 2LP program followed by a description of 4 workers solving it. The

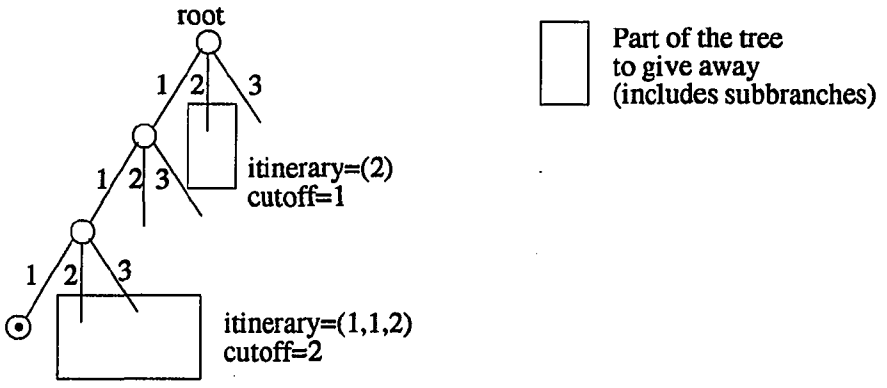

Figure 4.5: Sample works to give away in a computation tree

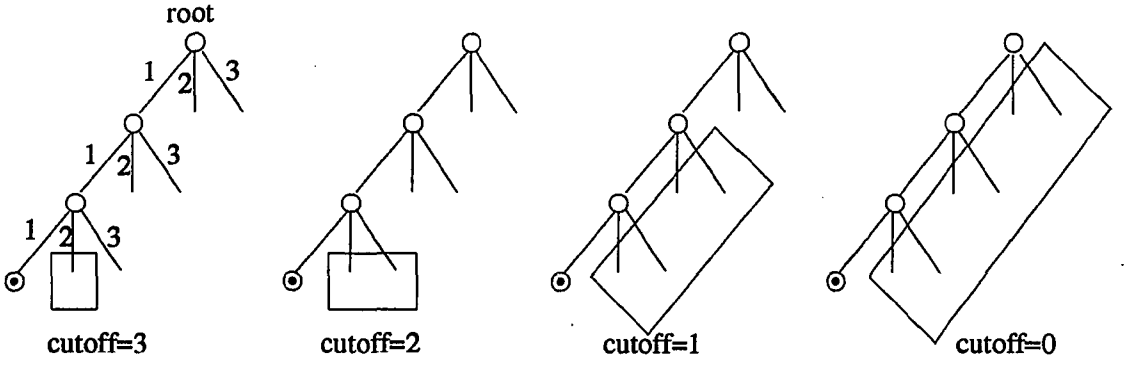

Figure 4.6: Scaling the amount of work given with the itinerary=(1,1,2) by varying its *cutoff level*. (The sender is currently working at the dotted node.)

job is to find all solutions to the 0-1 knapsack problem

$$\sum_{i=1}^{4} ix_i = 4, \quad x_i = 0,1 \quad \text{for all } i.$$

2LP Program:

```
2lp_main()
{
    continuous x[4];
    sigma(int i=0;i<4;i++) (i+1) * x[i] == 4;
    and(int i=0;i<4;i++) $fix(x[i]);
    find_all;
}
$fix(continuous x) { x == 0; }
$fix(continuous x) { x == 1; }
```

Worker Actions: (see Figure 4.7 where the solution nodes are marked with *)

```
Worker1:
    Received itinerary= ROOT  cutoff=0
    Sent itinerary= (2)  cutoff=1
    Sent itinerary= (1,2)  cutoff=2
    Sent itinerary= (1,1,2)  cutoff=3
    Successful itinerary= (1,1,1,2)*
Worker2:
    Received itinerary= (2)  cutoff=1
    Successful itinerary= (2,1,2,1)*
Worker3:
```

```
     Received itinerary= (1,2)  cutoff=2 . . . fathomed
Worker4:
     Received itinerary= (1,1,2)  cutoff=3 . . . fathomed
```

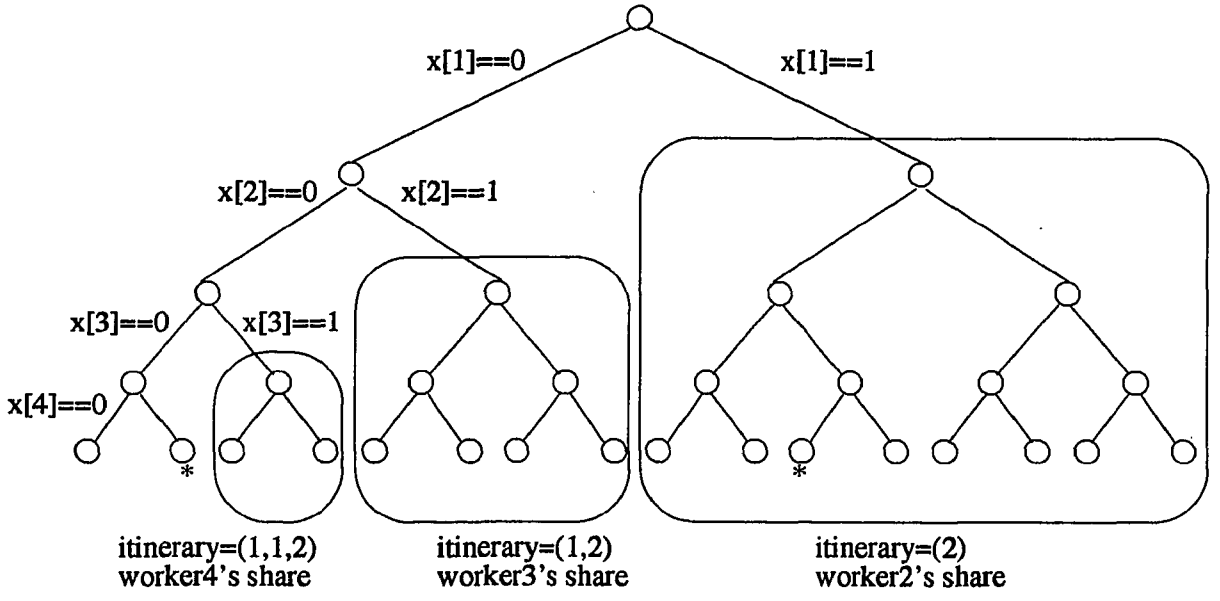

Figure 4.7: The computation tree for the 0-1 knapsack problem and the partitioning of its tree among workers using the strategy 1. The nodes not boxed belong to Worker1.

Chapter 5

The Work Distribution Strategies

For 2LP workers, *work* is a subtree to search. In a 2LP program, the subtrees correspond to alternatives from a choice point. Given that a tree starting at root has subtrees and each subtree in turn has sub-subtrees, etc., a worker that gets a subtree can give away parts of its subtree to other workers. The workers search their trees (or subtrees) in a depth-first fashion, and as they expand a node they push the choice points on their choice point stacks, and pop them one at a time as they finish searching each one. They know that when the stack is empty they need more work. At this time they publicly declare themselves as idle. A busy worker can give away work by simply taking out some choice points from its choice point stack and handing in to an idle worker. The work distribution strategies that we will describe now is about how a busy worker can take out some of its choice points and donate them to an idle worker.

We have to make few more points clear before going into details of work distribution. Obviously, the choice points of a worker are in chronological order on its stack. As long as the busy worker takes away those choices from its stack right after giving them away, no two workers will search the same subtree twice. If we consider the stack of choice points as a linear segment, a worker can choose to give any contiguous part of this segment (one or more choice points) from the bottom or top of the stack. However what is not possible

from the logic programming point of view is that the worker give away "discrete" parts of this stack as a single piece work. This might create backtracking discontinuities on the receiving worker's computation tree, and moreover if there are "cuts" in these segments, the semantics of the computation are compromised.

Thus a worker can give away any "contiguous" segment from its choice points provided that the segment is identifiable by an itinerary on its choice point stack along with its cutoff_level. A segment that can be represented by a single itinerary and a cutoff_level is always contiguous. The cutoff_level prevents the receiver from intruding on the sender's or some other worker's search space. So, when a worker gives away work, it gives part of its own search space and it has to mask from its stack only those choices that are given away.

Since a worker owns the part of the tree that is below its cutoff_level and cannot give away an itinerary above its current_level (since those nodes, if any, are not expanded yet), the length lg of the itinerary it can give should be in between its cutoff_level and current_level. Moreover, this itinerary should agree with the sender's current itinerary up through $lg - 1$ and then fork to one of the available branches that the sender owns. Therefore, a feasible work distribution strategy can be defined to be a function f of the cutoff_level and the current_level of the sender; f returns two integers, the length lg of the new itinerary and the length ncl of the new cutoff_level. These integers must satisfy

$$cutoff\_level \leq ncl \leq lg \leq current\_level.$$

The sender must then update its own choice point stack and the cutoff_level to excise the subtree it has just given away. For simplicity of discussion, except for the initial work distribution, we restrict ourselves to the strategies where the new cutoff_level of the receiver is equal to the sender's old cutoff_level, because this way we generate a bigger chunk of work with a smaller itinerary and the sender needs only to update its cutoff_level in order

to excise the subtree given away (see Figure 5.4).

In parallel 2LP, we seek to preserve the sequential semantics of a program. Whether or not a parallel search strategy preserves these semantics for a given program is a function of both the strategy and the program. Below we will describe a family of search strategies that are based on the principle of maintaining a partition of the computation tree. Programs for which these strategies preserve the sequential semantics will be called *amenable*. The partition of the computation tree among the worker processes will guarantee that the search is complete. All programs which contain only continuous variables and make no reference to witness point are amenable. This class is much broader and includes all programs considered in this paper. The class of amenable programs itself, when defined precisely, is undecidable. In an implementation where witness points are stacked rather than decomputed, references to the witness do not interfere with amenability. Also assignment statements occurring before the first choice point do not interfere with amenability. We address the issue of *cut* further below; this turns out not to be a problem in the 2LP context.

Another important property of this family of work distribution strategies is that at any given time during the search all the nodes to the left of the leftmost worker have been searched. This follows from the fact that workers only give away a contiguous part of their search space which lies to the right of their current itinerary and each worker performs a depth-first search of its subtree. This property obviates the need to keep a sequential worker in order to guarantee that the parallel search will never be slower than the sequential search if we ignore the overheads of parallelism [Kumar Rao].

An extremely generous case of work distribution is the strategy 3 which gives everything possible except the branch which starts at the sender's current node. In this case we have

$$cutoff\_level = ncl \leq lg = current\_level.$$

5.1 Initial Work Distribution

Initial work distribution refers to the way the parallel workers get their initial work from the master worker. We have tested with two different methods of initial work distribution. In the first method, the master divides the search tree into subtrees as equally as possible and sends each piece to a worker. In the second method, the master gives the whole search tree to one of the workers who in turn distributes its subtrees to other workers.

5.1.1 Method 1

The master worker which loads the worker processes to the available processors distributes the work as equally as possible by sending precalculated itineraries. Assuming a uniform ternary computation tree and 7 workers, the master will send the itineraries shown in Figure 5.1 to the workers.

itinerary	cutoff
(1,1)	2
(1,2)	2
(1,3)	2
(2,1)	2
(2,2)	2
(2,3)	2
(3,1)	1

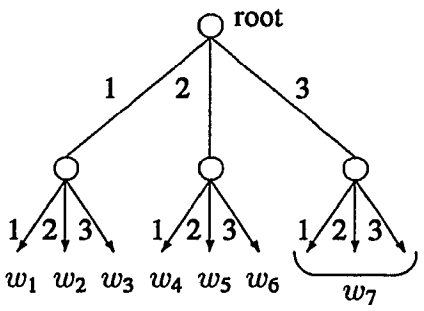

Figure 5.1: Initial work distribution with Method 1 to workers w_i

5.1.2 Method 2

This method assumes that the master sends one root itinerary (of length 0), and $n-1$ dummy itineraries (of length -1) to the available n workers. The ones that get the dummies stay idle. The worker that gets the root itinerary starts distributing work to the idle workers by one of the following two strategies:

- **Strategy A**

It expands the root node and gives one branch to each idle worker keeping always the first branch to itself. If there are still idle workers after distributing the root branches, it expands the next node and starts distributing branches again. It repeats this procedure until all the idle workers are satisfied. Assuming a uniform ternary computation tree and 7 workers (altogether) the works shown in Figure 5.2 will be distributed by the root keeper to 6 other idle workers.

itinerary	cutoff
(2)	1
(3)	1
(1,2)	2
(1,3)	2
(1,1,2)	3
(1,1,3)	3

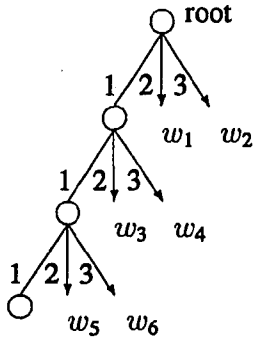

Figure 5.2: Initial work distribution with Method 2.A to workers w_i

- **Strategy B**

Starting from the root node and keeping always the first branch to itself, it distributes the rest of the branches on each level to an idle worker until all the idle workers are satisfied. Again, for the same ternary tree and 7 workers, the works shown in Figure 5.3 are given away to 6 idle workers.

5.2 Intermediate Work Distribution

In the middle of computations, when workers need to exchange work, we consider three basic strategies illustrated in Figure 5.4:

[1] Gives away all the right hand side of the topmost choice point. The itinerary that describes the work given is the smallest itinerary possible, and the ratio $\frac{work\ size}{itinerary\ length}$

itinerary	cutoff
(2)	0
(1,2)	1
(1,1,2)	2
(1,1,1,2)	3
(1,1,1,1,2)	4
(1,1,1,1,1,2)	5

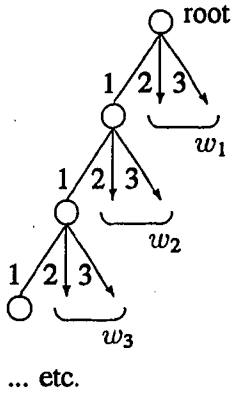

Figure 5.3: Initial work distribution with Method 2.B to workers w_i

is the highest possible, assuming a computation tree with uniformly distributed choice points. In terms of assignment statements, this strategy is

```
receiver_cutoff_level   =  sender_cutoff_level
receiver_current_level  =  sender_cutoff_level + 1
sender_cutoff_level     =  sender_cutoff_level + 1
```

[2] Gives away approximately half of its choice points by selecting an itinerary that has length roughly between the cutoff_level and the current_level. The approximation makes it scalable because the half of the choice points can be anywhere between its cutoff level and current level. In terms of assignment statements, this strategy is

```
k  =  2/3 * (sender_current_level - sender_cutoff_level)
receiver_cutoff_level   =  sender_cutoff_level
receiver_current_level  =  sender_cutoff_level + k
sender_cutoff_level     =  sender_cutoff_level + k
```

[3] Gives away everything possible by keeping only the current branch on which it is working. In most applications it causes thrashing because workers using this strategy

spend too much time on exchanging work. In terms of assignment statements, this strategy is

```
receiver_cutoff_level  =  sender_cutoff_level
receiver_current_level  =  sender_current_level
sender_cutoff_level  =  sender_current_level
```

The results of the benchmarks with different strategies have shown that the best strategies are the ones that give a reasonable size of the tree (ideally close to the half) with as little information as possible. An example of such a strategy is the first strategy described above.

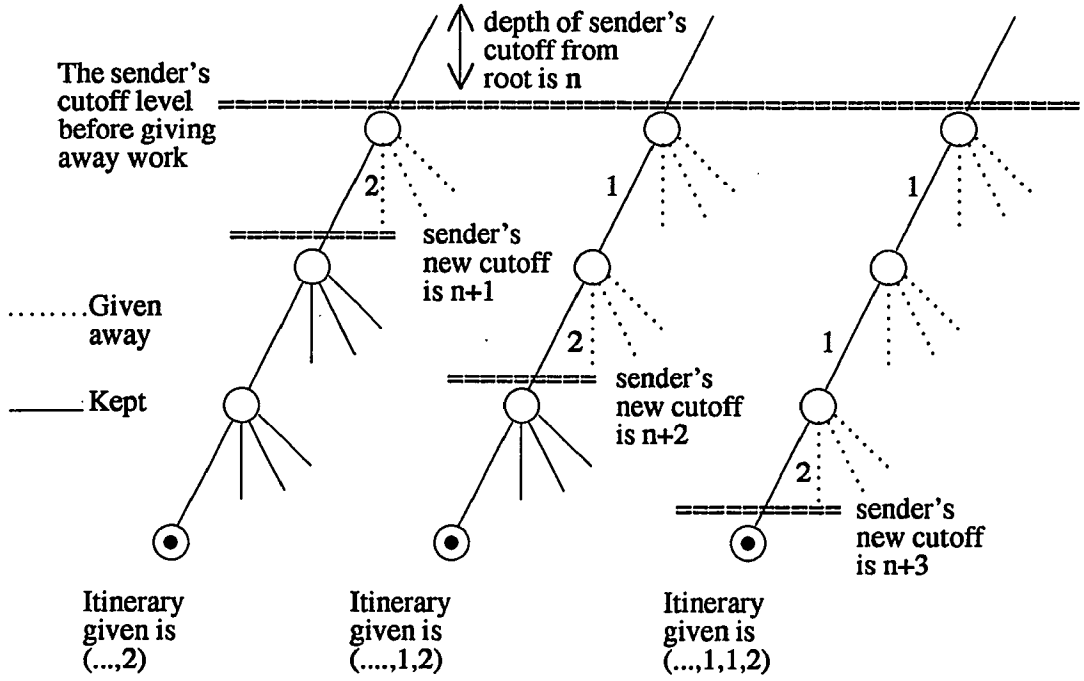

Figure 5.4: Giving away work using work distribution strategies 1, 2 and 3. (The sender is currently working at the dotted node.)

Chapter 6

Theoretical Issues in the Parallel Environment

Various theoretical models of parallelism have been studied. According to the *parallel computation thesis* [Chandra Kozen Stockmeyer] time bounded parallel machines are polynomially related to space bounded sequential machines. If we consider a parallel computer with unbounded number of processors such as the PRAM (Parallel Random Access Machine), the class of problems solvable by a PRAM in polynomial time is equal to *PSPACE*, the class of problems solvable by a sequential machine in polynomial space. At a finer level, for the class *NC* of strongly parallelizable problems in *P*, i.e., those solvable on a PRAM in polylog time [Pippenger], one requires at least a linear number of processors. These theoretical models are unsatisfying in our case where we have a fixed number of processors; we want speedup as a function of the available number of processors. In addition, while unbounded parallelism makes it easy to calculate the resource bounds, in reality it is impossible to achieve these ideal bounds because of overhead inherent in the parallelism, e.g., activation time of processors, communication bottlenecks, network latency, and most importantly the fixed number of processors.

Given the limitations of ideal unbounded parallelism, it is useful to investigate more realistic "bounded" parallelism. In the following discussion, we will use the *number of*

nodes visited as a complexity measure in addition to *time*. Empirical results have suggested that the number of nodes visited by a parallel worker (as well as a sequential worker) is a more stable measure of work done rather than the time spent. This measure is not only stable but also helps isolate the overhead due to parallelism, such as latency, and to abstract the work load of each worker regardless of the platform it is running on. By looking at the benchmarks with different numbers of processors and on different platforms, it is observed that the number of nodes visited by a worker is proportional to the time spent on execution by that worker.

Although the actual complexity of the search for a given program may be *NP*, *DEXP*, *NEXP*, or even *RE* [Cox McAloon Tretkoff], to fix ideas let us think of the search for the optimal solution in an optimization problem as an *NP* problem. After the optimum is reached and solved by a worker, the parallel workers may have a long way to go to verify that it *is* indeed the optimal solution; they are now dealing with a *CoNP* problem. However, the moment at which the *NP* problem turns into a *CoNP* problem can not be known by the workers, whose jobs are only to finish searching their own part of the tree.

If one possesses an algorithm to solve an *NP*-complete decision problem, one can solve the corresponding optimization problem by repeated calls to the decision problem, i.e., a binary search on the range of possible optima, that is $OPT(D) \in P^{NP}$, where $OPT(D)$ is the optimization version of the decision problem D. However to try to analyze the way 2LP works, it is more convenient to divide the actions of the system into an "existential" mode and a "universal" mode. We will thus distinguish two modes of search in the computation tree: (1) *existential* mode where the search terminates when a successful path is found; (2) *universal* mode where the process continues until the entire tree has been generated and no more solutions can be found. The latter mode corresponds both to the situation where the program is asked by *find_all* to determine all solutions and to the situation where the

program fails to find a solution.

The expectation is that in a situation where the communication and other parallel overheads are negligible, a strategy which maintains a partition of the search space will perform a search in existential mode at least as quickly as the sequential strategy, and as often happens much more quickly. On the universal side, with the same assumptions on parallel overhead, a conservative strategy will provide near linear speedup if the worker processes are kept busy.

Before analyzing the complexity of the 2LP model of parallel search, we prove its correctness.

Theorem 1 *The parallel search is complete.*

Proof: The proof of the theorem will use the following lemma.

Lemma 1 (Partition Invariant) *Let N_j be the set of nodes owned by worker W_j after any time step of the parallel search. After each time step the collection of sets $\bigcup_j N_j$ forms a partition of the search tree T.*

Proof: The proof is by induction on k, the number of time steps in the parallel search. After time step $k = 1$ W_1 owns $N_1 = T$, the entire tree. So assume that the invariant holds after time step k. The sets N_j are only changed by a work assignment. Consider the effect of an assignment of work to an idle worker W_i by a busy worker W_b during time step $k + 1$. The assignment of work takes the following form: W_b chooses (the nodes of) a subtree S that he owns, and assigns ownership of S to W_i, i.e., sets $N_b = N_b - S$ and $N_i = N_i \cup S$. Clearly this assignment maintains the partition invariant, i.e., $\bigcup_j N_j = T$ and the N_j are still pairwise disjoint.

Thus after any set of work assignments during step $k+1$ the invariant will be maintained. □

The proof of theorem now follows from the lemma, since the invariant holds after the final time step and termination only occurs when each worker W_j has completely searched the nodes in its N_j. Thus each node of the tree $T = \bigcup_j N_j$ has been searched at termination. □

Given a program $\mathcal{P}$, let v_s be the number of nodes visited by a sequential worker during the course of executing the program $\mathcal{P}$ to completion in universal mode, and v_{p_i} be the number of nodes visited by the parallel worker i ($i = 1, \ldots, N$). Then we have the equation

$$v_s = \sum_{i=1}^{N} v_{p_i} - O$$

where O denotes the nodes revisited for recomputation by the parallel workers. Considering the case of an optimization problem, let e_s be the number of nodes visited to get to the optimal solution (existential part), and a_s be the number of nodes visited after reaching to the optimal solution (universal part). Then

$$v_s = e_s + a_s$$

while

$$v_{p_i} \leq e_s + a_s/N + O/N.$$

Thus assuming the parallel overhead is negligible, the parallel system should provide linear speedup on the universal side of the computation; on the existential side, no slowdown should occur and reasonable speedup can be expected. Typically, in optimization problems, the universal side dominates and thus the linear speedups encountered in the benchmarks of Section 7.6.

Chapter 7

Parallel Implementations of 2LP

7.1 Tuple Space Model: C-Linda on Sun Network

7.1.1 Linda

Linda is a parallel programming model based on generative communication [Gelernter 85]. Generative communication is different from both the shared-memory and message-passing models [Bal Steiner Tanenbaum] because communicating processes do not use shared variables or messages addressed to a specific process, but instead they use an abstract environment called *tuple space*. It is generative because elements of this tuple space, *tuples*, are entities independent from the processes that create them. Tuple space is a logically shared associative (content-addressable) memory. The necessary mechanisms for inter-process communication, process creation and inter-process synchronization are provided through 6 basic operations: *in, out, rd, eval, inp*, and *rdp*. These operations are used to create and manipulate the *tuple space* which includes the active processes running on remote processors.

A tuple is an ordered sequence of typed fields, where each field is an expression that has a value or a potential value, such as ("cutoff", 2) or ("itinerary", 1, 1, 2, 2). Tuples can be inserted into, read or removed from, and evaluated in tuple space. The tuples that are being evaluated are called active tuples. They are the processes running on processors. When

their execution is finished, they turn into passive data tuples in tuple space. Any tuple in tuple space is accessible to any process. In a sense, tuple space can be considered as a dynamic pool of programs to execute, and some data needed to execute them as well as a shared memory used for communication among processes during execution.

The Linda model can be combined with a programming language to produce a parallel programming environment, e.g., the C-Linda parallel programming environment is obtained by integrating the tuple operations into the C programming language. C-Linda systems have been implemented on a variety of shared-memory and distributed-memory parallel systems. Its power and expressivity result from its use of tuple space for interprocess communication and synchronization.

7.1.2 Tuple Operations

The following are the aforementioned tuple operations that are considered powerful enough to turn any common programming language into a complete parallel programming environment:

out(t) The tuple t is added to the tuple space.

eval(t) *eval* is similar to *out*, except that t is evaluated after, rather than before, it enters tuple space. *eval* implicitly forks a new process to evaluate the tuple.

in(p) A tuple that matches the template p is withdrawn from tuple space. If no matching tuple is available in tuple space, the requesting process suspends until one becomes available. If more than one tuple matches the template, then one is chosen arbitrarily.

rd(p) *rd* is similar to *in* except that a copy of the matched tuple is returned and the tuple remains in tuple space.

inp(p), rdp(p) These operations are similar to *in* and *rd*, except that if no matching tuple is found, a failure value is returned immediately; otherwise a success value along with the tuple is returned.

Each tuple operation is atomic, so several different processes can safely manipulate tuple space simultaneously. For example, a semaphore `"semaphore"` can easily be implemented as follows:

```
V operation:   out("semaphore");
P operation:    in("semaphore");
```

To initialize a semaphore's value to n, execute `out("semaphore")` n times.

In a parallel programming environment, a replicated-worker program where each worker executes the same code, depends on a pool of task descriptions. These tasks can be added to the tuple space using

```
out("task_type", <task_description>);
```

and withdrawn by workers using

```
in("task_type", ?result);
```

Linda programs fork processes with the `eval` operation. If we want to turn a conventional loop

```
for (<loop_control>)
      <sub_search>;
```

which evaluates `<sub_search>`, into a parallel loop, first we define a function `sub_search()` and then we simply rewrite the loop as:

```
for (<loop_control>)
        eval("this_loop", sub_search());
```

If `sub_search()` returns a value, these results (which do not necessarily return in the order they are distributed) can be collected and stored by a master process using

```
for (<loop_control>) {
        in("this_loop", ?result);
        store(result);
}
```

7.1.3 2LP and C-Linda

In the Linda model of the 2LP system, a master process creates the worker processes on remote machines using *eval()* and waits until they terminate. The master does not interfere with the computations of workers. The medium of communication is tuple space. The worker processes keep the critical data in the tuple space for the purpose of communicating with each other. They are the following:

`idle_count`: the number of idle workers waiting for work. When this count reaches to the total number of workers, all workers terminate the execution and return to the master.

`current_best_zstar`: the value of the current best solution found so far by workers. This value is also used by workers to prune their search space when they find out that they can not beat this value if they stay on their current branch.

`work`: a structure of `itinerary[]` and `cutoff_level`. Depending on the number of idle workers waiting for work, zero or more `work` structures will be sent to the tuple

space by busy workers. Since a busy worker will "out" `work` only when there is an idle worker waiting for it, the `work` will never have a chance to wait in the tuple space.

`total_workers, shapeup_value, sleeptime`: These are the parameters of the parallel search. Their presence in tuple space is not critical because they can be hardwired to the worker code. `total_workers` is the total number of workers joining the search; `shapeup_value` is the number of "fails" before a busy worker checks the `idle_count` to find out if there is any idle worker out there waiting for work; and `sleeptime` is the sleep time for an idle worker before waking up and check to see whether any busy worker left `work` in tuple space. We have experimented with different `shapeup_value` and `sleeptime` to find their optimal values as well as various strategies that dynamically modify their values during execution. Although the optimal values and dynamic modification strategies for these parameters were different for different applications, we have found that fixing `shapeup_value` to 7 and `sleeptime` to 1 second works very well for all applications.

At the beginning, the master process initializes the tuple space variables and distributes work to worker processes as explained in Section 5.1. As each worker gets work `idle_count` is decreased and is increased when one becomes idle again. At every `shapeup_value` number of "fails", a busy worker will check the `idle_count` and release `work` if there is any idle worker; `idle_count` is incremented by a worker who has become idle, and is decremented by a worker who sends out work. Since tuple operations are atomic, no two workers will try to "out" `work` for an idle worker at the same time. A worker that has become idle updates the `idle_count` as follows:

```
in("idle_count", ?count);
count++;
```

```
out("idle_count", count);
```

and a busy worker executes the following code when it is ready to give away work:

```
in("idle_count", ?count);  /* idle_count is reserved */
if (count > 0) {           /* check if any idle is waiting */
   arrange some_work to give away;
   out("work", some_work);
   count--;
   out("idle_count", count); /* update and release count */
}
```

The basic outlines for the master and worker programs are the following (assuming there are N workers):

```
master() {
   initialize parameters;
   out("shapeup", S);  /* ... and other parameters */
   out("work", 0, 0);  /* root and N-1 dummy itineraries */
   for(i=0;i<N-1;i++) out("work", -1, -1);
   /* now activate the workers and wait for results */
   for(i=0;i<N;i++) eval("worker", worker());
   for(i=0;i<N;i++) in("worker", ?result);
}

worker() {
   rd("shapeup", ?S);  /* ... and other parameters */
```

```
    in("work", ?itinerary, ?cutoff);
    loop1 {
       search the subtree received and
       routinely check idle_count to give away work; also
       update current_best_solution if better one is found;
       when the search is finished increment idle_count;
       loop2 {
          sleep(1);
          in("idle_count", ?count);
          if (count = N) return(1);
          if inp("work", ?itinerary, ?cutoff) goto loop1;
       }
    }
}
```

Despite Linda's elegance and despite the simplicity of the tuple space model, on the Sun network connected through Ethernet, the model suffers from *latency*. Ethernet is a single bus system; all communications between processors are done on the same bus in broadcast fashion. This makes message collisions unavoidable. In case of collision, the sender keeps rebroadcasting after waiting random periods of time until the message reaches to the destination without collision. On the Ethernet bus, the maximum number of collision free operations per unit time is defined as *bandwidth*. (On a dedicated line, the bandwidth is the capacity of the line in bytes/second.) The delay between two operations is the latency. Thus if the processor communication increases, so does latency; this is unavoidable on Ethernet. The implication is that we can not increase the number of workers indefinitely

without causing congestion in the network traffic. Our experiments have shown that the maximum number of processors that can comfortably work together is about 24. If more than 24 processors are activated, Ethernet becomes a "party line", the performance starts degrading and makes it more difficult to finish the task. Most of the times, beyond 32, it even makes the activation of processors impossible.

7.2 Shared Memory Model: BBN Butterfly

7.2.1 The Butterfly

The BBN TC2000 computer is a multiprocessor computer that employs a shared memory. Each processor has its own memory which is divided into a private part and a shared part. All of the shared parts contribute to the total shared memory for the machine. This makes the TC2000 modular and scalable; processors and memory can be added board by board. All the shared memory can be accessed by any processor through an interconnection network called the Butterfly switch. It provides a fast and efficient access path; as opposed to a bus (e.g. Ethernet), the butterfly switch avoids saturation because its bandwidth increases with the number of processors.

The processors are Motorola MC88100 CPU chips that feature RISC architecture, pipelining, floating point execution module, and separate interfaces (two MC88200 chips) for data and instructions. The CPUs are based on 32-bit data and address words, however the TC2000 architecture supports 34-bit global physical address space for a maximum capacity of 16 gigabytes. The memory is a dynamic RAM and locking is used to synchronize access to a memory module.

The UNIX-like operating system nX is used on the Butterfly. Currently, we are using the Uniform System which is an extension library for C and FORTRAN as our parallel programming environment. Under nX, the available processors are configured into clusters

which are allocated for user tasks upon request. The size of a cluster can be decided by user from 1 to the available number of processors. The Lepido machine at Argonne has 38 processors. When a cluster is allocated, the user can run an application on it without interfering with other clusters.

7.2.2 2LP on BBN Butterfly

In the BBN version, the host machine serves as the master for the parallel 2LP system. It creates and initializes the shared variables, and then loads the 2LP code on to the nodes. Using these shared variables, the worker processes running on the nodes communicates with each other. The way the 2LP code runs is analogous to the C-Linda model except that the tuple space is now replaced by the shared memory, and the tuple operations used to access/update tuples are now replaced with shared memory primitives for shared variables.

Compared to the Intel i860 chip which is used in the Intel iPSC Hypercube, the Butterfly chip is about 2-3 times slower, however the fast shared memory access makes it very scalable on small tasks. Near linear speedups down to 2 seconds from 8 seconds on one processor is possible as seen in Table 7.1 (from experiments of Andy Cheng).

	Workers							
Goals	1	2	3	4	5	6	7	8
8-Queens	8.5	4.7	3.2	2.7	2.6	1.9	1.8	2.0
Salt&Mustard	4.4	2.4	1.9	2.3	-	-	-	-

Table 7.1: Some results to show the scalability of BBN Butterfly (time in secs.)

7.3 Message Passing Model: Intel iPSC Hypercube

7.3.1 Hypercube

Geometrically, a *hypercube* (n-cube) is an n-dimensional cube with a processor at each corner and a communication link (channel) between every two processors differing in a

single coordinate. A 3-dimensional cube is shown in Figure 7.1. Each node has a unique address, and is connected directly to 3 neighbors. The neighboring nodes differ by one digit and the differing bit position gives the channel number that connects two neighbors. For example, the nodes 110 and 010 differ in bit position 2 and they are connected through channel 2 (ch_2).

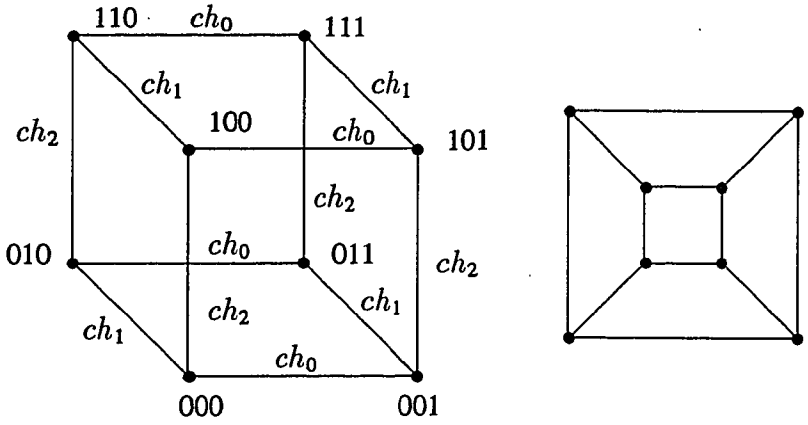

Figure 7.1: Two views of a 3-d hypercube and naming conventions in iPSC

7.3.2 Communication Aspects

A parallel computing system can be viewed as a group of processors connected together by communication links. (Shared memory can be thought of as a kind of communication link too.) As each processor uses its own CPU and local memory to execute programs, they need to communicate the intermediate results with each other. Depending on the amount of communication traffic, the time spent for interprocessor communication can be an important factor in the efficiency of parallel computing. In message passing systems, the message transfer can be done in several ways.

The message can be divided into *packets* of certain size and each packet is sent separately to be assembled at the destination. This method is called *packet switching*. If the processors are not fully connected, packet switching may include a *store-and-forward* protocol where

a packet must travel through several nodes at each of which it is temporarily stored and forwarded to the next processor on the route. Packet switching is a flexible way of sending a message to another processor because the packets of a message can travel to a destination in parallel over several routes and a down processor on route simply causes re-routing a packet through another route. However, it has disadvantages of extra overhead in communication such as disassembling a message into packets (packetization), appending address and control information to each packet, finding the best route, queueing and retransmission at intermediate nodes, and assembling the packets into a message at the destination.

Another method is *circuit switching* which solves some of the problems associated with the packet switching. In circuit switching, the communication link is established by some mechanism before the message transfer begins; so the message itself or its packets (if packetized) can be sent directly to the destination without having them wait at some points along the route.

The hypercube architecture has some interesting features. It provides several independent paths between any pair of nodes and the number of such paths that do not share any branches is exactly n in an n-cube; that is, any two nodes can have n parallel (simultaneous) communication paths. Another feature is that, for any node i there is a spanning tree rooted at i, and the most distant node from the root will be at most n links (channels) away (Figure 7.2) [Bertsekas Tsitsiklis].

A single node broadcast (interrupting other nodes) from root to all nodes takes n time units. It must be noted that, this tree is also used for a single node accumulation (polling the master node) from all nodes to the root, and the time required will be the same as that of single node broadcast. Our experiments with the hypercube have shown that using interrupt (single node broadcast) and polling the master node (single node accumulation)

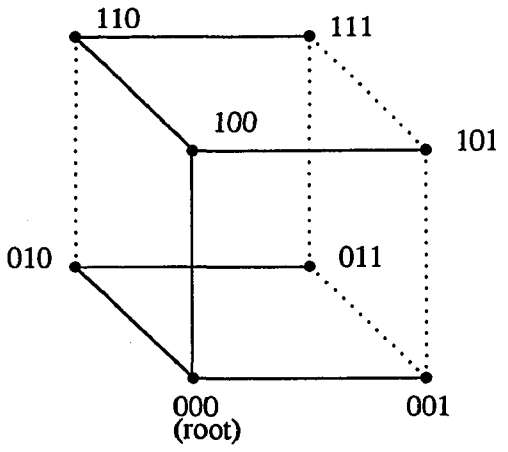

Figure 7.2: A spanning tree of a 3-cube rooted at node 000

for information is effectively the same. Interrupts did not gain us any speedups and in some cases it even made it slower than the polling despite the fact that in polling the processors have to wait a certain period of time to re-poll the master again.

7.3.3 The iPSC

Intel's iPSC Hypercube is a concurrent supercomputer based on message passing. An iPSC system has compute nodes and a front-end processor called the host. The version we have used for the benchmarks is an iPSC/860 with 8 nodes. Each node is an i860 processor with 16M memory. The nodes run the NX operating system, and communicate with each other and the host using message passing. The host (normally a workstation) is the system resource manager, runs UNIX operating system, and is used to load application programs to the nodes. Thus a typical iPSC application has a host program and a node program. The host program allocates a group of nodes (in powers of 2, a cube) and loads the node program on to the allocated nodes. Each node executes the node program, exchanges messages with other node processes via the master and sends the results back to the host. The host then deallocates the cube, making it available for other applications.

The nodes are fully connected using a Direct-Connect routing module (DCM) and

messages do not require any "store-and-forward" protocol. The DCMs connect the hypercube nodes as a circuit switched network. A routing circuitry in each node's DCM dynamically creates message paths through the DCMs of other nodes. Each routing circuitry can drive several channels. The routing scheme works as follows: the sender first sends a message header to the receiver. The header enroute to the destination node opens the gates in each DCM module, clearing a data path for the message. When it reaches the destination, the receiver acknowledges the receipt of the header and the sender sends the whole message directly to the destination without packetization. The major features of DCM are hardware routing decisions, elimination of "store-and-forward" protocol, unlimited message length, and bi-directional message traffic with high bandwidth (3Mb/sec) in each direction. Thus the DCM solves the problem of passing messages to distant nodes quickly.

7.3.4 2LP on iPSC

Although it is not the unique way to implement the 2LP system on a message passing architecture, following the original tuple space implementation for its simplicity, we found it natural to replace tuple space by simulating a small shared memory. The idea is to designate one of the processors as master to keep track of shared variables including itineraries. The use of some form of shared space among workers makes communication between workers much easier to handle at the programming level. However, it must be noted that our parallel implementation is a "minimalist shared memory" model because the amount of shared "memory" needed is limited to about 1Kb.

In this model, the real master processor of the nodes (the host), loads the master of the workers and the worker processes to the nodes. When the workers need to access/update a shared variable, they send a request message to the master worker who in turn serves the

requests of workers one at a time. Designating a node to serve as master (shared variable keeper) automatically serializes the shared variable accesses, and simulates the atomicity of the C-Linda tuple operations.

On the Intel's Hypercube the following routines are used to send, receive and send&receive (query) messages:

```
csend(msg_type, msg_buffer, msg_length, node, pid);
crecv(msg_type, msg_buffer, msg_length);
csendrecv(stype, sbuffer, slen, node, pid, rtype, rbuffer, rlen);
```

Since the master keeps all the shared information and acts as an information center, a worker node that has become idle updates the `idle_count` as follows:

```
csendrecv(idle_count from master with lock option);
/* different options are coded with different msg types */
count++;
csend(idle_count to master for update);
```

and a busy worker executes the following code when it is ready to give away work:

```
csendrecv(idle_count from master);
if (idle_count > 0) { /* check if any idle is waiting */
   arrange some_work to give away;
   csend(some_work to master to be delivered to an idle);
   idle_count--;
   csend(idle_count to master for update);
}
```

The basic outlines for the host, master and worker programs are the following (assuming there are N workers):

```
host() { /* running on the host (front-end processor) */
   allocate a cube of N+1 nodes;
   load master and node programs on the cube;
}

master() { /* running on node 0 */
   initialize parameters;
   csend(root itinerary to worker 1 on node 1)
   for(i=2;i<N;i++) csend(dummy itinerary to worker i);
   loop (until received done-msg from all N workers) {
      crecv(any type of message);
      process the message;
   }
}

worker() { /* running on nodes 1 thru N */
   csendrecv(parameters from master with read option);
   csendrecv(initial work from master);
   loop1 {
      search the subtree received and
      routinely check idle_count to give away work; also
      update current_best_solution if better one is found;
      when the search is finished increment idle_count;
      loop2 {
         sleep(1);
```

```
            csendrecv(idle_count from host with read option);
            if (idle_count = N) csend(done-msg to host) and exit;
            csendrecv(ask_work from host);
            if (got_some_work) goto loop1;
        }
    }
}
```

This model has obviously a potential bottleneck because all communications among workers go through the master. However, since we never had a chance to exceed 7 workers (the Gamma machine at Argonne has 8 nodes), the bottleneck never showed itself. Because of limited number of nodes, we were unable to test the hypercube geometry either for bottlenecks or for efficient pass through of communications.

7.4 DP Model: Message Passing on a Network

The DP (Distributed Processes) model is developed at Brooklyn College by the DP Project Group [Arnow Weiss]. The model is based on pure message passing build on the top of UDP/IP protocol and thus very flexible to include any type of machine to run programs on a network. It supports asynchronous and synchronous message passing as well as signal trapped messages to use for interrupts. There is a host program called *primary* and a node program called *secondary*. The nodes and the host communicates using messages. The host and node programs are distributed to all nodes together. Each processor decides whether to execute host or node program by checking its process id (pid); the zero pid corresponds to parent (master) process, the others correspond to child (worker) processes. In a sense, programming with DP has a flavor of UNIX programming with `fork` that

spawns child processes. Currently the versions of DP are running on the Sun network and PC network running Deskview.

The model is multi-layered, built on network primitives, portable and open to more abstractions. A Linda-like parallel model that will support a logically shared memory and tuple operations is under development by the group.

The DP model of 2LP is implemented and tested on the Sun network in a similar way to our hypercube model. The efficiency of the message passing model enabled us to get similar speedups as our C-Linda model running on the Sun network. The DP provides the following routines for parallel and distributed processing:

`DPinit, DPexit:` to initiate and terminate the programs on the nodes,

`DPsend, DPrecv:` to send and receive messages among processors,

`DPgetpid:` to get the process id; if mypid=0 then I am master else worker,

`DPgetlog:` to get a file pointer for output,

`DPcatchmsg:` to install user defined interrupt handler,

`DPgetmsg:` to receive a trapped message signaled by an interrupt,

`DPmarktime:` to print timing statistics.

7.5 ParaSoft Express Model: Message Passing on a Network

The *ParaSoft Express* (*ParaSoft* Corporation) is a parallel processing "toolkit" that provides tools and utilities to run programs on homogeneous or heterogeneous networks. It is based on message passing model. The main tools of the *Express* are the following:

1. low level and high level communication routines for communicating messages between processors, broadcasting, interrupting, etc.,

2. a transparent I/O system that allows each node to do I/O in single-, multi- or asynchronous mode,

3. a multitasking system for local and remote procedure calls,

4. an automatic domain decomposition library that can map problems to the topology of the parallel architecture; this allows programs to run on different number of processors by changing only a run-time parameter,

5. a source level parallel program debugger,

6. a graphical system for evaluating the performance and a parallel graphics system that can be used during runtime.

Unlike the other parallel systems, *Express* provides two different models for parallel programming, *Cubix* and *Host-Node* models. In Cubix, there is only one program to be executed on each node. However, Host-Node model is similar to the other message passing models; there is host program that runs on the host or one of the nodes designated as master, and there is node program that runs on the nodes. The only limitation in this model is that the node programs are not allowed to do I/O by themselves. We use the Host-Node model that is similar to our hypercube model; we need a master node to keep track of shared variables and itineraries. The outline of the host and node programs are the same as those explained in the hypercube section above.

Among many available routines in the toolkit, mainly the following ones are used in our model:

`exparam`: used to find the runtime parameters such as processor id and the number of nodes (workers) working,

`exchange`: used to simultaneously send and receive data,

`exread, exwrite`: used to send and receive messages among processors,

`exopen, exload, exclose`: used by the host program to allocate, load and deallocate nodes for parallel processing.

Another point about *Express* is that before running a program, the network should be configured according to needs and the *Express* daemons should be started on the nodes that will run the program. Facilities for these tasks are provided and very straightforward to use.

7.6 Experimental Results

In Tables 7.2, 7.3 and 7.4, we present some performance data for 2LP on three different parallel platforms. The benchmarks used are the capacitated warehouse location problems of [Beasley]. The raw data for these problems are obtained from the Operations Research Library of Imperial College, London. The problem description and its 2LP code are presented in Appendix A.5.

As it can be seen from the tables, we get near linear speedups down to one minute in general, and superlinear speedups are not uncommon for some problems of considerable size. After the problem solution time is reduced to a minute, any further increase in the number of processors starts slowing down the execution. The work distribution strategy used was the strategy 1 which gives the best results on the average compared to the other strategies.

In Table 7.5 the effectiveness of using shared itineraries in recomputations is shown on the same problems using 2 and 4 workers. The recomputation savings that arise from the shared itineraries are above 50% in both cases.

In Figures 7.3, 7.4 and 7.5, we present some speedup curves for various size problems selected from the tables. These figures show that as the problem size increases the speedup obtained approaches to linear.

	Workers				
Goals	1	2	3	6	7
cap41	18.2	13.4 *(1.36)*	11.6 *(1.57)*	9.2 *(1.98)*	9.5 *(1.92)*
cap42	30.8	19.5 *(1.58)*	17.6 *(1.75)*	12.9 *(2.39)*	11.2 *(2.75)*
cap43	46.5	28.6 *(1.63)*	24.0 *(1.94)*	17.9 *(2.60)*	15.4 *(3.02)*
cap44	75.7	43.0 *(1.76)*	34.4 *(2.20)*	24.0 *(3.15)*	22.3 *(3.39)*
cap51	267.3	140.1 *(1.91)*	96.4 *(2.77)*	54.4 *(4.91)*	49.6 *(5.39)*
cap61	80.2	44.4 *(1.81)*	37.7 *(2.13)*	21.0 *(3.82)*	19.7 *(4.07)*
cap62	187.5	101.5 *(1.85)*	75.4 *(2.49)*	43.7 *(4.29)*	35.9 *(5.22)*
cap63	286.6	145.9 *(1.96)*	102.9 *(2.79)*	58.0 *(4.94)*	53.3 *(5.38)*
cap64	322.3	161.2 *(2.00)*	116.6 *(2.76)*	68.6 *(4.70)*	60.2 *(5.35)*
cap71	91.7	51.0 *(1.80)*	38.3 *(2.39)*	22.8 *(4.02)*	21.4 *(4.29)*
cap72	201.7	106.6 *(1.89)*	79.2 *(2.55)*	47.2 *(4.27)*	37.6 *(5.36)*
cap73	273.1	136.8 *(2.00)*	98.0 *(2.79)*	55.8 *(4.89)*	50.5 *(5.41)*
cap74	228.8	119.8 *(1.91)*	90.6 *(2.53)*	52.1 *(4.39)*	50.2 *(4.56)*
cap81	25615	13524 *(1.89)*	9501 *(2.70)*	4302 *(5.95)*	3728 *(6.87)*
cap82	65058	35226 *(1.85)*	24992 *(2.60)*	11029 *(5.90)*	9840 *(6.61)*
cap83	107672	58052 *(1.85)*	40980 *(2.63)*	19049 *(5.65)*	16692 *(6.45)*

Table 7.2: Run times in seconds and *(speedups)* on the Intel iPSC Hypercube

	Workers					
Goals	1	2	4	8	16	32
cap41	51.5	36.8 *(1.40)*	27.4 *(1.88)*	33.1 *(1.56)*	29.0 *(1.78)*	31.9 *(1.61)*
cap42	81.3	51.9 *(1.57)*	36.2 *(2.25)*	36.7 *(2.22)*	25.4 *(3.20)*	25.0 *(3.25)*
cap43	122.6	73.9 *(1.66)*	53.6 *(2.29)*	44.6 *(2.75)*	32.6 *(3.76)*	27.6 *(4.44)*
cap44	198.1	111.6 *(1.78)*	70.3 *(2.82)*	47.8 *(4.14)*	38.0 *(5.21)*	41.8 *(4.74)*
cap51	702.6	360.9 *(1.95)*	200.9 *(3.50)*	109.2 *(6.43)*	71.0 *(9.90)*	49.5 *(14.19)*
cap61	214.3	120.7 *(1.78)*	74.8 *(2.86)*	48.0 *(4.46)*	33.5 *(6.40)*	31.2 *(6.87)*
cap62	493.6	266.5 *(1.85)*	155.6 *(3.17)*	80.9 *(6.10)*	65.3 *(7.56)*	38.7 *(12.75)*
cap63	750.4	379.4 *(1.98)*	214.2 *(3.50)*	123.3 *(6.09)*	78.6 *(9.55)*	52.5 *(14.29)*
cap64	839.9	419.0 *(2.00)*	243.7 *(3.45)*	131.8 *(6.37)*	92.1 *(9.12)*	65.7 *(12.78)*
cap71	244.1	135.6 *(1.80)*	81.3 *(3.00)*	55.8 *(4.37)*	30.7 *(7.95)*	35.0 *(6.97)*
cap72	527.8	283.2 *(1.86)*	158.5 *(3.33)*	87.1 *(6.06)*	58.5 *(9.02)*	45.1 *(11.70)*
cap73	708.1	355.8 *(1.99)*	195.2 *(3.63)*	118.2 *(5.99)*	77.7 *(9.11)*	48.2 *(14.69)*
cap74	592.9	308.2 *(1.92)*	184.7 *(3.21)*	111.8 *(5.30)*	64.3 *(9.22)*	57.0 *(10.40)*
cap81	70913	37682 *(1.88)*	20067 *(3.53)*	8968 *(7.91)*	4718 *(15.03)*	2370 *(29.92)*
cap82	177382	96810 *(1.83)*	48867 *(3.63)*	23212 *(7.64)*	12302 *(14.42)*	5811 *(30.53)*
cap83	290918	157883 *(1.84)*	76857 *(3.79)*	40469 *(7.19)*	19646 *(14.81)*	9530 *(30.53)*

Table 7.3: Run times in seconds and *(speedups)* on the Sun network using C-Linda

	Workers					
Goals	1	2	4	8	16	32
cap41	56.4	38.6 *(1.46)*	30.3 *(1.86)*	29.3 *(1.92)*	29.8 *(1.89)*	31.2 *(1.81)*
cap42	93.8	57.0 *(1.65)*	42.6 *(2.20)*	41.8 *(2.24)*	42.3 *(2.22)*	47.1 *(1.99)*
cap43	139.6	84.6 *(1.65)*	63.0 *(2.22)*	51.4 *(2.72)*	52.0 *(2.68)*	56.6 *(2.47)*
cap44	233.4	128.6 *(1.81)*	84.2 *(2.77)*	60.9 *(3.83)*	61.5 *(3.80)*	64.9 *(3.60)*
cap51	814.6	415.6 *(1.96)*	231.0 *(3.53)*	121.7 *(6.69)*	74.7 *(10.90)*	57.2 *(14.24)*
cap61	248.1	134.0 *(1.85)*	80.0 *(3.10)*	40.7 *(6.10)*	29.0 *(8.56)*	26.2 *(9.47)*
cap62	574.7	302.1 *(1.90)*	167.8 *(3.42)*	83.9 *(6.85)*	53.7 *(10.70)*	41.2 *(13.95)*
cap63	876.2	432.0 *(2.03)*	236.0 *(3.71)*	126.0 *(6.95)*	73.8 *(11.87)*	51.4 *(17.05)*
cap64	985.8	479.6 *(2.06)*	271.2 *(3.63)*	154.2 *(6.39)*	86.0 *(11.46)*	65.0 *(15.17)*
cap71	281.5	150.0 *(1.88)*	86.5 *(3.25)*	42.9 *(6.56)*	27.4 *(10.27)*	25.3 *(11.13)*
cap72	613.4	320.8 *(1.91)*	174.9 *(3.51)*	86.0 *(7.13)*	50.3 *(12.19)*	36.6 *(16.76)*
cap73	826.4	406.3 *(2.03)*	220.1 *(3.75)*	109.2 *(7.57)*	62.8 *(13.16)*	45.6 *(18.12)*
cap74	692.7	351.9 *(1.97)*	206.3 *(3.36)*	114.5 *(6.05)*	63.1 *(10.98)*	43.0 *(16.11)*
cap81	80268	42336 *(1.90)*	22628 *(3.55)*	10872 *(7.38)*	5332 *(15.05)*	2653 *(30.26)*
cap82	202583	109648 *(1.85)*	55431 *(3.65)*	27277 *(7.43)*	13497 *(15.01)*	6473 *(31.30)*
cap83	335203	180724 *(1.85)*	88023 *(3.81)*	45333 *(7.39)*	24146 *(13.88)*	11715 *(28.61)*

Table 7.4: Run times in seconds and *(speedups)* on the BBN Butterfly

	2 Workers		4 Workers	
Goals	Total length	Shared length	Total length	Shared length
cap41	2	0	55	23
cap42	9	3	53	21
cap43	14	6	63	16
cap44	10	3	43	8
cap51	39	24	48	8
cap61	17	7	106	56
cap62	14	6	216	112
cap63	56	38	245	127
cap64	37	21	231	118
cap71	24	13	111	38
cap72	19	7	115	53
cap73	33	19	154	89
cap74	48	32	158	69
cap81	45	28	335	169
cap82	28	16	736	454
cap83	79	57	307	193
TOTALS	474	(59%) 280	2976	(52%) 1554

Table 7.5: Comparing the total itinerary lengths and shared lengths

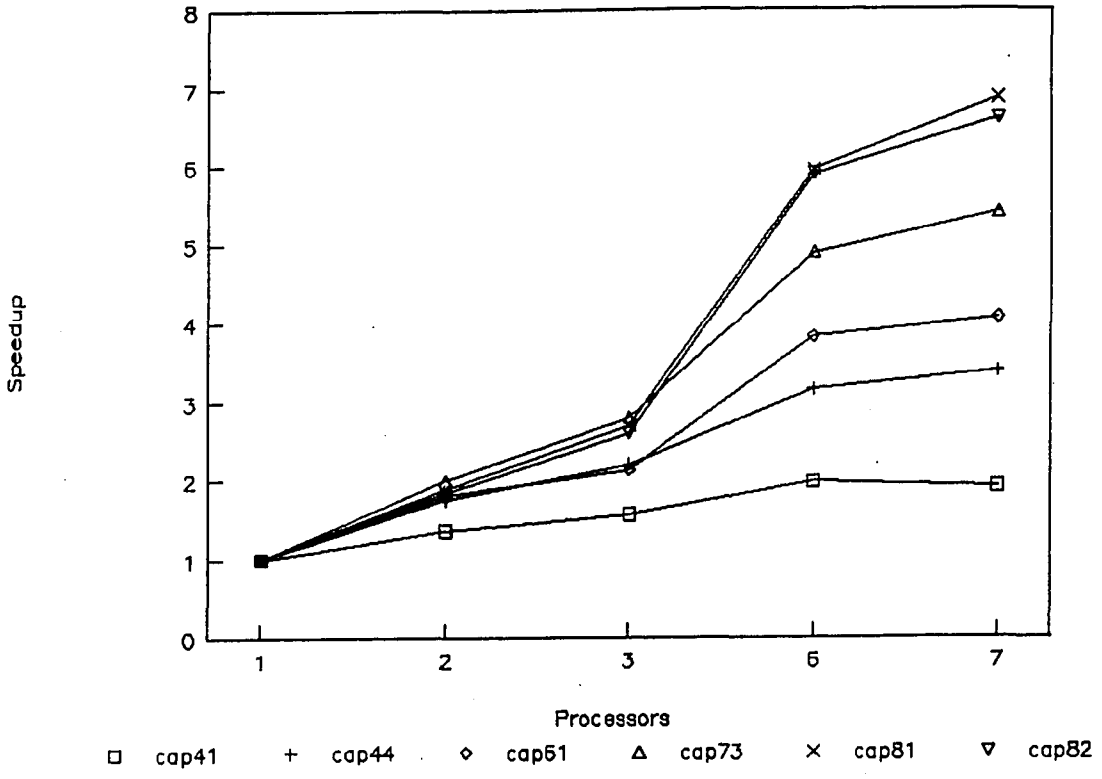

Figure 7.3: Speedup curves for various size problems on the Intel iPSC Hypercube

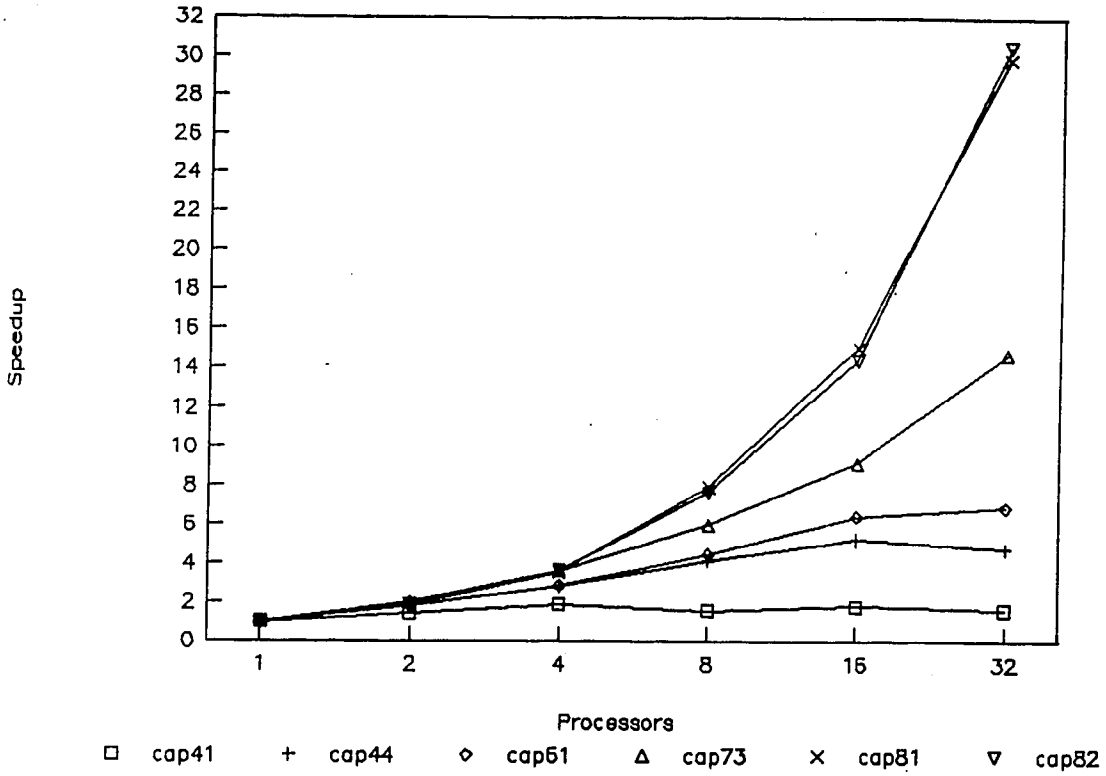

Figure 7.4: Speedup curves for various size problems on the Sun network with C-Linda

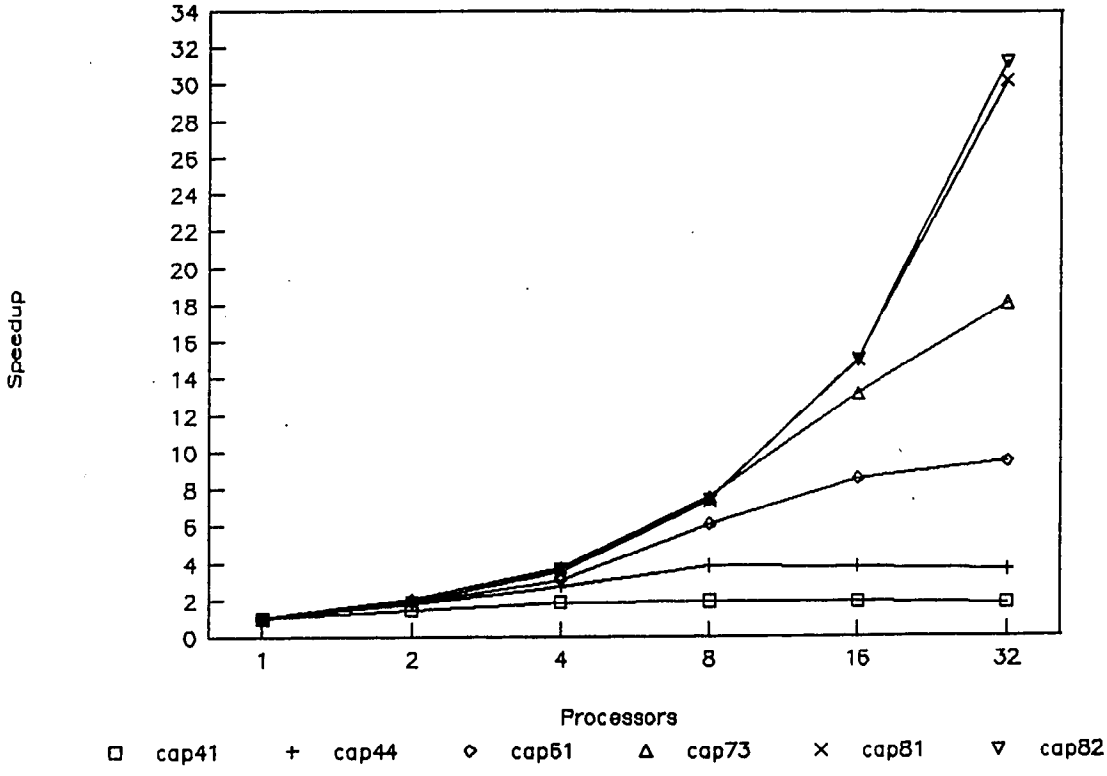

Figure 7.5: Speedup curves for various size problems on the BBN Butterfly

Appendix A

Mixed Integer Programming Using 2LP

A.1 Integer Programming and 2LP

In optimization problems, there are cases in which some or all of the variables are required to be integer. They are called *mixed* or *pure* integer programs. Integer programs are *nonlinear* by nature since the feasible region consists of discrete points. This makes integer programming (IP) harder than linear programming (LP). However LP methods are important to IP problems because if we drop the integer restrictions from an IP problem, the resulting continuous problem is an LP problem whose solution gives a reasonable bound on the solution of its IP counterpart, if not an integer solution.

Another difficulty with IP problems is that although finite algorithms exist, none of them is efficient from computational point of view. Also, as the number of integer variables increases, IP algorithms might take unexpectedly longer time. This contrasts the LP problems where large ones can be solved in a reasonable time frame.

A solution to IP problems can be thought as solving its LP counterpart and expecting an integer solution (prayer algorithm) or else rounding the continuous solution to the closest integer. In case of rounding, a rounded solution can not be guaranteed to be feasible and also if the rounded variable is a 0-1 decision variable, the result can not be justified.

There are two basic techniques for solving IP problems: cutting plane methods and search

methods. Cutting plane methods start with the continuous optimum and by systematically adding "cut" constraints try to reach the integer optimum. The basic idea behind the search methods is to enumerate all feasible integer points and try them in some order to find the optimum. Among the search methods, the most effective one is the branch-and-bound technique which starts with the continuous optimum, and tries to find the integer optimum by systematically partitioning the solution space and pruning the parts that contain no feasible integer solution or contain solutions no better than the current best solution found so far. 2LP, being a parallel constraint logic programming language, uses the branch-and-bound search technique which is very amenable to parallelism. The parallelism of 2LP programs is transparent to the programmer and is directly exploited by the 2LP system.

In the following sections, the declarative aspects of the 2LP as well as its power in handling the MIP problems are shown on typical applications from [Nemhauser Wolsey], [Taha], and [Williams]. We will first start with the following simple LP problem and its 2LP formulation. Note that the *continuous* variables by default are nonnegative, hence the nonnegativity constraints do not appear in the 2LP code.

```
Linear Programming Problem:
---------------------------
maximize z = 4 x + 3 y
subject to:
             2 x + 3 y <=  6
            -3 x + 2 y <=  3
             2 x +   y <=  4
                   x, y >=  0
```

```
2LP Code:
---------
continuous x, y;
2lp_main() {
    $load_constraints();
    $optimize();
}
$load_constraints() {
    2*x + 3*y <= 6;
```

```
    -3*x + 2*y <= 3;
     2*x +   y <= 4;
}
$optimize() {
    max: 4*x + 3*y;
}
```

A.2 The Transportation Problem

In a 2LP program, the continuous variables correspond to the decision variables of a linear programming problem, and double/int variables correspond to the input or other variables to which values can be assigned.

To illustrate the idea, we can consider a transportation problem: suppose there are m sources and n destinations, a_i is the amount of supply available at source i, b_j is the demand of destination j, and c_{ij} is the unit transportation cost from source i to destination j. These are the input variables of the problem and they will be declared as `double`. The objective is to determine the amount to be transported from source i to destination j such that the total transportation costs are minimized. Thus the decision variables of the problem are x_{ij}, the amount to be shipped from source i to destination j.

The linear programming model is:

$$
\begin{aligned}
min \quad & z = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij}\, x_{ij} \\
s.t. \quad & \sum_{j=1}^{n} x_{ij} \leq a_i, \qquad i = 1, \ldots, m \\
& \sum_{i=1}^{m} x_{ij} = b_j, \qquad j = 1, \ldots, n \\
& x_{ij} \geq 0
\end{aligned}
$$

A straightforward 2LP code is:

```
//define M and N here
double a[M], b[N], c[M][N];  //input variables
continuous x[M][N];          //decision variables

2lp_main() {
    $initialize_data();
    and(int i=0;i<M;i++) $supply_constraints(i);
    and(int j=0;j<N;j++) $demand_constraints(j);
    $objective();
}
$initialize_data() {
    //initialize the input variables
    //by reading or assigning values
}
$supply_constraints(int i) {
    sigma(int j=0;j<N;j++) x[i][j] <= a[i];
}
$demand_constraints(int j) {
    sigma(int i=0;i<M;i++) x[i][j] == b[j];
}
$objective() {
    min: sigma(int i=0;i<M;i++)
            sigma(int j=0;j<N;j++) c[i][j] * x[i][j];
```

```
}
```

A.3 The Subset Sum Problem

The problem is to find a feasible solution to a 0-1 knapsack equality constraint

$$\sum_{i=1}^{n} a_i x_i = M, \quad x \in B^n.$$

This is an *NP*-hard problem and is used in cryptography where problems of this form are constructed to have a unique solution that corresponds to a message to be transmitted. In such a system the coefficients a_i for $i \in N$ are public information, the message transmitted is M. The problem must have very large coefficients and be "impossible" to solve, except by the receiver who knows a trick for any M. This problem is formulated in [Nemhauser Wolsey] and a fast heuristic algorithm is given which uses a reduced basis algorithm that has a good chance of finding a solution if the problem has a certain form.

We will now introduce a similar problem that shows how 2LP handles this type of problem. The role of a 2LP program is to generate a consistent set of constraints subject to logical conditions. For example to find a "digital" solution to the knapsack

$$\sum_{i=1}^{500} i\, x_i = 1127250, \quad x_i \in \{0,1,2,3,4,5,6,7,8,9\} \quad \text{for all } i,$$

the most straightforward solution is by means of a "generate-and-test"; values for the x_i are generated and then tested for consistency with the constraints. The 2LP code would be

```
2lp_main()
{
        .
        continuous x[500];
        and(int i=0;i<500;i++) x[i] <= 9;
        sigma(int i=1;i<501;i++) i*x[i-1] == 1127250;
```

```
        and(int i=0;i<500;i++) $digit(0,x[i]);
}

$digit(int i, continuous x) { i == 10; !; fail; }
$digit(int i, continuous x) { x == i; }
$digit(int i, continuous x) { $digit(i+1,x); }
```

In the 2LP code, note that the constraints are laid down before values for the `x[i]` are generated by the calls to the `$digit` routine; this routine is a logical OR loop which generates values for the `x[i]`. The effect in the knapsack is that the values are tested automatically as they are generated and that the constraints provide for a builtin lookahead. By this we mean that as the value `x[i]` is generated, there is an automatic check that setting later values to the maximum value of 9 would yield a sum $\leq$ 1127250 and that setting these values to the minimum of 0 would yield a sum $\geq$ 1127250. The reader will have observed that in this example the lookaheads are especially critical because the unique solution is $x[i] = 9$ for all i. Thus the 2LP code is an example of the "constrain-and-generate" paradigm as opposed to "generate-and-test."

A.4 Fixed-Charge Problem

In a typical production planning problem involving N products, the production cost for product i may consist of a fixed cost K_i independent of the amount produced and a variable cost c_i per unit. Thus, if x_i is the production level of product i, its production cost function may be written as

$$c_i(x_i) = \begin{cases} K_i + c_i x_i, & x_i > 0 \\ 0, & x_i = 0 \end{cases}$$

and the objective function is

$$min \quad z = \sum_{i=1}^{N} c_i(x_i).$$

This objective function is nonlinear in x_i because of discontinuity at the origin, and thus untractable analytically. The problem can be written as a tractable one by introducing binary decision variables of the form

$$y_i = \begin{cases} 0, & x_i = 0 \\ 1, & x_i > 0 \end{cases}$$

Then, the zero-one MIP formulation follows

$$\begin{aligned} min \quad & z = \sum_{i=1}^{N} (c_i\, x_i + K_i\, y_i) \\ s.t. \quad & 0 \le x_i \le M\, y_i, \qquad i = 1, \ldots, N \\ & y_i \in (0,1), \qquad i = 1, \ldots, N. \end{aligned}$$

where M is a sufficiently large positive constant that makes $x_i \le M$ redundant. The transformation is introduced only for analytic convenience. Indeed, the added binary variables are extra in the sense that they do not reveal any new information about the solution. For example, $y_i = 1$ in the optimal solution is already implied by $x_i > 0$.

When we code the problem in 2LP, the 2LP code will not have the extra binary variables; it will preserve the original "look" of the problem. The following shows the relevant segments from the 2LP code.

```
double      cfc, c[N], K[N];    //input variables
continuous fixed_cost, x[N];  //decision variables


2lp_main() {
```

```
    ...
    and(int i=0;i<N;i++) $decide(i);
    ...
    $optimize();
}
...
$decide(int i) {  //to produce
    cfc = cfc + K[i];     //set the cumulative fixed cost
    fixed_cost >= cfc;    //and move its constraint
}
$decide(int i) {  //not to produce
    cfc = cfc - K[i];     //reset the cumulative fixed cost
    x[i] == 0;            //and its constraint variable
}             //Note that "fixed_cost >= cfc" will automatically
              //be removed when backtracking occurs and
              //the 2nd version of $decide is run.
$optimize() {
    min: fixed_cost + sigma(int i=0;i<N;i++) c[i] * x[i];
}
```

A.5 The Capacitated Warehouse Location Problem

The capacitated warehouse location problem (CWLP) is the problem of locating a number of warehouses so as to meet the total demand of a set of customers at minimum cost. Each customer has an associated demand and there are constraints on the total demand that can be met from a warehouse. Below is the MIP formulation and the 2LP code for

this problem. The difference between the MIP model and the 2LP code is that the 0-1 decision variables of the MIP model are coded into the control logic of the 2LP code, thus reducing the number of variables and constraints. In this example, the variables saved are $y_i, i = 1, \ldots, M$, and the constraints saved are $y_i \in (0, 1), i = 1, \ldots, M$. This saving becomes more crucial as the problem size gets bigger.

The input variables of the problem:

M : the number of potential warehouse locations,

N : the number of customers,

K_i : the capacity of warehouse i,

FC_i : the fixed cost associated with opening warehouse i,

D_j : the demand of customer j,

C_{ij} : the cost of supplying customer j's total demand from warehouse i.

The decision variables of the problem:

x_{ij} : the fraction of customer j's total demand supplied from warehouse i,

y_i : 1 if warehouse is open, 0 otherwise (needed in MIP formulation only),

`fixed_cost`: the total fixed cost of opening the necessary warehouses (used in 2LP code).

The mixed-integer formulation of the problem:

$$
\begin{aligned}
min \quad & \sum_{i=1}^{M} FC_i\, y_i + \sum_{i=1}^{M}\sum_{j=1}^{N} C_{ij}\, x_{ij} \\
s.t. \quad & \sum_{i=1}^{M} x_{ij} = 1, && j = 1,\ldots,N \\
& \sum_{j=1}^{N} D_j\, x_{ij} \le K_i\, y_i, && i = 1,\ldots,M \\
& 0 \le x_{ij} \le 1, && i = 1,\ldots,M,\ j = 1,\ldots,N \\
& y_i \in (0,1), && i = 1,\ldots,M
\end{aligned}
$$

The 2LP code of the problem:

```
#define M 16  //dimensions in Problem Cap41 of the
#define N 50  //Imperial College Operations Research Library


double K[M], D[N], C[M][N], FC[M];      //data
double cfc;  //cumulative fixed cost


2lp_main()
{
     continuous x[M][N], fixed_cost;
     $initialize_data();
     and(int j=0;j<N;j++) $column_j_sums_to_1(j,x);
     $objective(x,fixed_cost);
     and(int i=0;i<M;i++) $to_build_or_not_to_build(i,x[i],fixed_cost);
}


$to_build_or_not_to_build(int i, continuous x[], fixed_cost)
     { //to build
```

```
        cfc = cfc + FC[i];
        fixed_cost >= cfc;
        sigma(int j=0;j<N;i++) D[j] * x[j] <= K[i];
      }

$to_build_or_not_to_build(int i, continuous x[], fixed_cost)
      { //not to build
        cfc = cfc - FC[i];  //reset non-continuous variable
        and(int j=0;j<N;j++) x[j] == 0.0;
      }

$column_j_sums_to_1(int j, continuous x[][N])
      { 1 == sigma(int k=0;k<M;k++) x[k][j]; }

$objective(continuous x[][N], fixed_cost)
      {
        min: fixed_cost +
             sigma(int i=0;i<M;i++)
                 sigma(int j=0;j<N;j++) C[i][j] * x[i][j];
      }

$initialize_data()
      {
         //read in from external source
      }
```

A.6 Machine Scheduling

The problem is to complete n different operations on a single machine in the minimum time and to satisfy sequencing, noninterference and delivery time constraints. Let x_j be the start time for the operation j, a_j be its processing time, and d_j be its delivery time. Then these constraints can be defined as follows:

1. Sequencing Constraints: If operation i is required to precede operation j, then

$$x_i + a_i \leq x_j.$$

2. Noninterference Constraints: Two different operations i and j can not be processed at the same time. In terms of constraints, this can be expressed as

$$\textbf{either} \quad x_i - x_j \geq a_j \quad \textbf{or} \quad x_j - x_i \geq a_i.$$

These "either-or" constraints create a nonconvex solution space. Thus MIP model requires the introduction of binary variables y_{ij} defined by the following rule into the noninterference constraints.

$$y_{ij} = \begin{cases} 1, & \text{if } i \text{ precedes } j \\ 0, & \text{if } j \text{ precedes } i. \end{cases}$$

With a sufficiently large M, the "either-or" constraints can be expressed with the following *simultaneous* constraints, of which one is bound to be redundant, by employing the binary precedence variables y_{ij}.

$$\begin{aligned} My_{ij} + (x_i - x_j) &\geq a_j \\ M(1 - y_{ij}) + (x_j - x_i) &\geq a_i. \end{aligned}$$

The 2LP alternative is:

```
$non_interference(int i, j) { x[i] - x[j] >= a[j]; }
$non_interference(int i, j) { x[j] - x[i] >= a[i]; }
```

As it can be seen, we do not need decision variables; the nondeterminism in the execution of logic programs takes care of the "decision" automatically while preserving the mathematical formulation as is. Moreover, it is also possible not to use the noninterference constraints at all. In the 2LP program below, an extra variable `clock` that keeps track of the current time makes these constraints redundant as well the binary precedence variables.

3. Delivery Time Constraints: If the operation j should be finished before time d_j, then

$$x_j + a_j \leq d_j.$$

Finally, if t is the total time required to finish all n operations, the MIP model becomes

$$\begin{aligned} min \quad & z = t \\ s.t. \quad & x_j + a_j \leq t, \quad j = 1, \ldots, n \end{aligned}$$

together with the sequencing, noninterference and delivery time constraints.

We now show the 2LP formulation of this problem on a specific instance for simplicity. Consider 8 operations to be scheduled on a single machine that results in manufacturing of 2 products. The precedence diagram of the operations is shown in Figure A.1.

The 2LP code for the problem is:

```
double      d[8], p[8], clock;
continuous  x[8];
```

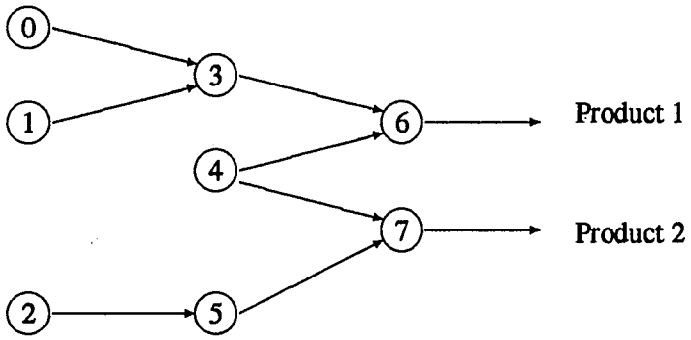

Figure A.1: Sequencing of operations for the single machine problem

```
2lp_main() {
    $initialize_production_times();
    $initialize_delivery_times();
    $sequencing_constraints();
    $delivery_constraints();
    $optimize();
    $schedule_all();
}
$sequencing_constraints() {
    x[0] + a[0] <= x[3];
    x[1] + a[1] <= x[3];
    x[2] + a[2] <= x[5];
    x[3] + a[3] <= x[6];
    x[4] + a[4] <= x[6];
    x[4] + a[4] <= x[7];
    x[5] + a[5] <= x[7];
```

```
}
$delivery_time_constraints() {
    and(int i=0;i<8;i++) x[i] + a[i] <= d[i];
}
$schedule_all() {
    clock = 0;
    and(int i=0;i<8;i++) $one_at_a_time();
}
$one_at_a_time() {
    $pick_one(0);
}
$pick_one(int n) { n == 8; !; fail; }  //operations 0 thru 7 only
$pick_one(int n) { $schedule(n); }     //try this operation
$pick_one(int n) {
    clock = clock - p[n];  //"de-schedule" the previous operation
    $pick_one(n+1);        //and try another one
}
$schedule(int n) {
    clock = clock + p[n];  //update clock
    x[n] == clock - p[n];  //operation n is scheduled
}
$optimize() { min: x[6] + x[7]; }  //if the last 2 min, all min
$initialize_production_times() {
    p[0]= ...;   ...   p[7]= ...;
}
```

```
$initialize_delivery_times() {
    d[0]= ...;    ...    d[7]= ...;
}
```

Bibliography

[Ahuja Carriero Gelernter] S. Ahuja, N. J. Carriero and D. H. Gelernter, Linda and Friends, *IEEE Computer* 19:8, August 1986, 26-34.

[Ait-Kaci] H. Ait-Kaci, The WAM: A (Real) Tutorial, Digital Paris Research Laboratory, Report No. 5, January 1990.

[Ali Karlsson] K. A. M. Ali and R. Karlsson, The Muse Approach to Or-Parallel Prolog, SICS Research Report, SICS/R-90/9009, October 1990.

[Atay McAloon Tretkoff] C. Atay, K. McAloon and C. Tretkoff, 2LP: A Highly Parallel Constraint Logic Programming Language, CUNY Graduate Center, Computer Science Dept., Technical Report 92-4-14, April 1992.

[Bal Steiner Tanenbaum] H. E. Bal, J. G. Steiner and A. S. Tanenbaum, Programming Languages for Distributed Computing Systems, *ACM Computing Surveys* 21:3, September 1989, 261-322.

[Beasley] J.E. Beasley, An algorithm for Solving Large Capacitated Warehouse Location Problems, *European Journal of Operational Research* 33 (1988), 314-325.

[Berndt] D. J. Berndt, C-Linda Reference Manual, Scientific Computing Associates, New Haven, CT, 1988.

[Bertsekas Tsitsiklis] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Carriero Gelernter 89-1] N. Carriero and D. Gelernter, Linda in Context, *CACM* 32:4, April 1989, 444-458.

[Carriero Gelernter 89-2] N. Carriero and D. Gelernter, How to Write Parallel Programs: A Guide to the Perplexed, *ACM Computing Surveys* 21:3, September 1989, 323-357.

[Carriero Gelernter 90] N. Carriero and D. Gelernter, *How to Write Parallel Programs: A First Course*, MIT Press, Cambridge, MA, 1990.

[Chandra Kozen Stockmeyer] A. Chandra, D. Kozen and L. Stockmeyer, Alternation, *JACM* 28:1, January 1981, 114-133.

[Chandra Lewis Makowsky] A. K. Chandra, H. R. Lewis and J. A. Makowsky, Embedded Implicational Dependencies and their Inference Problem, *STOC*, Milwaukee, 1981, 342-354.

[Chang Lee] C. -L. Chang and R. C. -T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[Chvatal] V. Chvatal, *Linear Programming*, W. H. Freeman and Co., New York, 1983.

[Clocksin Mellish] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981.

[Cohen] J. Cohen, Constraint Logic Programming Languages, *CACM* 33:7, July 1990, 54-68.

[Colmerauer 87] A. Colmerauer, Opening the Prolog III universe: a new generation of Prolog promises some powerful capabilities, *Byte*, August 1987, 177-182.

[Colmerauer 90] A. Colmerauer, An Introduction to Prolog III, *CACM* 33:7, July 1990, 70-90.

[Cox McAloon] J. Cox and K. McAloon, Decision Procedures for Constraint Based Extensions of Datalog, To appear in *Constraint Logic Programming*, F. Benhamon and A. Colmerauer (eds.), Addison-Wesley, Reading, MA, 1992.

[Cox McAloon Tretkoff] J. Cox, K. McAloon and C.Tretkoff, Computational Complexity and Constraint Logic Programming Languages, To appear in *Annals of Mathematics and AI*, 1992, and also in *Logic Programming: Proceedings of the 1990 North American Conference*, MIT Press, Cambridge, MA, 1990.

[Dantzig] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[Davis Matiasevitch Robinson] M. Davis, Y. Matiasevitch and J. Robinson, *Proceedings of Symposia in Pure Mathematics: Mathematical Developments Arising From Hilbert's Problems*, Vol.28 (1976), 323-378.

[Davis Putnam] M. Davis and H. Putnam, A Computing Procedure for Quantification Theory, *JACM* 7:2, March 1960, 201-215.

[Dijkstra] E. W. Dijkstra, W. H. J. Feijen and A. J. M. van Gasteren, Derivation of a Termination Detection Algorithm for Distributed Computations, *Information Processing Letters* 16 (1983), 217-219.

[Dincbas *et al.*] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, The Constraint Logic Programming Language CHIP, *Proceedings of International Conference on Fifth Generation Computing Systems*, 1988.

[Disz Lusk Overbeek] T. Disz, E. Lusk and R. Overbeek, Experiments with OR-Parallel Logic Programs, Argonne National Laboratory, MCS Division TM-87, 1986.

[Flatt Kennedy] H. P. Flatt and K. Kennedy, Performance of Parallel Processors, *Parallel Computing* 12 (1989), 1-20.

[Fletcher Matthews] R. Fletcher and S.P.J. Matthews, Stable modification of explicit LU factors for simplex updates, *Mathematical Programming* 30 (1984), 267-284.

[Fourer Gay Kernighan] R. Fourer, D. M. Gay and B. W. Kernighan, A Modeling Language for Mathematical Programming, *Management Science* 36:5, May 1990, 519-554.

[Gallier] J. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper & Row, New York, 1986.

[Garey Johnson] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[Garfinkel Nemhauser] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming*, John Wiley & Sons, New York, 1972.

[Gelernter 85] D. Gelernter, Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems* 7:1, January 1985, 80-112.

[Gelernter 88] D. Gelernter, Getting the Job Done, *Byte* 13:12, November 1988, 301-310.

[Green] C. C. Green, Theorem Proving by Resolution as a Basis for Question Answering Systems, *Machine Intelligence* 4, B. Meltzer and D. Mitchie (eds.), American Elsevier, 1969, 183-205.

[Hausman] B. Hausman, *Pruning and Speculative Work in OR-Parallel Prolog*, Ph.D. Thesis, SICS, Report No. TRITA-CS-9002, March 1990.

[Hentenryck] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA, 1989.

[Hoare 74] C. A. R. Hoare, Monitors: An Operating System Structuring Concept, *CACM* 17:10, October 1974, 549-557.

[Hoare 78] C. A. R. Hoare, Communicating Sequential Processes, *CACM* 21:8, August 1978, 666-677.

[Jaffar Lassez] J. Jaffar, J-L. Lassez, From Unification to Constraints, *Logic Programming Conference*, Tokyo, Springer-Verlag, June 1987.

[Jaffar Lassez 87] J. Jaffar and J-L. Lassez, Constraint Logic Programming, *Proceedings of POPL*, Munich, 1987.

[Jaffar Michaylov] J. Jaffar and S. Michaylov, Methodology and Implementation of a CLP System, *Proceedings of the 1987 Logic Programming Conference*, Melbourne, MIT Press, Cambridge, MA, 1987.

[Jaffar *et al.*] J. Jaffar, S. Michaylov, P. Stuckey and R. H. C. Yap, The CLP(R) Language and System, IBM Report, Yorktown Heights, NY, November 1990.

[Kalé] L. V. Kalé, Parallel Execution of Logic Programs: The Reduce-OR Process Model, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press, Cambridge, MA, May 1987, 616-632.

[Kanellakis Kuper Revesz] P. Kanellakis, G. Kuper and P. Revesz, Constraint Query Languages, *PODS*, Nashville, 1990.

[Kernighan Ritchie] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Khumawala] B. M. Khumawala, An Efficient Branch and Bound Algorithm for the Warehouse Location Problem, *Management Science* 18, 1972, B718-B731.

[Kindervater Lenstra] G. A. P. Kindervater and J. K. Lenstra, Parallel Computing in Combinatorial Optimization, Cornell University, School of OR and IE, Technical Report No. 727, January 1987.

[Korf] R. E. Korf, Depth-First Iterative-Deepening: An Optimal Admissible Tree Search, *Artificial Intelligence* 27, 1985, 97-109.

[Kowalski] R. Kowalski, Logic as a Computer Language, *Logic Programming*, K. L. Clark and S. -A. Tarnlund (eds.), Academic Press, New York, 1982, 3-16.

[Kumar Ramesh Rao] V. Kumar, K. Ramesh and V. N. Rao, A Parallel Implementation of Iterative-Deepening-A*, *Proceedings of the National Conference on AI (AAAI-87)*, 1987, 878-882.

[Kumar Rao] V. Kumar and V. N. Rao, Scalable Parallel Formulations of Depth-First Search, *Parallel Algorithms in Machine Intelligence and Vision*, V. Kumar, P. S. Gopalakrishnan and L. Kanal (eds.), Springer-Verlag, 1990.

[Lai Sahni] T-H. Lai and S. Sahni, Anomalies in Parallel Branch-and-Bound Algorithms, *CACM* 27:6, June 1984, 594-602.

[Lassez Huynh McAloon] J-L. Lassez, T. Huynh and K. McAloon, Simplification and Elimination of Redundant Arithmetic Constraints, *Proceedings of NACLP*, Cleveland, 1989, 37-51.

[Lassez Maher Marriott] J-L. Lassez, M. J. Maher and K. G. Marriott, Unification Revisited, IBM Report, Yorktown Heights, NY, December 1986.

[Lassez McAloon] J-L. Lassez and K. McAloon, A Canonical Form for Generalized Linear Constraints, IBM Report, Yorktown Heights, NY, 1989.

[Lloyd] J. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984.

[Loveland] D. W. Loveland, *Automated Theorem Proving*, North-Holland, 1978.

[Luenberger] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*, Addison-Wesley, Reading, MA, 1973.

[Lusk *et al.*] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brans, M. Clsson, A. Ciepielewski and B. Hausman, The Aurora OR-Parallel Prolog System, Preprint MCS-P122-0190, Argonne National Laboratory 1990.

[Maier Warren] D. Maier and D. S. Warren, *Computing with Logic*, The Benjamin/Cummings, Menlo Park, CA, 1988.

[McAloon Tretkoff] K. McAloon and C. Tretkoff, *2LP:* A Logic Programming and Linear Programming System, Brooklyn College of CUNY, CIS Department, Technical Report No. 1989-21.

[Mendelson] E. Mendelson, *Introduction to Mathematical Logic*, Wadsworth & Brooks, Monterey, CA, 1987.

[Mudambi] S. Mudambi, Performance of Aurora on a Switch-Based Multiprocessor, *Proceedings of NACLP 1989*, E. Lusk and R. Overbeek (eds.), MIT Press, Cambridge, MA, 1989, 697-712.

[Nemhauser Wolsey] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, New York, 1988.

[Nilsson] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Press, Palo Alto, CA, 1980.

[Nussbaum Agarwal] D. Nussbaum and A. Agarwal, Scalability of Parallel Machines, *CACM* 34:3, March 1991, 57-61.

[Papadimitriou] C. H. Papadimitriou, On the Complexity of Integer Programming, *JACM* 28:4, October 1981, 765-768.

[Papadimitriou Steiglitz] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Pippenger] N. Pippenger, On Simultaneous Resource Bounds, Proceedings of the 20th FOCS, IEEE, 1979.

[Robinson] J. A. Robinson, A Machine Oriented Logic Based on the Resolution Principle, *JACM* 12:1, January 1965, 23-41.

[Sakai Aiba] K. Sakai and A. Aiba, *CAL:* Theoretical Background of Constraint Logic Programming and its Applications, *Journal of Symbolic Computation* 8:6 (1989), 589-604.

[Schrijver] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons, New York, 1986.

[Shapiro] E. Shapiro, Alternation and the Computational Complexity of Logic Programs, *Journal of Logic Programming*, Vol.1 (1984), 19-33.

[Sterling Shapiro] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.

[Taha] H. A. Taha, *Operations Research*, Macmillan, New York, 1976.

[Warren 83] D. H. D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI International, 1983.

[Warren 87-1] D. H. D. Warren, OR-Parallel Execution Models of Prolog, *Proceedings of TAPSOFT*, Pisa, Italy, Springer-Verlag, March 1987, 243-259.

[Warren 87-2] D. H. D. Warren, The SRI Model for OR-Parallel Execution of Prolog - Abstract Design and Implementation Issues, *Proceedings of the 1987 Symposium on Logic Programming*, 92-102.

[Warren] D. S. Warren, Implementing Prolog Interpreters and Compilers: Tutorial, *Third IEEE Symposium on Logic Programming*, September 21-25, 1986.

[Whiteside Leichter] R. A. Whiteside and J. S. Leichter, Using Linda for Supercomputing On a Local Area Network, Yale University, Technical Report TR-638, June 1988. YALEU/DCS/TR-638, June 1988.

[Williams] H. P. Williams, *Model Building in Mathematical Programming*, John Wiley & Sons, New York, 1985.