MULTITHREADED CONSTRAINT PROGRAMMING AND APPLICATIONS

by

Fabian Zabatta

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

1999

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.

Date

Professor Kenneth McAloon
Chair of Examining Committee

Date

Professor Stanley Habib
Executive Officer

Professor David Arnow
Professor Jim Cox
Dr. Lioun Wee Chen

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

MULTITHREADED CONSTRAINT PROGRAMMING AND APPLICATIONS

by

Fabian Zabatta

Adviser: Professor Kenneth McAloon

Constraint programming is a powerful and robust software technology for modeling and solving difficult combinatorial problems. However, as problems grow larger in scale and complexity, they become increasingly difficult to solve and their execution times can exhibit inordinate growth. One solution is multithreaded parallelization. Multithreaded parallelization can greatly improve performance, not only increasing the size of solvable problems but also improving solutions that were previously limited by time.

This thesis addresses the fundamental issues concerning the problem of application-based multithreaded constraint programming. The focus of the thesis is on methods for achieving application-based parallelism in constraint programming using threads. Included are the presentations of algorithms such as Dynamic Thread Creation, a new dynamic load balancing scheme specific to multithreading and the Multithreaded Least Discrepancy Search and the Multithreaded Best-First With Backtracking Search, two new parallel search strategies that break away from the classic backtracking schemes normally associated with constraint programming. These new algorithms are applied to benchmark problems with data from the literature and empirical results are reported.

Acknowledgements

I first would like to thank Professors Kenneth McAloon, David Arnow, James Cox, and Dr. Lioun Wee Chen, the members of my dissertation committee. I would like to express special gratitude to my advisor Kenneth McAloon for the introduction to constraint programming, the consistent guidance, and for always keeping my best interests in mind. I would also like to thank Dr. Carol Tretkoff, a mentor who helped inspire me to study parallel search.

Pursuing a Ph.D. would have been much more difficult without the financial support I have received over the past years. I would like to thank Professor Stan Habib of the CUNY Graduate School and Professor Kenneth McAloon for working together to insure the series of fellowships and scholarships I have received. In addition, I would like to acknowledge that this research was supported in part by ONR grant N00014-96-1-1057.

The next thanks go to my colleagues at the Logic Based Systems Lab at Brooklyn College, where the basis of this research was performed. In particular, I would like to express my gratitude to Gerhard Wetzel, Kevin Ying, and Galathara Kahanda for their valuable input.

The last thanks go out to my family. I would not be where I am today if not for the selflessness of my parents, Giovanni and Anna Zabatta. They came to the United States as immigrants and worked hard to provide a better life for their children. Grazie per mia maravigliosa vita. Thanks also go to my sisters, Lucia and Rafaela, for their constant encouragement. Lastly, I would like to thank my soon to be wife, Lori Hutter, for her unconditional love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Constraint programming (CP) is a declarative programming technique that has grown from the collaboration of several research communities including Artificial Intelligence, Computational Logic, Programming Languages, and Operations Research. It has become an indispensable software technology for modeling and solving difficult combinatorial problems. It has been successfully applied to numerous problems such as job shop scheduling [NA,95], portfolio selection [WZ,98], production planning [Lemke,96], robot programming [Pai,95], sports league scheduling [MTW,97] [Henz,98], and various other Artificial Intelligence and Operations Research applications. Its highly declarative nature and its powerful solving methods have led to its commercial success. Many well known companies such as British Airways, Chrysler, Long Island Lighting Company, JD Edwards, and Whirlpool use constraint programming to manage various aspects of conducting business [Pountain,95] [Wallace,96] [ILOG-OPT,98].

1.2 Problem Description

Although powerful and robust, constraint programming is not a panacea. As problems grow larger in scale and complexity, they become harder and harder to solve. What's more, their execution times can exhibit inordinate growth. For some cases, such as in real time systems, a large execution time is unacceptable and for some large problem instances, solving can become impractical or even impossible. In these cases, constraint programming is often abandoned for heuristic methods or restricted by setting a limit on the execution time. In both cases, two outcomes are possible. Either solutions are found, resulting in the best solution within the limit being returned or no solution is found, resulting at best, in a partial solution being returned. In the case of optimization problems, limiting execution time or using heuristic methods may forgo optimal solutions. Although non-optimal solutions are acceptable for some situations, there are instances where businesses or whole industries could save millions in costs by obtaining an optimal solution. Thus, for some cases obtaining an optimal solution can be very attractive. Furthermore, for non-optimization problems, limiting execution time or using heuristic methods can cause solution convergence to become non-deterministic. Thus, methods to decrease execution time are important.

1.3 Proposed Solution

1.3.1 Parallel Search

Constraint programming relies on searches (Section 2.1.2). In most cases, a large portion of the execution time of a constraint program is a result of search time. Thus, improvements of search performance can greatly improve overall performance. In some instances, effective problem modeling and the use of appropriate solving techniques for the given problem can improve search performance. However, for many complex problems, search time can still be unacceptably long. One method to overcome this is through parallelization. Parallelization can lead to several performance benefits, including:

- **Speedup:** The parallelization of a CP application may lead to a significant performance gain over its sequential counterpart. For example, using two processors for an application, instead of one, may decrease the execution time by a factor of two.

- **Large Scale:** Large problem instances that were impractical to solve may now be solved with the additional computing power provided by parallelization. In addition, parallelization may also provide a method to optimize large problem instances.

- **Improved Heuristics:** Heuristic solutions that were limited by time may be improved by parallelization. For example, consider a CP application with the requirement that a solution must be returned within a fixed time limit. In this case, we will accept the best solution returned within the time limit. If we parallelize this application, the best solution found within the time limit, may be significantly better than the result returned if the application was executed sequentially.

In fact, by making the application parallel we may ultimately optimize the solution.

- **Responsiveness:** In systems that require solutions in real-time or pseudo real-time, parallelism may be the only way to achieve acceptable performance. For example, consider a supply chain management application where a customer places orders through the Internet. The goal of the application is to process the order to see if and how the order can be met, and return an answer to the customer in a timely fashion. In this case, parallelization can improve the rate at which a response is returned to the customer and help increase customer satisfaction.

1.3.2 Multithreaded Parallelization

Until recently, multiprocessing required a substantial financial investment in hardware and in software. The mammoth multiprocessor machines of the past cost millions to buy and sometimes, even more to use and maintain. Most institutions could not afford multiprocessing machines. This led researchers to seek alternative methods for the exploitation of parallelism, resulting in distributed parallel computing. Message passing libraries such as PVM [GBDJMS,94], MPI [MPI,94], and DP [Arnow,95] were created, that allowed one to distribute a problem over a network of workstations. This allowed the use of the aggregate power and memory of many computers to create virtual supercomputers. This potential power motivated many researchers to pursue distributed environments, leading to new parallel models. However, today we are faced with new challenges and presented with ever improving computing resources. Computing technology is improving at a remarkable pace. What's more, as hardware such as processors and memory have increased in performance, their prices have decreased. This has created new standards for computing models and performance, especially in the multiprocessor arena. Just as recently as a few years ago, a high performance multiprocessor machine cost over $100,000, see [Thompson,96]. In addition, most of the multiprocessor machines of the past could only operate under a proprietary operating system, which in turn severely limited the number of available software tools. This is no longer the case. Today, because of new advances in design (symmetric multiprocessing), a four processor machine can cost under $12,000 [ZY1,98]. Furthermore, mainstream operating systems such as Windows NT and Solaris have incorporated design features (threads) that allow easy exploitation of these multiprocessor machines. These two factors have made multithreaded parallelism attractive.

1.3.3 Past Parallel Work

Parallel constraint programming has been an active area of research with several themes. One theme, *concurrent constraint programming* [SR,90], is based on the idea of having concurrent agents cooperate by communicating through a shared set of variables and constraints. This work originated from concurrent logic programming and was generalized to constraint programming by [Maher,87] and [Saraswat,93]. Concurrent constraint programming has given rise to new constraint languages such as cc(FD) [VSD,91] and OZ [HSW,93], in addition to formal specification through concurrent semantics [DRKP,91] [SRP,91] [HSW,93] [MRS,94] [MR,95]. However, the focus of concurrent constraint programming has been on the expressiveness of modeling independent events, not application performance.

Another theme of parallel constraint programming has been increasing application performance. Early works such as [LBD,88] [VanHen1,89] [Atay,92] [MS,94] tried to increase application performance by hard coding parallelism into their respective tools. The benefit of this design is that the hard coding shields the application programmer from the complications of parallelism. However, the abstraction comes at a cost. It does not allow the application programmer to fine tune parallelism or to alternate between parallel search strategies. Later works such as [AMT,95] [Muller,97] improved on the previous research by providing an intermediate level of internal parallelization. In these cases, new high-level parallel instructions were added to their respective tools. The new instructions allowed a programmer to create parallel applications with some degree of control. However, in most cases, only one parallel search strategy, parallel backtracking, was provided. It is obvious that one parallel search strategy is not best suited for all problems. Different problems may benefit from different strategies. One method to solve this is to use application-based parallelism: given a thread-safe constraint programming tool (see Section 5.1) and an application, use threads to parallelize the application. This allows a user to implement parallel techniques best suited for a given problem, leading to a more efficient solution. However, application-based parallelism is different from the previous research and presents new challenges. For example, in all the previous research, the developers had access to the internal data structures of their respective tools. This allowed them to implement proven work distribution techniques such as stack copying [VSRL,93], stack sharing [LBD,88], and itinerary based distribution [Atay,92]. Application-based parallelization does not have the benefit of accessing internal data structures. Instead, parallelization must be achieved with a more general environment. This generality has benefits. Namely, it provides a parallel model of computation that is independent of the constraint programming tool used. Thus, application-based parallel algorithms have a greater chance of portability between different constraint programming tools. However, application-based parallelism has not been fully researched. This is particularly true for commercial constraint programming tools. Yet, the advent of relatively inexpensive symmetric multiprocessors, the adoption of threads into mainstream operating systems, and the commercial acceptance of constraint programming, make this work readily applicable.

1.4 Contributions

This thesis addresses the fundamental issues concerning the problem of application-based multithreaded constraint programming. This includes:

- **Constraint Tools:** The type of constraint tools that can be used for multithreading CP applications.

- **Work Division:** Methods for partitioning a CP application to allow parallelism. This includes methods for defining nodes and managing problem data and problem state in parallel.

- **Search Methods:** Two new multithreaded search strategies are presented, along with their applications.

- **Performance:** Background and methods for achieving significant performance gains are presented, including a new dynamic load balancing strategy for multithreaded models.

- **Applications:** Applications are benchmarked and empirical results are reported.

1.5 Outline

The thesis begins with a detailed discussion of constraint programming (Chapter 2), including its history and descriptions of several readily available constraint programming tools. It then details the topic of multithreading (Chapter 3) and multithreaded parallelization (Chapter 4). This includes the presentation of a new dynamic load balancing scheme specific for multithreaded architectures (Section 4.3). The next chapter (Chapter 5) discusses issues and algorithms concerning the problem of multithreaded parallel constraint programming. These algorithms are then used to create two new parallel search strategies that break away from the classic scheme of chronological backtracking that is normally associated with constraint programming (Chapter 6). To show the effectiveness of the algorithms, we apply them on several applications and report empirical results. The last chapter, Chapter 7, summarizes and concludes the thesis.

Chapter 2

Constraint Programming

2.1 Background

Constraint programming (CP) is a declarative programming technique that separates problem modeling from problem solving. This helps insure that the problem to be solved is precisely defined. In addition, this separation can simplify the revision or the extension of a CP application when the corresponding problem changes and also allows one to alternate between different solution approaches without changing the model. In the next sections, we will discuss modeling and solving in constraint programming. Section 2.1.1 describes CP problem modeling and Section 2.1.2 describes the methods used for solving the models.

2.1.1 Problem Modeling

In constraint programming, a problem is modeled in terms of its unknowns. This is usually accomplished by defining a set of decision variables and a set of constraints. A *decision variable* is a pair $\{x, D\}$ where x is a variable (symbol) and D is its *domain*, defined to be its range of possible values. There are many types of decision variables. Each is distinguished by the type of its domain. For example, a decision variable x can be:

- **Boolean:** It can be either true or false.

- **Discrete:** It can take on integer values, e.g. 1 to 5.

- **Continuous:** It can take on real values in one of the intervals: $(-\infty, b]$, $(-\infty, +\infty)$, $[a, +\infty)$ or $[a, b]$

- **Symbolic:** It can take on defined values, e.g. if x represents color, it can take on values of red, green, or blue.

In addition, decision variables can be built-up from these basic types to form new types of variables for problems that require a greater degree of complexity. For example, consider a scheduling application where a new variable type is needed to represent the scheduling of a task on one of two resources. In particular, the new variable type must represent the start time, duration length, and the chosen resource for a task. We can define a new composite variable type that consists of two discrete variables and one Boolean variable. One discrete variable can be used to represent start time, the other can be used to represent the duration of a task, and the Boolean variable can be used to represent which of the two resources was chosen.

A *constraint* is a relation that restricts the possible values of decision variables. To be precise, a constraint on k decision variables $X_1,\ldots,X_k$ is a relation $R(X_1,\ldots,X_k) \subseteq D_1 \times \ldots \times D_k$ where D_i is the domain of X_i for $i = 1,...,k$. Like decision variables, constraints also come in many forms. Constraints are distinguished by the number of variables in their relation and by the structure of their relations. For example some constraint types are:

- **Logical:** x = True *AND* y = False
- **Arithmetic:** $x * y = 300$
- **Cardinality:** Set x only has three elements.
- **Distribution:** The value 2 may only appear three times in array A.
- **Disjunctive:** $x \geq 2$ *OR* $x = 0$

2.1.2 Problem Solving

2.1.2.1 Finite Domains

A *constraint satisfaction problem* (CSP) is given by a finite set of decision variables and constraints. If a given CSP requires the optimization of an objective, it is called a *constraint satisfaction optimization problem* (CSOP). A *feasible solution* or simply a *solution* to a CSP is an assignment to the given decision variables with values from their respective domains that satisfies the given constraints. Formally, given a CSP with n decision variables $X_1,\ldots,X_n$ and m constraints $R_1,\ldots,R_m$, a solution is a mapping $f(X_i) \rightarrow d_i$, $d_i \in D_i$, where D_i is the domain of X_i, for $i = 1,..,n$, such that for each j, $R_j(f(X_{1j}),\ldots,f(X_{kj}))$ holds.

In constraint programming, each constraint R of a CSP is considered as a subproblem and techniques are developed for handling frequently encountered constraints. With each constraint is associated a *domain reduction algorithm* that removes values that are not feasible from the domains of decision variables that occur in the constraint. In addition, communication is facilitated among the constraints to improve efficiency. The basic method used, called *constraint propagation*, links the constraints through their shared variables to provide a level of *consistency*, a degree to which infeasible values are explicitly excluded by constraints. This is accomplished by systematically reducing the domains of the decision variables whenever there is an implication regarding one of the variables of a constraint, as illustrated in Example 2.1. It is important to note that in the case where the domains of the decision variables are finite, constraint propagation will always reach a point where no further domain reduction is possible. This is called a *fixed point*. Furthermore, the domains of decision variables are always reduced in the same way regardless of the order that the constraints are considered. Therefore, the same fixed point will always be reached.

Example 2.1:

x, *y*, and *z* are discrete decision variables whose domains are {1, 2, 3} as shown in Figure 2.1:

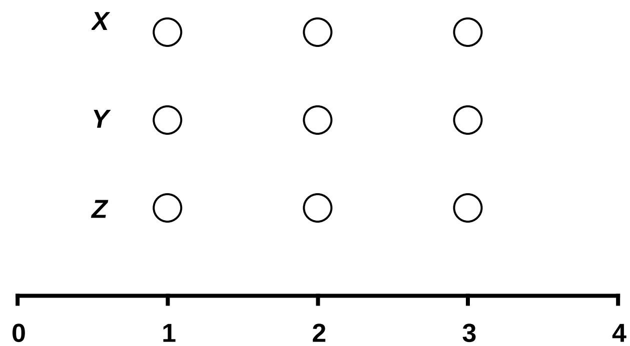

Figure 2.1: Possible values of *x*, *y*, and *z* in Example 2.1

We can post the following constraints on *x*, *y*, and *z*:

$x - y = 1$	(2.1)
$y < z$	(2.2)
$x > z$	(2.3)

By propagation:

After posting Constraint 2.1, we can deduce that the smallest value of *x* must be one greater than *y*. This makes the value of 1 for *x* and 3 for *y* inconsistent. We can remove these inconsistent values and reduce the domain of *y* to {1, 2} and the domain of *x* to {2, 3}, as shown in Figure 2.2.

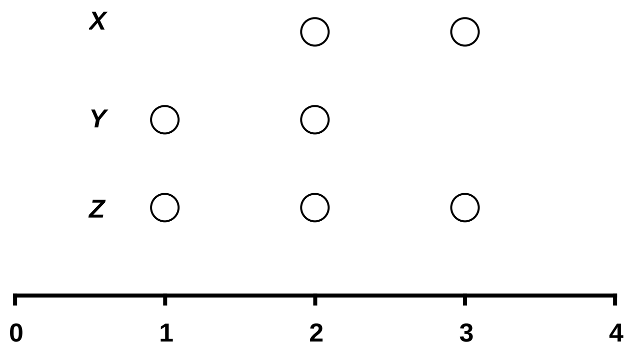

Figure 2.2: Possible values of *x*, *y*, and *z* after propagation of Constraint 2.1 in Example 2.1.

After posting Constraint 2.2, we can deduce that the value of *z* must be one higher that the value of *y*. Thus the value of 1 for *z* is inconsistent and we can reduce the domain of *z* to {2, 3}, as shown in Figure 2.3.

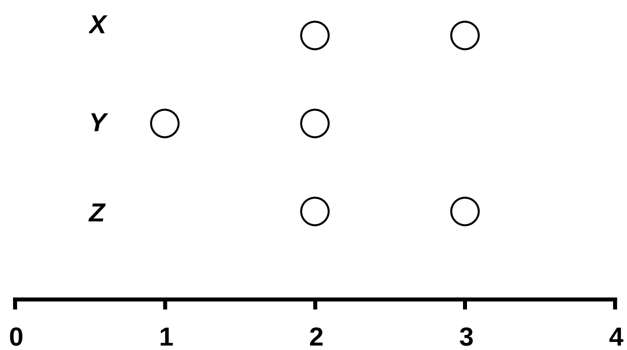

Figure 2.3: Possible values of *x*, *y*, and *z* after propagation of Constraint 2.2 in Example 2.1.

After posting Constraint 2.3, we can deduce that *x* is one greater than *z* thus the value 2 for *x* and 3 for *z* is inconsistent. Thus, we can reduce the domain of *x* to {3} and the domain of *z* to {2}. After the reduction of the domain of z, we recall Constraint 2.2 and thus we can reduce the domain of y to {1}, as shown in Figure 2.4.

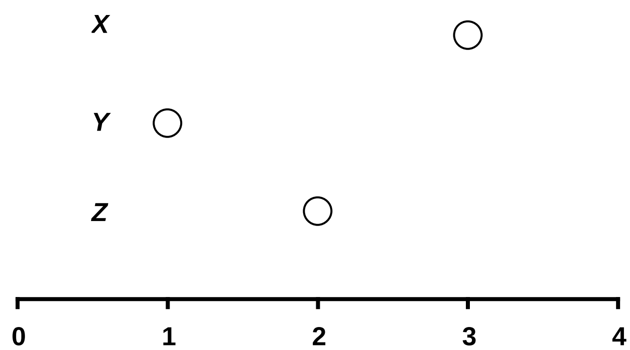

Figure 2.4: Values of *x*, *y*, and *z* after propagation of Constraint 2.3 in Example 2.1.

Thus, we have $x = 3$, $y = 1$, and $z = 2$.

In most cases, constraint propagation alone is insufficient for finding solutions. First, for efficiency reasons, constraint propagation is not meant to detect all inconsistencies. Therefore, not all consequences of posted constraints are deduced. Secondly, there can be instances where a CP application can have more than one solution. Consequently, these two cases can lead to situations where the domains of decision variables can contain more than one value even after constraint propagation has reached a fixed point, see Example 2.2. Thus, a search (Section 2.1.2.4) must be employed to choose values for variables and eliminate inconsistencies. To accomplish this, the search must reduce the domain of each decision variable until it has a cardinality of one, while assuring that all of the constraints are satisfied.

Example 2.2:

x and *y* are discrete decision variables whose domains are {1, 2, 3}, as shown in Figure 2.5.

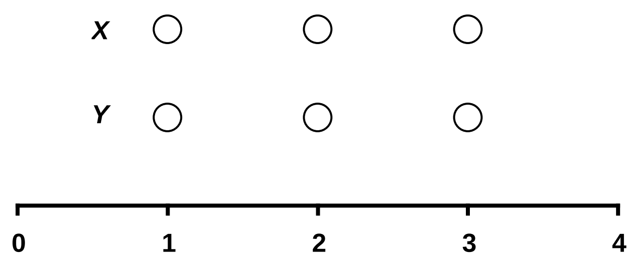

Figure 2.5: **Possible values of *x* and *y* in Example 2.2.**

We can post the following constraints on *x* and *y*:

$$x + y > 3 \qquad (2.4)$$

$$y > x \qquad (2.5)$$

By propagation:

After posting Constraint 2.4, the domains of *x* and *y* remain unchanged.

After posting Constraint 2.5, the domain of *x* is reduced to {1, 2} and the domain of *y* is reduced to {2, 3}, as shown in Figure 2.6.

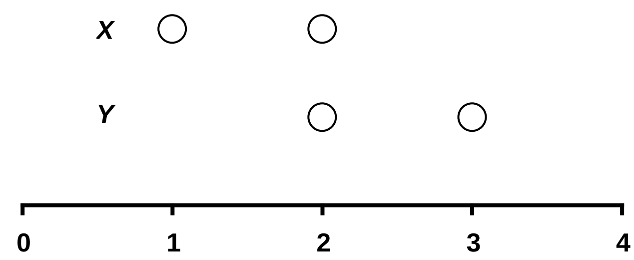

Figure 2.6: Possible values of *x* and *y* after propagation of Constraint 2.5 in Example 2.2.

After initial propagation, we have reached a fix point. However, the variables *x* and *y* have multiple values in their domains.

2.1.2.2 Continuous Domains

In the case of CP applications that have variables with continuous domains and linear constraints, simplex based methods such as Linear Programming (LP) and Mixed Integer Programming (MIP) are used [Dantzig,63]. In this case, the solving engine is often referred to as a *linear solver*. Several systems such as Prolog III [Colmerauer,87], CLP(*R*) [JM,87], CHIP[DVSAGB,88], 2LP [MT,95], and the ILOG Optimization Suite (ILOG Planner) [ILOG-OPT,98] [ILOG-Plan,98] incorporate a linear solver. From the standpoint of constraint programming, the techniques of LP and MIP are special cases of the general CP framework. They provide the means to obtain efficient solutions for a specific context. Yet, linear solvers share many of the same attributes with finite domain solvers. For example, both require search for the use of logical constraints and integer variables. In addition, the preprocessing used in MIP called *bound tightening* is a form of domain reduction.

2.1.2.3 Hybrid Methods

In many cases, a given type of solver is best suited for a specific class of problems. However, there exist problems with many varying attributes such that no specific solver is best suited. In these cases, different solvers may be used in cooperation to form what is know as *cooperating solvers* [DB,95] [da Silva,98] [MTW,98] [RW,98]. For example, most industrial problems involve a mix of linear constraints, logical constraints, and competing preferences and objectives. In these cases, a finite domain solver could be used in conjunction with a continuous domain solver such as a linear programming library. The finite domain solver can handle the logical constraints and search strategies while the linear programming library handles the linear constraints. In addition, communication can be facilitated between the solvers thus improving bounds, strengthening constraints, and ultimately resulting in accelerated solutions. The power of cooperating solvers is becoming increasingly recognized. It has found its way into industrial CP tools such as the ILOG Optimization Suite, see section 2.3.8, where cooperating solvers are internally embedded.

2.1.2.4 Search

Regardless of the solving technique used, constraint programming often requires the use of a search. A *search* is a method of problem solving defined by sets of states. A search problem consists of:

- a state space
- a set of initial states
- a set of allowable operations associated with each state
- a set of goal states (a subset of the state space) that represent solutions

The goal of a search is to traverse the state space starting from one or more initial states using allowable operations to generate new states until a goal state is finally generated. In constraint programming, a state is a CSP. It consists of a set of decision variables and a set of constraints. An allowable operation on a state is a domain reduction of a decision variable or the addition of a new constraint. Performing one of these operations on a given state creates a new state, a new CSP that is a subproblem of the original.

The status of a state is often described through its decision variables. A decision variable can be bound or unbound. A *bound* decision variable is one whose domain contains only one value, and thus the decision variable is fixed and equal to the value at that given point in time. An *unbound* variable is one whose domain contains more than one value. Thus, it is a variable that has not been solved at that given point in time. At any point in time during a search, decision variables can be bound or unbound. Their collective arrangement represents the status of the given state.

In constraint programming, the set of goal states consists of the states where all decision variables have been bound and all the constraints have been satisfied, i.e. a CSP solution. For a CSOP, solving requires accounting for all possible values for the decision variables that satisfy the constraints, i.e. accounting for all goal states. For CSPs solving requires finding any possible value for the decision variables that satisfy the constraints, i.e. finding any goal state.

It is common to represent a search problem by a *search tree*, where *nodes* are states and the top level of the tree is the set of initial states. A successor node is called *child* and a node with no children is called *leaf node*. A node is a leaf, if it does not lead to a solution or it is a *goal node*, (a goal state) a solution. A node that has children is called a *parent node.* If a parent has *b* children, the parent is said to have a *branching factor* of *b*. If every parent node has *b* children, then the search tree is said to have a branching factor of *b*. The resulting search tree is often referred to as the *search space*. The search consists of traversing the tree in pursuit of a goal node. In many cases, nodes can be avoided in the traversal, by *pruning*, not considering nodes that can not lead to a solution.

There are several strategies to traverse a search tree. In CP, the most common method used is labeling with chronological backtracking. *Labeling* attempts to incrementally extend a node by fixing a decision variable at one of its possible values and then propagating its consequences, thus creating a new node. If the new node violates the posted constraints or an inconsistency is otherwise detected, backtracking is performed to the most recent node that still has alternatives.

There are two decisions to be made when performing a search via labeling: which variable is to be fixed (*variable-ordering*) and which value it should be fixed at (*value-ordering*). Choosing the proper ordering strategies can greatly improve efficiency of the search. One of the most commonly used strategies for variable ordering is the *fail-first strategy* [HE,80]. Here the decision variables with the fewest possible remaining alternatives (the fewest elements in their domains) are selected first. These variables usually are the most difficult to satisfy and can cause failure "early and often" in the search. The rationale here is that detecting early failures will be beneficial in the end since, if failure is inevitable, failing early will save wasted time.

Once a variable is selected for branching, a value must be chosen for it. The order in which these values are tried can improve efficiency, particularly in a CSP. On the other hand, CSOPs, CSPs that require finding all solutions, and problems with no solution, may not improve significantly by value ordering since all remaining values must be accounted for anyway. One common strategy for value-selection is the succeed-first strategy. The *succeed-first strategy* tries to pick a value for a variable that is most likely not to have a conflict. The rationale here is that by completing past instantiations we can reduce backtracking.

2.2 History

2.2.1 From Logic Programming to CLP

Constraint Programming is a descendent of logic programming that has emerged from the collaboration of several research communities including Artificial Intelligence (AI), Computational Logic, Programming Languages, and Operations Research (OR). It can be traced back to the early 1960’s. One of the pioneering systems was Sketchpad [Sutherland,63], an interactive drawing system allowing the user to build geometric objects from language primitives and certain constraints. Other pioneering systems were Laurière's ALICE [Laurière,78], Steele's CONSTRAINTS [Steele,80], and Borning's ThingLab [Borning,84].

The idea of declarative programming was inherited from logic programming and can be traced back to the early days of Prolog, a programming language based on a subset of first-order logic. Prolog represented a fundamentally new approach to computing, as compared to more traditional procedural languages, such as BASIC, Fortran, or Pascal. It shifted program development away from problem solving to problem formulation, using highly declarative logic statements. This made Prolog a natural choice for developing intelligent systems, expert systems, and deductive database systems. Prolog's actual and alleged inefficiencies caused researchers to seek more efficient methods, resulting in the development of Constraint Logic Programming (CLP) [JL,87]. CLP was an extension of logic programming that aimed at replacing unification (used by Prolog) with a more general operation, *constraint satisfaction*, a black-box approach that used mathematical tools such as simplex to deal with numerical constraints and consistency checking and constraint propagation techniques to handle symbolic constraints. Although, constraint satisfaction improved performance, disadvantages became evident because of the black-box approach. A *black-box* approach abstracts a user from the solving method and thus, the computation that led to the computed answer may not be fully understood. In addition, the techniques used for solving within the black-box may not the right ones to solve a given problem efficiently. Thus, new approaches have been proposed to overcome the black-box shortcomings. These newly developed methods were referred to as glass-box and no-box approaches. A *glass-box* approach provides several solving methods, allowing the user to choose the method of solving which is best suited for a given problem or given constraints, leading to a more efficient solution. One such system that incorporated a glass-box approach was CHIP [DVSAGB,88], a Prolog-like programming language that combined consistency techniques for finite domains, equation solving for Boolean algebra and an algorithm for numeric constraints over rational numbers. Although a glass-box provides flexibility, it does not allow the complete tailoring of a solution approach. This has led to the *no-box* approach, where a user specifies a set of constraint handling rules, which implement a solver explicitly [Frühwirth,95]. This was also used in the CALOG framework [Wetzel,97].

2.2.2 CP = Effective Modeling + Efficient Solution

During the development of CLP, alternative paths were being taken by researchers. Researchers experimented with different types of constraints and solving techniques. Some tried to extend the functionality of Logic Programming by adding new features, such as in the case of Prolog III [Colmerauer,87]. Others tried to break away from the logic approach and founded the more general idea of CP, represented in the ideas of effective modeling and efficient solution. The ideas were simple. CP should allow easy and natural problem formulation while providing the most efficient solving methods. This led to the development of a wide range of CP software tools, some of which are presented in the next section. These new approaches used techniques such as object oriented modeling to improve problem formulation and a combination of Operations Research and Artificial Intelligence techniques to improve the efficiency of solving.

2.3 Constraint Programming Software Tools

Most often, a CP application is a component of a multifaceted application. For example, consider an application for an oil refinery, that consists of a CP application that schedules the flow of crude oil to tanks, a hardware control application that moves oil from pipelines to tanks (as given by the output of the CP application), and a graphic interface that shows the movement of the oil and the current capacities of the oil tanks. Now, consider the development of such an application. Each component must be designed, implemented, and tested. However, in most cases, time and cost constraints restrict the components from being built from scratch. Thus, the project is dependent on software tools for rapid development. In the case of the CP, several such tools exist.

CP software tools are most common in the form of new programming languages or software libraries written for current programming languages (like C++), each approach having advantages and disadvantages. New programming languages can create new constructs that allow for a more natural problem formulation. However, a new programming language requires learning a new syntax and thus, can have a steeper learning curve. In addition, new languages often do not have comparable support or performance of a mature language such as C or C++.

Libraries, on the other hand, allow a user of the underlying programming language to learn a tool quicker. For the library-based tool, the user need only learn a few new functions provided by the library. However, library constructs are restricted by the underlying programming language, which in some cases can make problem formulation awkward. However, the advent of object-oriented languages such as C++ has helped make problem formulation more natural. Presented in the next sections is a collection of CP software tools.

2.3.1 CHIP

Constraint Handling in Prolog (CHIP) [DVSAGB,88] is a CLP language that combines the declarative aspects of Prolog with the efficiency of specialized constraint solving techniques. It was originally developed at the European Computer-Industry Research Center in Munich (ECRC) and is currently being marketed by Cosytec S.A., a Paris based company. CHIP is a superset of Prolog that combines techniques from different paradigms in order to allow powerful symbolic and numerical constraint manipulation. It was one of the first CLP languages to incorporate cooperating solvers. It combined consistency techniques for finite domains, equation solving for Boolean algebra, and an algorithm for numeric constraints over rational numbers.

2.3.2 ECL^{i}PS^{e}

ECL^{i}PS^{e} [WNS,97] is a CLP platform that is based on the work of CHIP. It is currently being further developed at London's Imperial College at IC-Parc, a university-industry collaboration that tackles hard planning and scheduling problems. ECL^{i}PS^{e} consists of a modeling language, a platform on which to run models, and several solving libraries for handling symbolic and numeric constraints. The modeling language has the same syntax as Prolog. However, the solution methods used are different from Prolog and therefore, ECL^{i}PS^{e} avoids some of Prolog's inefficiencies. In addition, unlike Prolog, ECL^{i}PS^{e} has elements of a glass-box approach to solving, thus allowing a user to use a combination of solving and propagation techniques for the problem at hand. Early versions of ECL^{i}PS^{e} have been internally parallelized by using distributed parallel computing [MS,94] and now provide limited user-level parallelism on shared-memory multiprocessors. The scheme uses a worker-manager paradigm. The user can specify how many workers (processes), can be used for a given problem, and can add or remove workers at any given time. The user specifies what part of the code should attempt to execute in parallel by adding parallel annotations before a section of code. However, the user can not control the method of work division, work assignment, or the type of parallel search strategy used. Instead, the system controls parallelism automatically.

2.3.3 LAURE

LAURE [Caseau,91] is an object-oriented CLP language based on sets and relations. It provides declarative semantics based on logic and a set of algorithms for resolution. LAURE grew from the work of relational deductive databases. Knowledge in the system is represented by a collection of binary relations stored as objects in a database. The database can be queried or updated by using the provided object-oriented logic-based language.

2.3.4 CLAIRE

CLAIRE [CL2,96] is a functional and object-oriented CP platform with advanced rule processing capabilities. It includes an interpreter, compiler, and a set of tools that works both on compiled and interpreted code. CLAIRE was inspired from the work of LAURE. It attempted to contain many of the design features of LAURE with added simplicity and C++ compliance. CLAIRE has had great success with scheduling problems such as the Job Shop Problem [CL1,96].

2.3.5 clp(FD)

The clp(FD) [DC,93] system consists of a constraint logic programming language (over finite domains) and the wamcc [Diaz,91] Prolog compiler, which translates Prolog to C via the Warren Abstract Machine [Warren,83]. It was one of the first systems to provide methods for defining high-level constraints. This was achieved by using a single primitive constraint as the core propagation mechanism, as was first seen in the concurrent system of cc(FD) [VSD,91]. In clp(FD), a high-level constraint is defined in terms of the primitive constraint. At compile-time, the high-level constraint is translated into a combination of the primitive constraints. This set of primitive constraints determines the propagation scheme for solving the high-level constraint.

2.3.6 OZ

OZ [HSW,93] is an object-oriented CP language that combines constraint programming with concurrency. Over time, OZ has undergone a series of functional changes. The first version, OZ 1 [HSW,93], supported concurrency implicitly by creating threads automatically without any input from the programmer. The next version, OZ 2, changed this design by adding new constructs that allowed a programmer to explicitly create threads, see [Haridi,97]. Although both early versions of OZ supported concurrency through threads, they did not support parallelism through threads. Namely, the threads supported and provided by OZ did not execute in parallel on multiprocessors. Instead, the threads were interleaved on a single processor. However, the latest version OZ 3, which is embedded into a programming system called Mozart, supports parallelism. Specifically, OZ 3 supports distributed parallelism, which is provided through the addition of new constructs that control the creation and communication of OZ processes, see [MOZART].

2.3.7 2LP

The 2LP system [MT,95], "Linear Programming and Logic Programming", consists of a CP language with a C-like syntax and an interpreter. The system was developed in the early 1990's at the Logic Based Systems Lab at Brooklyn College of the City University of New York. The 2LP language is both an algebraic modeling language and a logical control language. It combines the solving methods of linear programming with the declarative nature of logic programming. This allows the natural and easy problem formulation associated with logic programming to be combined with the powerful and proven solution methods of simplex-based linear programming. There have been parallel implementations of 2LP. An early work [Atay,92] used parallel distributed computing to internally parallelize 2LP, thus making parallelization transparent to the user. The next effort Parallel Integer Goal Programming [AMT,95], provided an intermediate level of internal parallelization in a distributed environment. Specifically, 2LP was integrated with DP [Arnow,95], a message passing library, to address the computational demands of integer goal programming. This work included the addition of five new instructions to 2LP. These new instructions allowed a programmer to create distributed applications without concern for process creation, synchronization, or load balancing. Instead, these tasks were internally handled by a support library. [Muller,97] extended the previous work to multithreading. This included making 2LP thread-safe and providing new multithreading facilities through the addition of new instructions.

2.3.8 ILOG Optimization Suite

The ILOG Optimization Suite is a set of software components that are used to model and solve computationally demanding problems. Its design incorporates many glass-box features such as allowing the programmer to specify specific search strategies. The suite consists of a core component and three add-on components, all of which take the form of C++ object libraries. Each library defines several C++ "constrained" variable classes and the methods to model and solve them. Each component is seamlessly integrated with the others to facilitate easy collaboration between different components.

ILOG Solver [Puget,94] is the core component of the Optimization Suite. It implements the basic engines for constraint programming. The add-on components are built on Solver. They incorporate additional constraints, constraint variables, and solving algorithms for specific types of problems. The add-ons are:

- **ILOG Planner:** Provides facilities for linear constraints solved with simplex-based and Barrier-based algorithms incorporated in an object-oriented interface.

- **ILOG Scheduler:** Provides facilities for handling time, sequencing tasks, and managing scheduling problems.

- **ILOG Dispatcher:** Provides facilities for vehicle routing problems.

ILOG's interface includes an object called a *manager* that provides methods to manage CP models. A manager handles most of the internal issues of the solving engine, such as initializing internal data, input, and memory allocation. In addition, decision variable declarations, constraint postings, and goal postings must be performed through a manager. Therefore, associated with each manager is an instance of the solving engine containing a problem formulation, which includes decision variables, constraints, and goals. A manager also has a built-in "glass box" backtracking search facility. The "glass box" feature allows the application programmer to guide the search by controlling choices such as the selection of branching variables or executing a function at a given event. A CP application can consist of one or more managers, each having its own variables and constraints. Managers are completely separate entities and thus can not share variables or constraints. In addition, a manager can only work on one problem at a time. A manager can be created in one of two modes, "edit mode" or "search mode". The edit mode allows a user to add or remove constraints as required by a CP application. While a manager is in edit mode, constraint propagation is postponed. To start propagation, the manager must be placed in search mode. This is performed through a member function (of the manager). While in search mode, constraints are propagated and thus can not be removed. Furthermore, if a manager is started in search mode, constraints are posted incrementally.

Chapter 3

Multithreading

3.1 Symmetric Multiprocessing

Symmetric multiprocessing (SMP) machines are becoming increasing popular because of their low cost-to-performance ratio. A *symmetric multiprocessing architecture* is a tightly coupled multiprocessor system, where processors share a single copy of the operating system (OS) and resources that usually include a common bus, memory, and I/O system, see Figure 3.1. The term "symmetric" refers to the fact that all processors in the system are functionally equivalent and thus each can perform any task. This gives SMP systems a distinct advantage over asymmetric designs when it comes to load balancing since the system can schedule a task to any available processor and rapidly migrate it to other processors.

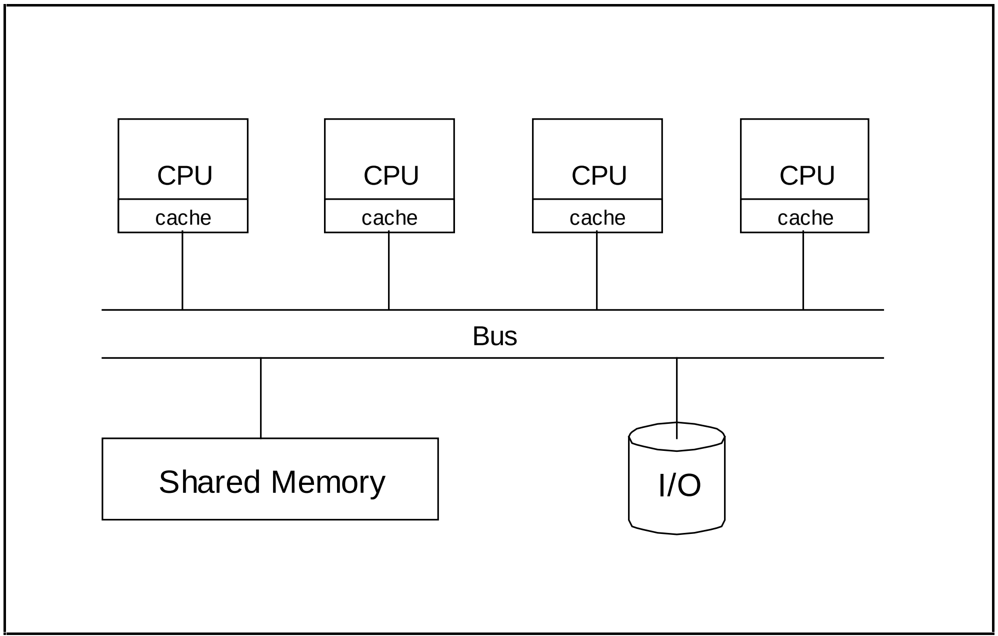

Figure 3.1: A basic symmetric multiprocessor architecture.

3.2 Threads

In order to exploit concurrency and parallelism, the OS of a SMP machine needs to provide facilities to take advantage of the underlying architecture. Operating systems of SMP machines accomplish this by incorporating multitasking and multithreading facilities. In the classic case, a *multitasking* operating system divides the available processor time among the processes of the system to increase process throughput. While executing, a process has only one unit of control. Thus, the process can only perform one task at a time. SMP operating systems divide the classical notion of a process into smaller execution objects usually referred to as *threads*, see Figure 3.2. The actual name is operating system dependent, as we shall see in the next section. Threads are the basic entities to which the OS allocates processor time. A process of a SMP OS can contain one or more threads. Each thread has its own *context*[1], yet it shares address space and resources, such as open files, timers, and signals, with other threads of the same process. This design permits threads to function independently while keeping cohesion among the threads of the same process. This creates many benefits:

- **Inexpensive:** As a result of sharing resources, the time to create a thread is generally 5 to 30 times faster than creating a new process and the time to perform a context switch for two threads of the same process is faster than performing a context switch between two threads of different processes [SUN-MPG,94].

- **Overlap Processing:** Since each thread has its own context, each can be separately dispatched and scheduled by the operating system kernel to execute on a processor. Therefore, a process can have one or more units of control. This enables a process with multiple threads to overlap processing. For example, one thread could continue execution while another is blocked by an I/O request or synchronization lock.

- **Concurrent Execution:** On a multiprocessor machine, a process can have threads execute concurrently on different processors. Thus, a computation can be made parallel to achieve speedup over its serial counterpart.

---

[1] This refers to its state, defined by the values of the program counter, machine registers, stacks, and other data.

- **Communication:** Sharing the same address space allows threads of the same process to easily communicate by using shared global variables.

- **Scalability:** Once a program is written with threads, it can execute with a single thread on one processor or multiple threads on multiprocessors. Thus, the code does not have to be rewritten if more processors are added in the future.

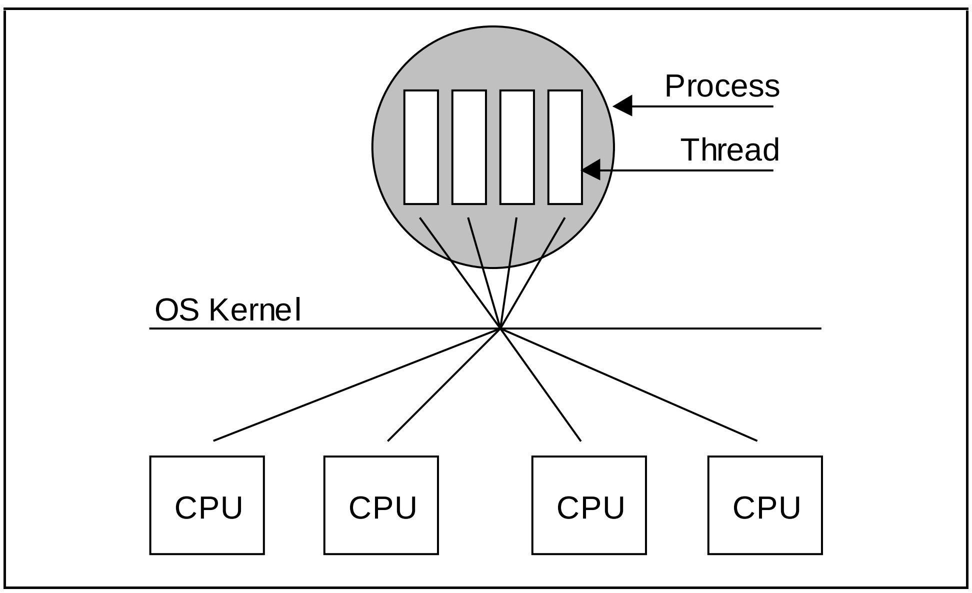

Figure 3.2: **A basic multithreaded process model.**

3.3 Two Thread Models

Although, the basic idea is the same, thread implementations can vary from one OS to another. These differences can be as simple as terminology or as complex as physical processor mapping. To illustrate this point we will take a closer look at the thread implementations of Windows NT and Solaris, two mainstream operating systems.

3.3.1 NT’s Threads and Fibers

A *thread* is Windows NT’s smallest kernel-level object of execution. Processes in NT can consist of one or more threads. When a process is created, one thread is generated along with it, called the *primary thread*. This object is then scheduled on a system wide basis by the kernel to execute on a processor. After the primary thread has started, it can create other threads that share its address space and system resources but have independent contexts, which include execution stacks and thread specific data. A thread can execute any part of a process' code, including a part currently being executed by another thread. It is through threads, provided in the Win32 application programmer interface (API), that Windows NT allows programmers to exploit the benefits of concurrency and parallelism.

In addition to threads, Windows NT also provides fibers. A *fiber* is NT’s smallest user-level object of execution. It executes in the context of a thread and is unknown to the operating system kernel. An NT thread can consist of one or more fibers as determined by the application programmer. Some authors, such as [BL,96] [SUN-MIC,96], assume that there is always a one-to-one mapping of user-level objects to kernel-level objects. However, this is not always the case. Windows NT does provide the means for many-to-many scheduling. However, NT's design is poorly documented and the application programmer is responsible for the control of fibers such as allocating memory, scheduling them on threads, and preemption. This is different from Solaris's implementation, as we shall see in the next section. See [MSDN] [Richter,94] for more details on fibers. An illustrative example of NT's design is shown in Figure 3.3.

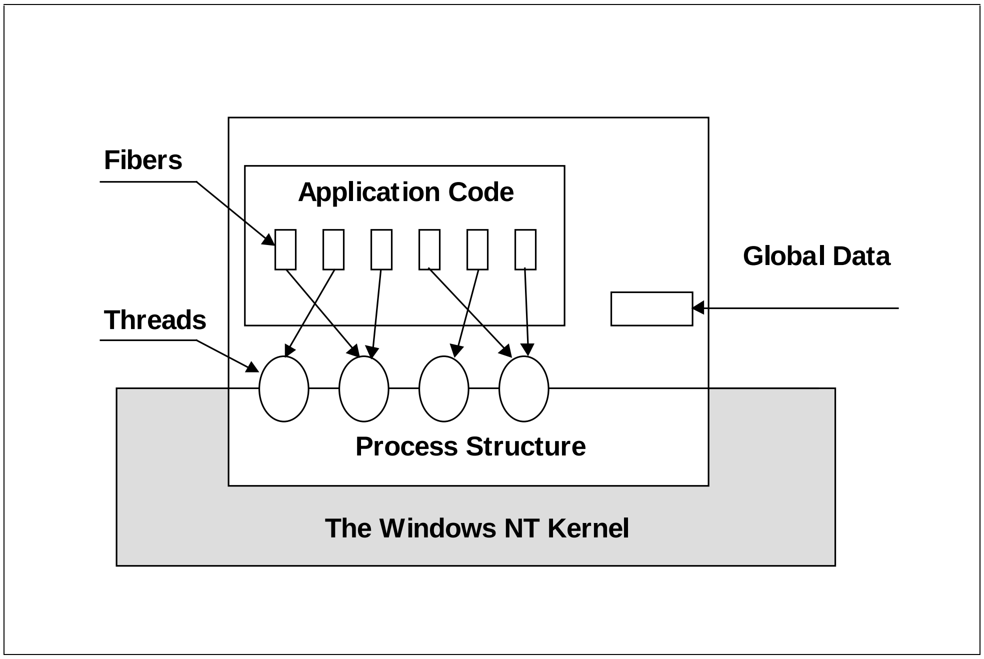

Figure 3.3: The relationships of a process and its threads and fibers in Windows NT.

3.3.2 Solaris's LWPs and Threads

A *light weight process* (LWP) is Solaris's smallest kernel-level object of execution. A Solaris process consists of one or more light weight processes. Like NT's thread, each LWP shares its address space and system resources with LWPs of the same process and has its own context. However, unlike NT, Solaris allows programmers to exploit parallelism through a user-level object that is built on light weight processes. In Solaris, a *thread* is the smallest user-level object of execution. Like Windows NT's fibers, they are not executable alone. A Solaris thread must execute in the context of a light weight process. However, unlike NT's fibers which are controlled by the application programmer, Solaris's threads are implemented and controlled by a system library. The library controls the mapping and scheduling of threads onto LWPs automatically. One or more threads can be mapped to a light weight process. The mapping is determined by the library or the application programmer. Since the threads execute in the context of a light weight process, the operating system kernel is unaware of their existence. The kernel is only aware of the LWPs that threads execute on. An illustrative example of this design is shown in Figure 3.4.

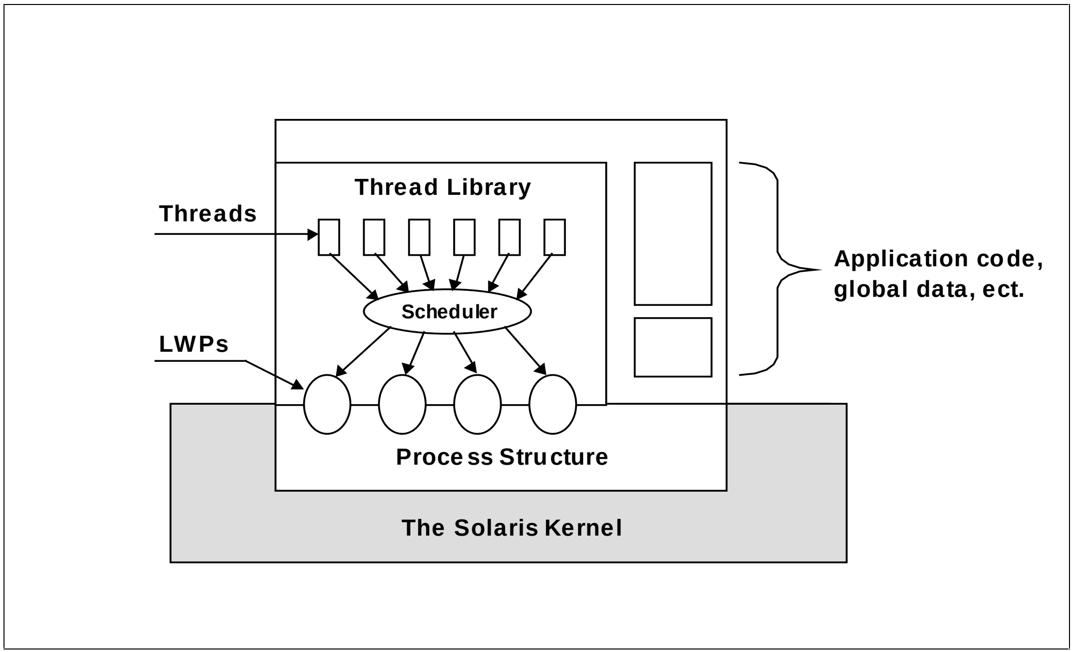

Figure 3.4: The relationships of a process and its LWPs and threads in Solaris.

Solaris's thread library defines two types of threads according to scheduling. A *bound* thread is one that permanently executes in the context of a light weight process in which no other threads can execute. Consequently, the bound thread is scheduled by the operating system kernel on a system wide basis. This is comparable to an NT thread.

An *unbound* thread is one that can execute in the context of any LWP of the same process. Solaris uses the thread library for the scheduling of these unbound threads. The library works by creating a pool of light weight processes for any requesting process.

The initial size of the pool is one. The size can be automatically adjusted by the library or can be defined by the application programmer through a programmatic interface. It is the library's task to increase or decrease the pool size to meet the requirements of an application. Consequently, the pool size determines the concurrency level of the process. The threads of a process are scheduled on LWPs in the pool, by using a priority based, first-in first-out (FIFO) algorithm. The priority is the primary algorithm and FIFO is the secondary algorithm (within the same priority). In addition, a thread with a lower priority may be preempted from a LWP by higher priority thread or by a library call. In the case where all threads have the same priority, the scheduling algorithm is solely FIFO. In this case, once a thread is executing, it will execute to completion on a light weight process unless it is blocked. If blocked, the library will remove the blocked thread from the LWP and schedule the next thread from the queue on that LWP. The new thread will execute on the LWP until it has completed or been blocked. The scheme will then continue in the same manner. For more information on Solaris's design and implementation see [BL,96] [KSS,96] [SUN-MA,91] [SUN-MPG,94].

3.4 Synchronization

Since threads can execute concurrently, synchronization is required to prevent resource and data conflicts. For example, one thread could be updating the contents of a structure while another thread is reading the contents of the same structure. It is unknown what data the reading thread will receive, the old data, the newly written data, or possibly a mixture of both. Most operating systems provide synchronization primitives to prevent this from occurring. The provided primitives are most commonly in the form of new data types and associated functions. In most cases, the variables must be explicitly initialized and destroyed through the provided functions. Some basic synchronization variables are described in the following three subsections.

3.4.1 Mutex

A mutex is a data structure that grants one thread temporary exclusive access to a shared resource. A mutex has two states, *locked* and *unlocked,* that are set by the atomic operations *LOCK* and *UNLOCK*. For each shared resource, all threads accessing that resource must use the same mutex. When a thread requires access to a resource it must first LOCK the mutex and upon completion, it must UNLOCK it. If the mutex is in the locked state, a requesting thread either waits for the mutex to be restored to the unlocked state or returns with an error code, depending on the lock routine.

3.4.2 Semaphore

A semaphore is a data structure that allows a limited number of threads to access a resource. A semaphore is useful in controlling access to a shared resource that can only support a limited number of users. It consists of an integer that can be accessed through two atomic operations: *wait* and *signal* (classically known as *P* and *V).* The signal operation increments the integer and the wait operation decrements the integer. If the integer is equal to zero, a thread calling the wait operation will wait until the integer is greater than zero and then decrement it. It should be noted that a semaphore is a shared resource and thus its use requires mutual exclusion. Therefore, a semaphore must be used in conjunction with a mutex.

3.4.3 Condition Variable/Event

A condition variable/event is a data structure that allows one thread to notify another that an event or condition has occurred. A condition variable is useful when a thread needs to know when to perform its task. For example, a thread that copies data from a buffer would need to be notified when new data is available. Like a semaphore, a condition variable is a shared resource and thus it too requires mutual exclusion for its use.

3.5 Concurrent Data Structures

Multithreaded applications require the use of data structures that can function correctly under concurrent operations. A simple way to achieve this is through single lock serialization using synchronization objects such as the ones described in Section 3.4. However, single lock serialization limits the number of concurrent operations to one. This can severely limit the performance gains of parallelization. In order to maximize performance gains from parallelization, data structures must maintain a high level of concurrency. Several researchers such as [BB,87] [RK,88] [Jones,89] [CS,91] [HMPS,96] [MSL,96] have developed concurrent data structures for shared memory architectures that can be directly applied to multithreaded applications. These data structures fall into two categories: blocking and non-blocking. A *blocking* data structure uses synchronization objects based on the "test and set" instruction, such as a mutex. In this case, a thread that requires an operation may be blocked from execution if that operation is not available at the time of the call. A *non-blocking* data structure uses synchronization objects based on the "compare and swap" instruction. In this case, a thread that requires an operation that is currently unavailable, may perform a loop until it is available. However, some architectures do not support the "compare and swap" instruction, and thus to maintain a high degree of portability we only discuss blocking data structures. In the next two subsections, we will focus on two such data structures that are used throughout our work.

3.5.1 Two Lock Concurrent Queue

A *queue* is a first-in first-out data structure that is often implemented with a linked-list. A queue has two basic operations: enqueue and dequeue. The *enqueue* operation adds one item to the tail of the linked-list and the *dequeue* operation removes the item at the head of the list. A two lock concurrent queue [MSL,96] is one that allows enqueue and dequeue operations to be performed simultaneously. This is accomplished by using two locks, one for the head of the list and one for the tail.

3.5.2 Multiple Lock Concurrent Priority Queue

A priority queue is a data structure for storing and retrieving items according to priority. It has two basic operations: insert and delete. The *delete* operation removes the item with the highest priority from the queue and the *insert* operation adds an item to the queue in such a way that priority order can be maintained.

A priority queue is most often implemented with a *heap*, a binary tree implemented with an array. Each node of the tree corresponds to an element in the array. The root of the tree is element 1 of the array and the left and right children of a node i is represented by elements $2i$ and $2i+1$ of the array. The tree is completely filled on all levels except for possibly the lowest. Thus, level l is empty if level l-1 is not full. To represent a priority queue the heap guarantees that a parent node has a higher priority than any of its children and thus, the root node has the highest priority of the entire heap.

A multiple lock concurrent priority queue [HMPS,96] maintains one lock for each element in the array and a single lock for the entire priority queue structure. This allows deletions to occur simultaneously with insertions and insertions to occur simultaneously with other insertions. However, deletions can not occur simultaneously with other deletions.

3.6 Multithreaded Cross-Platform Development

Multithreaded applications complicate cross-platform development since threads and their synchronization objects are operating system dependent. The next four subsections give some solutions to the problem of multithreaded cross-platform development. These include the used of external wrapper libraries and custom coding.

3.6.1 Pthreads

IEEE has defined a thread standard POSIX 1003.1c-1995 that is an extension to the 1003.1 Portable Operating System Interface (POSIX). The standard, called *Pthreads*, is a library-based thread API. It allows one to develop thread applications cross platform. However, IEEE does not actually implement the library. It only defines what should be done. This leaves the implementation of Pthreads up to the operating system developer. In most cases, the Pthreads library is built on the developer's own thread implementation. It is simply a wrapper over the developers' own implementation and thus, all features may or may not exist. In the case where the OS does not have a thread implementation, the library is solely user based, and thus can not exploit multiprocessors.

3.6.2 ACE Threads

The ADAPTIVE Communication Environment (ACE) is free open source library that provides an object-oriented environment for cross platform concurrent application development [Schmidt,94]. ACE provides a set of reusable C++ wrappers for concurrency, including object classes for multithreading and synchronization [Schmidt,95] across a wide range of platforms. Thus, using ACE increases code portability and reusability. In addition, ACE's design, allows object-oriented applications to maintain design consistency throughout the application. This is because most operating systems' thread and synchronization APIs are written in C which would require mixing different programming paradigms.

3.6.3 RogueWave Threads.h++

Threads.h++ [RogueWave] is a commercial C++ library that provides a platform-independent threading model. In terms of functionality, it is similar to ACE threads. It hides the differences among different platforms and provides a common set of portable object-oriented classes that are intended to simplify the development of cross-platform multithreaded libraries and applications. However, unlike ACE threads, it also provides additional classes for managing and distributing threads and their workloads.

3.6.4 Custom Development and Preprocessor Directives

Of course, developers can write their own wrapper libraries to solve the portability issue. This has the benefit of allowing developers to customize their libraries to meet the specific demands of their applications. However, library development may take a considerable amount of time. Another option is to use a compiler's preprocessor to insert the correct API for the system on which it is being compiled. However, this can make the code less readable and may be problematic for long term maintenance. For instance, if a thread API changes, the whole application must be rewritten.

Chapter 4

Dynamic Thread Creation

4.1 Multithreaded Parallelism

Many computations can be parallelized by dividing the computation among several threads. Since each thread can execute concurrently, the computation can be performed in parallel if executed on a multiprocessor. However, not all parts of a computation are parallelizable. Thus, some portions of the computation must execute sequentially. Clearly, as *Amdahl's Law* states, the performance to be gained by parallelization is limited by the fraction of time that parallelization can be used.

A common measure of performance for a parallel computation is *speedup*, the performance gain achieved by the parallel computation over its sequential counterpart. A goal of making a computation parallel is to achieve a speedup that is linear or near linear in the number of processors (threads) being used. The actual speedup that is attainable by making a computation parallel is a function of how the computation is divided and the amount of parallel overhead. *Parallel overhead* is defined as the ratio of work done by the parallel formulation to that done by the sequential formulation. This is shown in Equations 4.4 and 4.5.

Let

$$N = \text{Number of threads (processors)} \tag{4.1}$$

$$SF = \text{Work done by a sequential formulation} \tag{4.2}$$

$$PF = \text{Total work done by a parallel formulation} \tag{4.3}$$

then

$$\textit{Parallel Overhead} = \frac{PF}{SF} \tag{4.4}$$

$$\textit{Theoretical Speedup} \leq N\left(\frac{SF}{PF}\right) \tag{4.5}$$

Parallel overhead can be incurred from many sources such as idle time due to load imbalances, required synchronization, contention for shared resources and computation time for work partitioning. In the case of threads, additional overhead may be incurred from the operating system. Namely, since a thread is a kernel object, it is the task of the operating system to schedule the threads on processors. Thus, a thread is often switched on and off processors by the operating system. These context switches can lead to additional overhead. However, many techniques exist, such as processor binding or priority manipulation that will allow one to fine-tune the OS scheduling to increase the parallel performance. For more details see [Tucker,93] [KSS,96].

4.2 Multithreaded Parallel Search

For a search, work is the exploration of the search space. Thus, to parallelize a search disjoint properties of the search space must be exploited. This usually involves partitioning of the search space. A goal of a partitioning scheme is to divide the search space in such a way that processor utilization will be high and overhead from parallelization will be low. Furthermore, high processor utilization should come from time spent on the search, not time spent dividing or managing the search space.

Partitioning schemes can be grouped into two categories: static and dynamic. A *static* strategy divides the search space before beginning the search. Each thread is given a disjoint part of the search space to explore. After it has explored its space, the thread terminates. This type of scheme has the benefit of requiring little synchronization among threads and minimal computational overhead for making the search parallel. However, search is non-deterministic. Thus, the computational effort that is required to explore a part of a search space is unknown. Therefore, the execution time of two threads exploring two disjoint portions of search space might vary and cause one thread to finish exploring its search space before another, making the processor on which it was executing idle and consequently, leading to a poor speedup. In fact, the non-deterministic nature of search can lead to many anomalies [LS,84] which can adversely effect load balance and speedup. Thus, another option is to divide the search space dynamically so that a thread with no work can obtain work from a busy thread. This is often referred to as *dynamic load balancing.* Several dynamic load balancing schemes exist [BS,81] [WLY,85] [RK,87] [PB,90] [MD,92]. Some dynamic load balancing schemes are *receiver initiated*, i.e. when a processor runs out of work, it makes a request to another processor for work. Other schemes use a *farming model,* where one processor is used to manage the search space while other processors perform tasks that are "farmed out" by the manager. For a complete summary of load balancing schemes see [KGR,94]. Most load balancing schemes were originally designed for distributed systems and are message based. The schemes have been successfully ported to tightly coupled systems by using shared memory and additional synchronization, for sending or receiving work. However, these message-based models are counter intuitive for shared memory systems.

4.3 Dynamic Thread Creation

4.3.1 Introduction

In [ZY2,98], we presented a new parallel search paradigm: an asynchronous method of dynamic load balancing that exploits the inexpensive nature of thread creation while providing a model that is natural for symmetric-multiprocessors. The idea is simple, yet effective in obtaining linear or near linear speedups for parallel searches. The scheme differs from the common message based models by eliminating the general communication paradigms that are required to perform load balancing. This creates a level of abstraction that minimizes the management of computational resources and largely avoids synchronization.

4.3.2 Thread Creation

We begin by recalling that the cost of creating a thread is inexpensive as compared to creating a process [SUN-MPG,94]. It has also been shown that the computational time for thread creation is less than the time required to suspend and then reactivate a thread

[BL,96]. Therefore, creating and destroying threads as they are required is usually better than managing a pool of threads that wait for independent work [BL,96]. Moreover, the aim is to create new threads for computations, whose execution time is large enough to compensate for the *overhead of thread creation*, the sum of time it takes to partition the search space, create a new thread, and initialize the new thread's state (see Section 5.5.2).

4.3.3 Active Thread Limit

A thread that is currently executing to perform an assigned task is called an *active thread*. To obtain a theoretical maximum speedup for a computationally demanding application on a given machine, the number of active threads should be equal to the number of processors of the machine [ZY1,98]. Using fewer threads than processors results in the processors being underutilized and using more threads than processors results in overhead (context switches) from the processors being overloaded [ZY1,98]. It should be noted that there have been cases where a greater number of threads than processors have provided better performance in search applications [Muller,97]. However, from [RK,93] we can derive that for the average case, excessive thread creation would weaken performance. We therefore impose an upper limit on the number of active threads that a process can contain. We refer to that limit as the *active thread limit* (ATL). For practical purposes, the ATL is user defined. Thus, the application programmer can specify the maximum number of active threads, which in effect specifies the maximum concurrency level of a process. This is somewhat similar to setting the concurrency level in Solaris threads, as described in Section 3.3.2. However, it should be noted that the ATL does not represent the number of threads to be used. It solely represents a limit on the number of active threads at any one given time. Thus, the total number of threads created and used could be greater than the ATL.

To gain a better understanding of the ATL, let us consider a system with four processors and various values for the ATL. Since there are four processors, the ideal ATL value is four, i.e. one thread for each processor. If the ATL is set to a value less than four, some processors will not be utilized since the maximum number of active threads is less than the number of processors. However, there is a benefit to setting the ATL to a value less than the number of processors. Namely, doing so provides a method to run applications without fully monopolizing all the resources of a machine. For example, in the case of a four-processor system, setting the ATL to three leaves one processor free for other uses. If the ATL is set to a value greater than four, the number of active threads will be greater than the number of processors and thus, addition overhead will be incurred from the operating system performing context switches. This is analogous to having multiple processes time-sliced on a single processor machine.

4.3.4 The Algorithm

The algorithm begins by creating one thread with the complete search space to explore. The task of the thread is to explore the search space, using any strategy, for potential solutions. If the active thread limit is not reached and the work is large enough to be shared, the thread partitions the given search space into two disjoint portions using any type of work splitting strategy, such as the ones described in [RK,87]. It will then create a new thread so each can work on one of the disjoint portions. Then the above scheme is applied to the newly created thread. A general version of the algorithm is illustrated in two flowcharts shown in Figures 4.1 and 4.2. Figure 4.1 is the flowchart of "main" and Figure 4.2 is the flowchart of a thread.

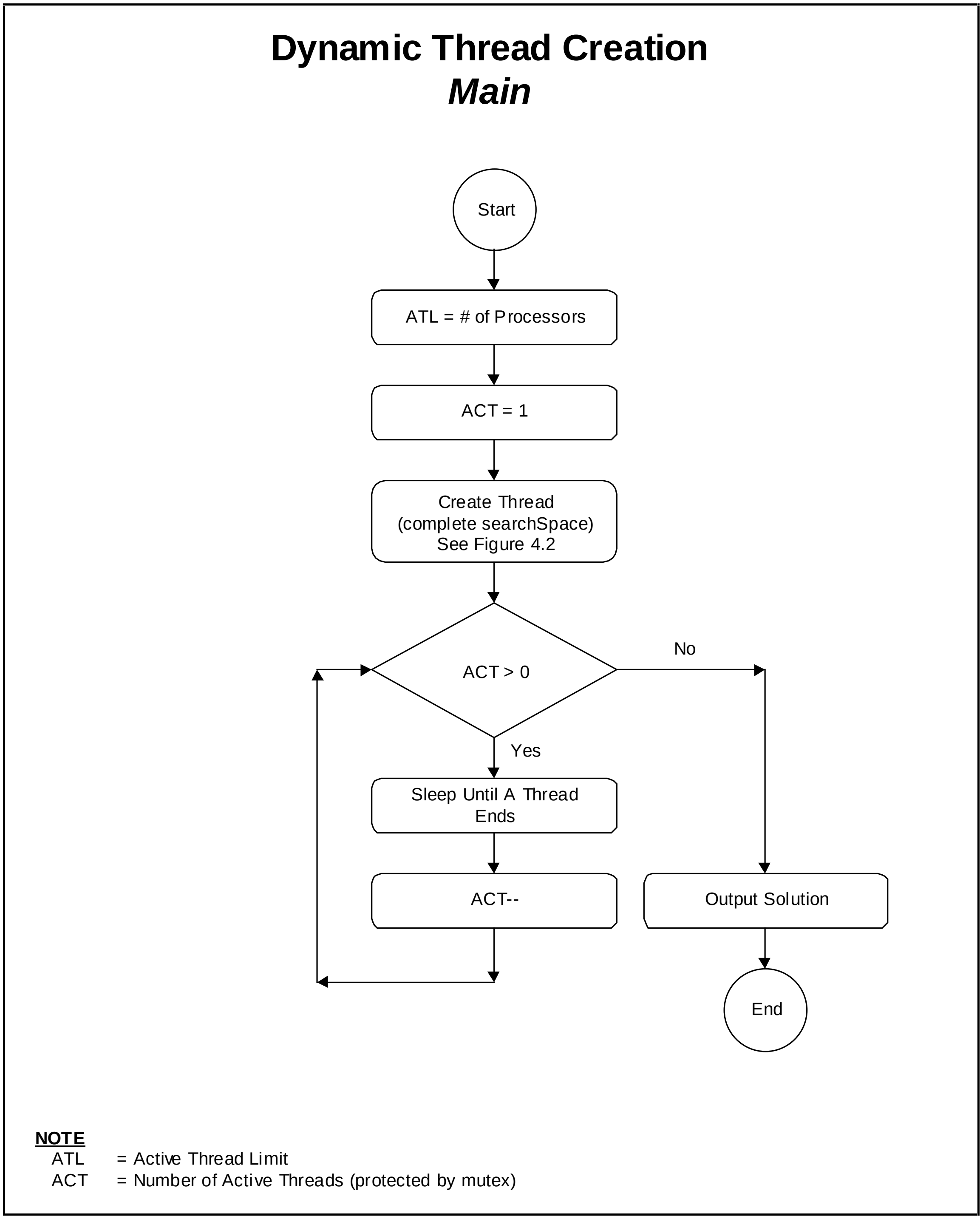

Figure 4.1: Flowchart of "main" for Dynamic Thread Creation load balancing.

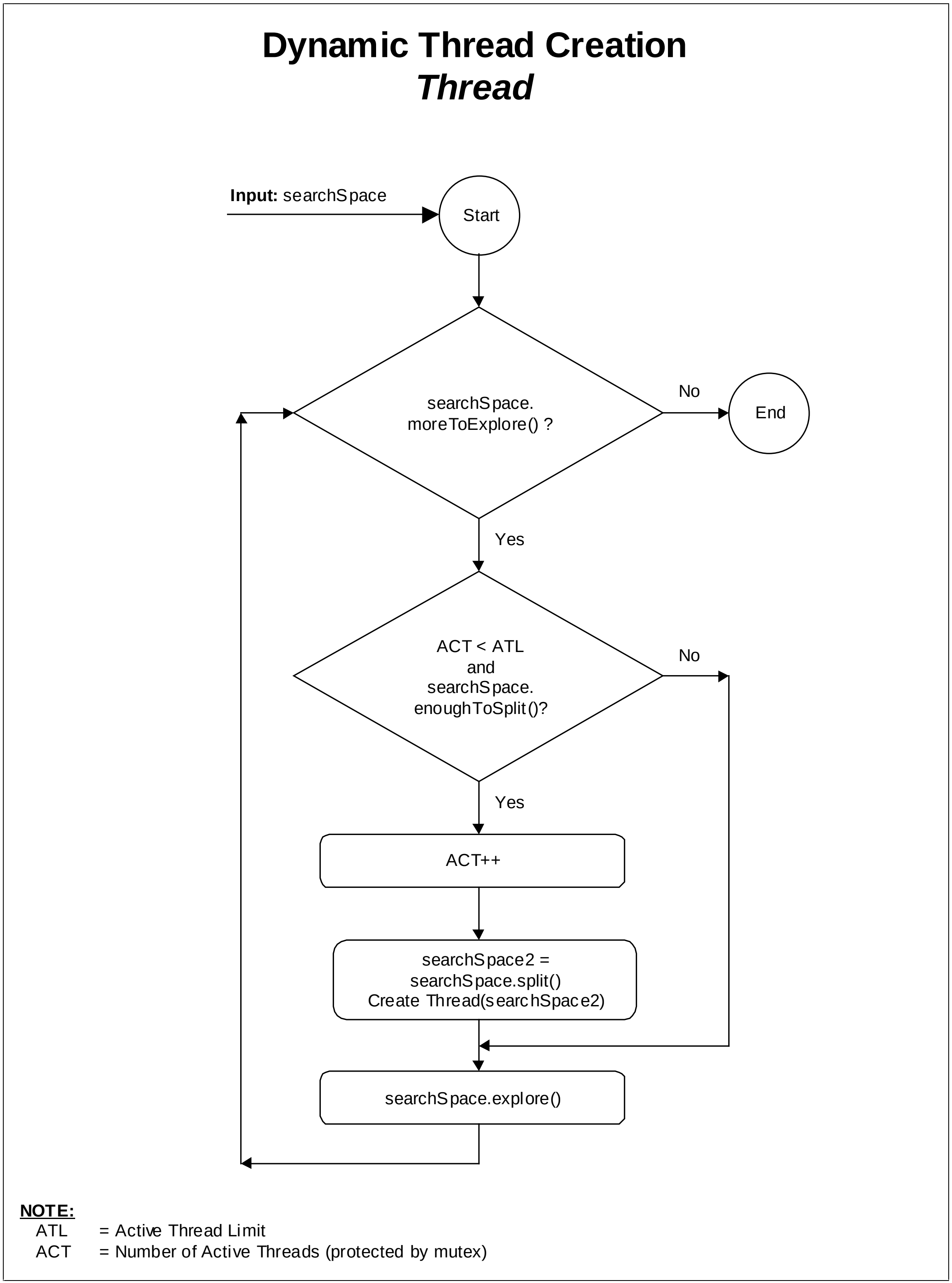

Figure 4.2: Flowchart of a thread for Dynamic Thread Creation load balancing.

The scheme allows the initial search space to be evenly distributed among the active threads as the number of active threads reaches its limit. The threads then search, terminate, or create new threads in an asynchronous fashion. The initial work division and the time to reach full parallel capacity are accomplished in logarithmic time. This is different from most static load balancing schemes, where the initial search space is sequentially partitioned prior to parallel execution. Yet like static schemes, it has the benefit of requiring little synchronization.

After a thread has finished searching its allocated space, it terminates and thus the number of active threads is reduced, allowing an active thread to create a new thread. This is very different from other schemes such as in message passing, where a free thread would either request work, steal work [BLL,94] or be assigned work. Instead, the inexpensive cost of thread creation is exploited within an allowable limit, as required by the run time conditions of a search. The scheme has similarities to multi-level load balancing described in [KGR,94], since the threads become arranged in the form of a tree. However, it differs because the Dynamic Thread Creation is completely dynamic. The threads are not statically ordered. Instead, the structure of the thread tree is formed automatically and will change with the execution of the program. This continually adjusts the computational resources to the part of the search space that requires it most.

In this thesis, we use Dynamic Thread Creation to create two new multithreaded search strategies for constraint programming (Chapter 6). In addition, we applied the new strategies to benchmark problems from the literature. Empirical results of the benchmarks and the impact of Dynamic Thread Creation on the search strategies are reported and analyzed in Sections 6.2.3.2 and 6.3.3.2.

Chapter 5

Multithreaded Parallel Constraint Programming

There are several issues that arise in working with multithreaded parallel constraint programming. Here we detail some of the major issues and provide a basic multithreaded CP model (5.4). The issues discussed include the effect of threads on CP tools (5.1), concurrent execution using management models (5.2), work partitioning (5.3), and methods to create non-backtracking searches (5.5).

5.1 Thread Safety

Multithreaded constraint programming places a restriction on the type of constraint programming tool that can be used for multithreaded applications. In order to write a multithreaded application using a given constraint programming tool, the tool must be thread-safe. A tool is said to be *thread-safe* when the provided functions may be called by more than one thread at a time without requiring any other action on the caller's part. The issue of thread-safety arises from the use of shared resources among threads, the most common being shared memory. Since threads of the same process can access the same memory locations, the use of shared data poses a threat to data integrity. Thus, the implementers of CP tools must assure that data integrity is persevered under concurrent execution. There are two basic solutions for providing thread-safe functions. The first method is not to use global data, instead require all data to be passed as arguments to the provided functions. The other method is to use global data in conjunction with synchronization variables to protect the integrity of the data. A detailed discussion of this can be found in [Muller,97], where 2LP was made thread-safe by using a combination of both methods.

Implementers of CP tools can go further than thread-safety. There is a notion of multithreaded-safety or MT-safety as provided by some of the standard C and C++ libraries. *MT-safety* expands on thread-safety by supporting a "reasonable" level of concurrency. An example of a MT-safe function is Solaris' *malloc*; it allows multiple threads to acquire memory concurrently with very little blocking.

Today many commercial tools such as the ILOG Optimization Suite provide a thread-safe environment, while as of this writing, we are not aware of any CP tool that provides a MT-safe environment. We believe the lack of MT-safe tools is due to the tedious work that is required for its implementation. For instance, to provide MT-safety, developers of a currently existing CP tool can not simply wrap mutual exclusion locks around unsafe code, since that does not permit concurrent access to the code. Instead, they must assure the underlying layers of software upon which their tool was built can support a "reasonable" level of concurrency. Thus, all data structures used internally by the tool must support concurrency such as the data structures described in Section 3.5.

5.2 Management Model

A CP tool that provides services for managing problem data (variables, constraints, and goals) such that different problems can be modeled simultaneously while assuring data integrity (among the different models) is referred to here as providing a *management model*. We refer to the services that provide management and search facilities for a given problem as a *manager*. However, a given CP tool need not contain an actual object called a manager to provide a management model. A manager is a conceptual object that can be comprised of one or more services. The only restrictions of a management model are that a manager can only work on one given problem at a time and different managers can not share the same CP data. A classic example of such a model is ILOG's Optimization Suite (Section 2.3.8).

A management model such as ILOG's, provides an effective means for application-based parallelization. However, a management model raises several issues concerning the achievement of concurrent execution. These issues arise from the type of relationship established between threads and managers. Two possibilities present themselves:

- Different threads can share one common manager.

- Different threads can use different managers.

Using one manager for several threads presents many difficulties. These difficulties stem from the fact that one manager can only work on one given problem at a time. This means that although threads can concurrently share the same manager, if they do, they can not concurrently work on different subproblems. In addition, if threads share the same manager, they can not concurrently work on the same problem: if they did, each thread would overwrite what the other has done (recall threads share the same memory space). One may argue that this may be resolved by the use of synchronization. However, the extra synchronization required to provide data integrity outweighs the performance gain of parallelization. Thus, in order to achieve a significant performance gain from parallelization, concurrently executing threads should use different managers. However, the use of different managers raises issues of its own. These issues stem from the fact that different managers do not share the same CP data. Therefore, in order for managers to work on the same problem, data including variables, constraints, and goals must be duplicated for each manager. In effect, this duplication allows each manager to have a copy of the original problem formulation. In order to achieve a performance gain from parallelization, alterations must be made to each manager's copy of the problem such that each manager contains a different subproblem of the original. This allows different threads to use different managers, each with a different subproblem. Therefore, each thread will have a different task to perform and the application can potentially achieve a performance gain from parallelization.

5.3 Subproblem Formulation

There are two basic methods for partitioning a CP problem: constraint-based partitioning and variable-based partitioning. *Constraint-based partitioning* creates subproblems by partitioning a problem through its constraints. This can be done by partitioning a constraint into two or more disjoint subsets. These new subsets can be distributed among the managers such that each manager has a its own disjoint subset, thus creating two or more subproblems. For example, we are given a two-manager scenario where each manager has a duplicate of the following:

x, *y*, and *z* are decision variables whose domains are {0,...,120}

Posted constraints on *x*, *y* and *z*:

$$x + y = 120 \tag{5.1}$$

$$y - z = 100 \tag{5.2}$$

$$(x = 100 \;\; AND \;\; y \geq 5) \; OR \; (y \leq 100 \;\; AND \;\; x < 25) \tag{5.3}$$

We can divide Constraint 5.3 into two disjoint subsets such that Constraint 5.3 is removed from the original constraints and constraints 5.3_{M1} and 5.3_{M2} are added to their respective managers.

$(x = 100 \;\; AND \;\; y \geq 5)$ (5.3_{M1}) Manager 1's constraint

$(y \leq 100 \;\; AND \;\; x < 25)$ (5.3_{M2}) Manager 2's constraint

The problem is now formulated into two subproblems using two managers. Now, a performance gain is achievable if each subproblem is executed in parallel by a different thread.

Constraint-based partitioning does not always work as easily as shown in the example. Most problem formulations are more complex and usually do not have a constraint with clearly disjoint properties. This makes the partitioning of a constraint difficult. For example, consider a problem with solely arithmetic constraints, such as:

$x + y \geq 50$

$x * y \leq 100$

In this case, constraint-base partitioning is severely hindered since a constraint with clearly disjoint properties does not exist. Variable-based partitioning is better in this regard.

Variable-based partitioning creates subproblems by partitioning a problem through its decision variables. This can be achieved by several methods. One method is to split the domain of a decision variable into disjoint sub-domains. Thereafter, different managers can be given the disjoint sub-domains of the decision variable, thus creating subproblems. In the case that the domains of the decision variables are small (such as in Boolean decision variables), alternate methods could be used for partitioning such as explicitly setting variables. These variables can be removed from the original problem formulation and given to different managers as constants. For example:

> We are given a two-manager scenario with *x*, *y*, and *z* as decision variables whose domains are {0,1} and a set of undisclosed posted constraints. We can remove one of the original decision variables, say *x* and create two

constants, one for each manager. The two constants can be $X_{M1} = 1$ for manager 1 and $X_{M2} = 0$ for manager 2. The managers can then use their respective constants in place of the original decision variable x. This creates two disjoint subproblems using two managers that can be executed in parallel by different threads.

An issue for variable-based partitioning is which variable should be partitioned. If a decision variable is improperly chosen for partitioning, speed-up from parallelization may be insignificant. In most cases, the best choice depends on the application and how it is modeled. Several general strategies suggest themselves, however some are:

- Choose decision variables that are shared by many constraints.
- Choose decision variables that have the largest domains.
- Choose decision variables that have the smallest domains.
- Choose decision variables that have the greatest influence on objective functions.

5.4 Multithreaded Backtracking Models

5.4.1 The Basic Backtracking Model with Load Balancing

We will now discuss a basic approach for application-based multithreaded parallelization. This approach builds on the previous sections and utilizes the built-in search facilities normally provided by a manager. In most cases, this is chronological backtracking. Thus, the basic multithreaded approach is analogous to a parallel backtracking search. However, here we apply the Dynamic Thread Creation scheme for load balancing.

The process begins by creating a single thread. A thread contains its own CP problem formulation, i.e. a manager, decision variables, constraints, and goals, as described in Section 5.2. The task of the thread is to use the built-in search facilities of its manager to solve the problem. However, if at any time during the search, the active thread limit (Section 4.3.3) is not reached and the number of unbound decision variables is large enough to compensate for the overhead of thread creation, a new thread will be created. In this case, the executing thread will choose an unbound variable and partition its domain into two disjoint sub-domains. It will then create the new thread with its own copy of the CP formulation and pass one of the sub-domains and a copy of its state information[2]. The newly created thread will then apply the state information and execute its manager's built-in search facilities. The above scheme is then applied to the newly created thread. This process will terminate when all the threads terminate and a thread will terminate when it has finished its search.

---

[2] A detailed explanation of state copying is given in Section 5.5.2.

5.4.2 Multithreaded Backtracking for Optimization

In the case of a CSOP, refinements can be made to the basic backtracking model to improve performance. Possibilities include implementing a branch and bound scheme, an iterative deepening scheme, or dichotomization. In these cases, we use a global optimum solution stored in shared memory to help eliminate paths from consideration that can not lead to better solutions than the global optimum. As an example, we describe a multithreaded backtracking branch and bound search (MT-BBABS) below.

To fix ideas, assume a minimization problem. The algorithm begins by initializing the global optimum to the upper bound of the problem. At any time while searching, if a thread's *local upper bound*, the upper bound on its subproblem, is greater than the global optimum, it can cause its manager to fail and backtrack. This prunes the search tree by eliminating branches where a better solution is not possible. In addition, if at any time, a thread finds a solution that is better than the global optimum, it can update the global optimum. This allows all threads to have the best bounds for pruning. The complete algorithm is illustrated in Figures 5.1 and 5.2. Figure 5.1 shows the flowchart of "main" and Figure 5.2 shows the flowchart of a thread. For simplicity, we illustrate the algorithm using labeling (Section 2.1.2.4) for subproblem generation.

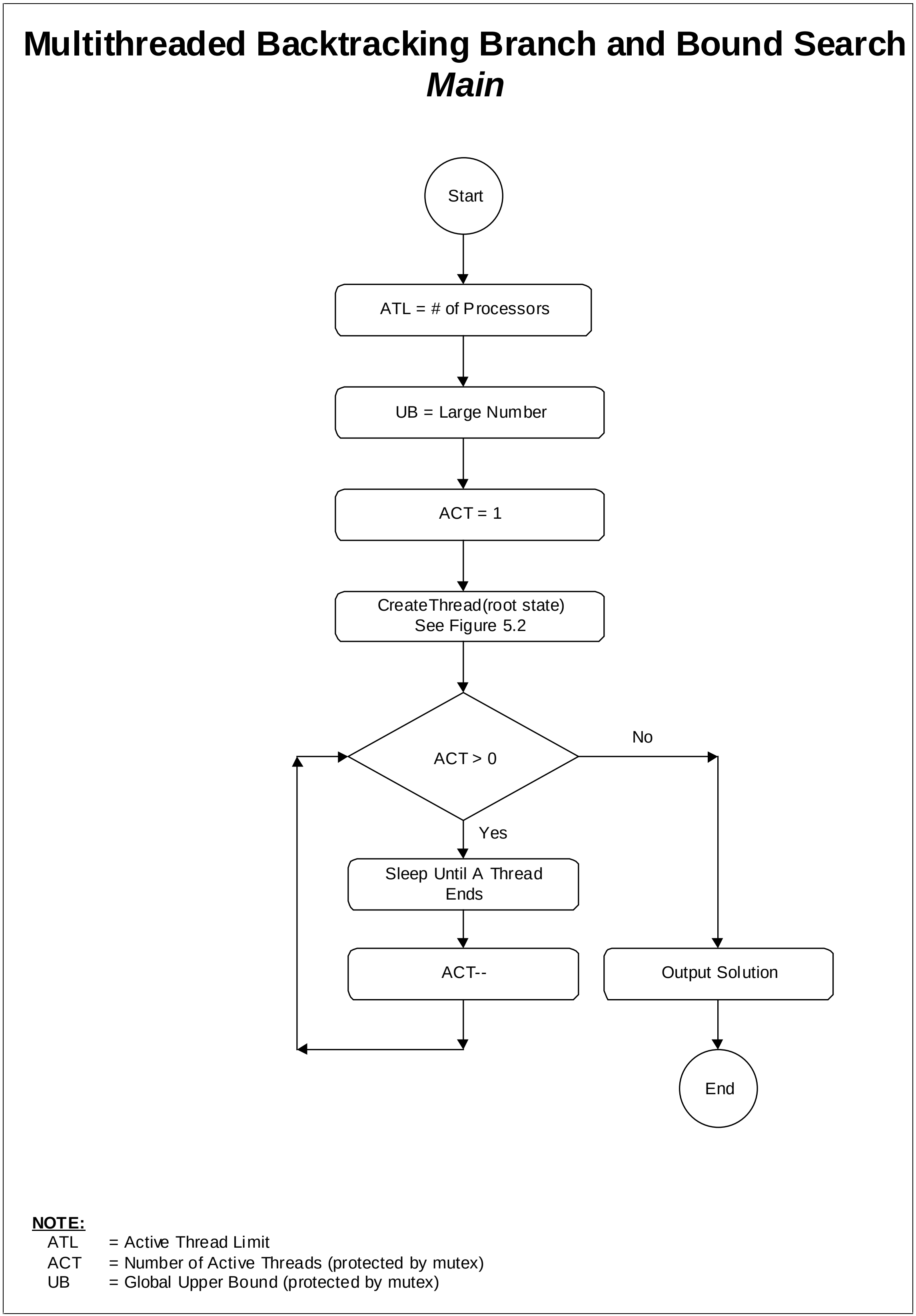

Figure 5.1: Flowchart of "main" for the MT-BBABS.

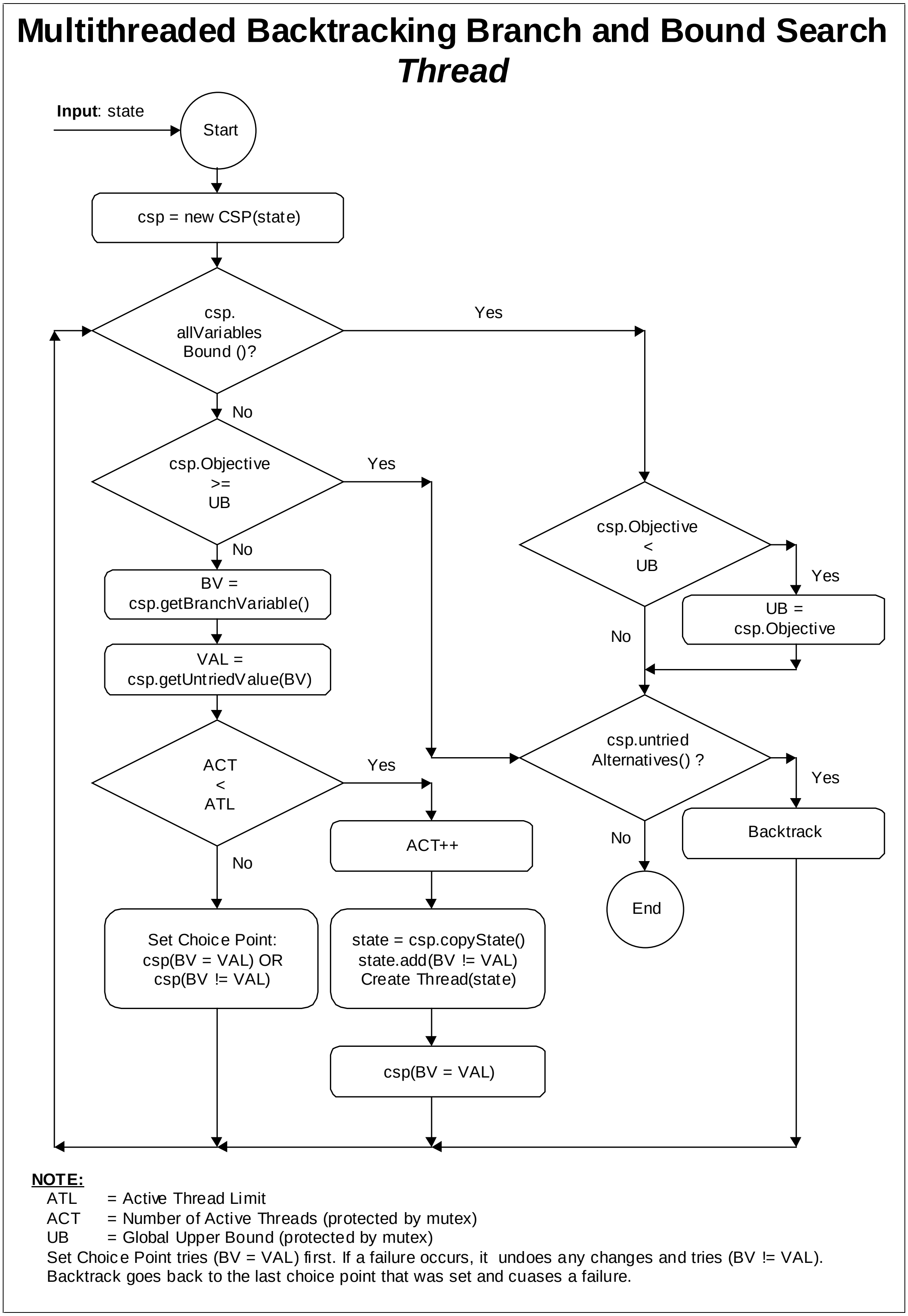

Figure 5.2: Flowchart of a thread for the MT-BBABS.

5.5 Multithreaded Non-Backtracking Models

Although the basic multithreaded backtracking model provides a method to achieve parallelism, it may not be best suited for all problems. That is, different problems may benefit from different techniques. Thus, alternative approaches should be sought. However, the CP research community has primarily focused on searches that are based on backtracking or could be simulated with backtracking. This may be due to fact that the use of non-backtracking approaches requires the programmer to depart from the built-in search facilities of a manager. This places the burden of search management on the programmer. In this case, the CP tool is primarily used as a propagation engine and the programmer must perform tasks such as representing nodes, creating the children of nodes, and traversing the tree of nodes. The next subsections 5.5.1 and 5.5.2 discuss these issues and give the foundation for creating non-backtracking searches in CP.

5.5.1 Node Representation

In the classic backtracking case, a programmer is not particularly concerned with the representation or storage of nodes. That is because in a management model it is the manager's task to represent and store nodes. In that case, new nodes are generated with one manager using chronological backtracking, such that constraints are added to the current node to form new nodes and removed to backtrack to previous nodes. Although efficient, that model severely limits the type of search strategies that can be performed. This is because a manager controls a problem's state. Therefore at one moment in time, one manager allows for the concurrent existence of only one node. Thus, only searches that are based on backtracking are applicable since non-backtracking searches, such as best first search, require nodes to concurrently exist in a queue. We overcome this limitation by using different methods of node representation.

5.5.1.1 External Node Representation

One method for the current existence of nodes is to use an external (to the solving engine) data structure to store state information. This structure would contain only data that was relevant to the state of the given problem. For example, in the simple case of a CSOP with binary variables and an integer objective, a node could be a data structure consisting of two integers to store minimum and maximum values of the objective, and an array of integers to represent the current status of the decision variables. This is shown below using C++ syntax.

```
class node {
public:
        int     minObjective;
        int     maxObjective;
        int     decisionVariableStatus[NUM_OF_VARS];
};
```

Since this representation is external to the solver, several instances of the data structure can be instantiated to create multiple nodes. When the expansion of a node is required, we can use a thread to copy the data of the given node into a "host" CP formulation, execute the formulation's propagation engine and then copy the resulting state information into a new node. This is illustrated in Figure 5.3.

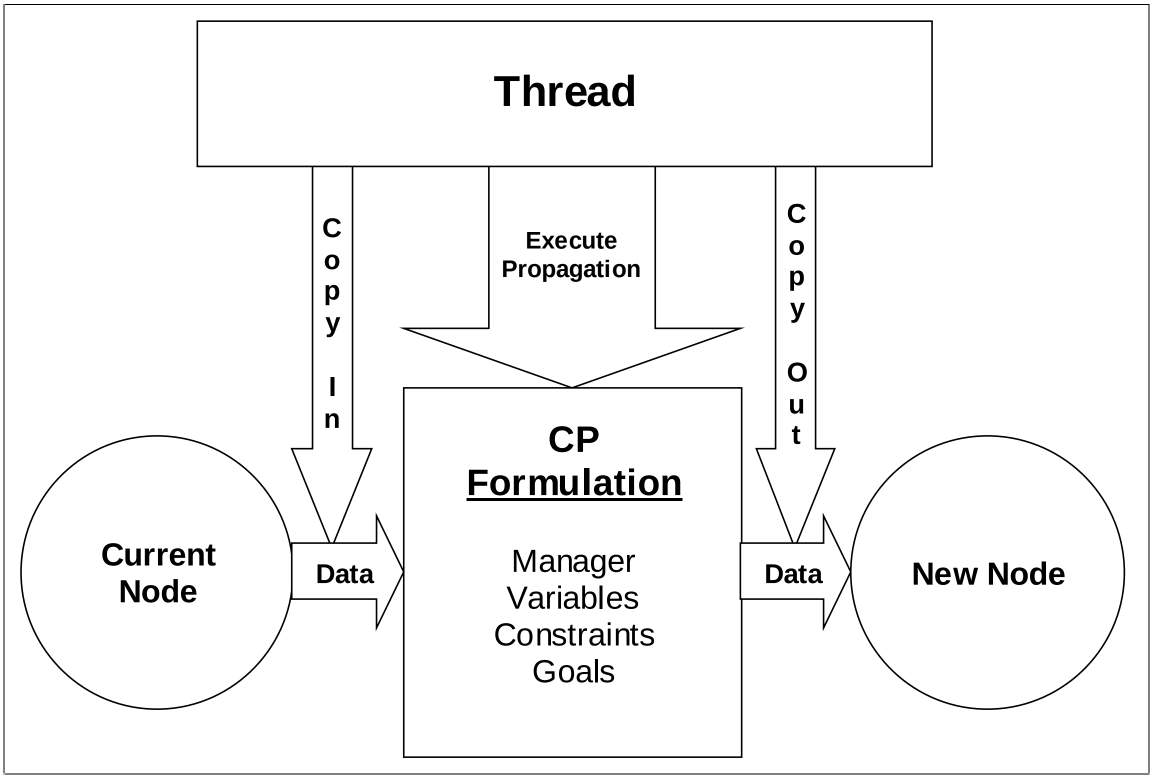

Figure 5.3: Diagram for using external node representation.

Although external node representation is not difficult to implement, the continuous copying of state information to and from a CP formulation can be inefficient. This is particularly true, if the number of variables is large.

5.5.1.2 Explicit Node Representation

An another method for the concurrent existence of nodes is to use explicit node representation. In this case, we exploit memory to increase application performance by maintaining relevant information, a commonly used practice in hardware and software caching. For constraint programming, this can be done by defining a node as a CP formulation. In this case, a node consists of a manager and the given problem's decision variables, constraints, objectives, and goals. Let us reconsider the example in Section 5.5.1.1, a CSOP with binary variables and an integer objective. An explicit node representation for the example is given below using ILOG Solver and C++ pseudo code.

```
class node {
public:
        node();
        IlcManager          manager;
        IlcIntVar           objective;
        IlcBoolVarArray     decisionVariables;
};
node :: node() :  manager(IlcNoEdit) {
        decisionVariables   = IlcBoolVarArray(manager, NUM_OF_VARS);
        objective           = IlcIntVar(manager, MIN_OBJ, MAX_OBJ);
        manager.setObjMin(objective);
        manager.add(CONSTRAINTS);
        manager.add(GOALS);
}
```

As we can see, a node is an explicitly defined CSOP. Thus in this case, a node maintains state information in its natural structure. The benefit here is that useful information does not have to be recomputed as with often as with external representation method. However, explicit representation is less memory efficient than external representation, since in that case only little information is stored, where in this case, the whole problem structure is stored.

5.5.1.3 New Possibilities

Both methods of node representation allow nodes to concurrently exist with other nodes. This concurrent existence allows alternative search strategies such as a best-first search to be used. Furthermore, since nodes are separate entities, threads can share nodes. For instance, one thread can perform a partial search on a node and then pass that node to another thread, where it can continue the search and so on. This creates a greater degree of flexibility allowing many types of parallel strategies to be used.

5.5.2 Efficient State Storage and Retrieval

The use of nodes as separate entities requires the programmer to be responsible for the storage and retrieval of state information. Specifically, in order to create the children of a parent node, the programmer must create subproblems of the parent. This can be accomplished in three steps:

1. Create new CP formulations.
2. Duplicate the parent's state in the new formulations.
3. Use partitioning to create subproblems from the new formulations.

5.5.2.1 Creating New CP Formulations

The first step in creating a node's children is to create copies $CSP_{01},...,CSP_{0n}$ of the initial state CSP_0, where n is the number of children to be generated and CSP_0 is the original CP formulation. In the case that we are using explicit node representation, creating copies of CSP_0, actually entails creating new CP formulations for the given problem, i.e. for each CSP_{0j} a new manager, decision variables, constraints, and goals. In the case that we are using external node representation, "host" CP formulations are required. In this case, we do not need to create new problem formulations. Instead, we must assure that the "host" formulations are in the initial state (clear of any previous state information).

5.5.2.2 Duplicating State

To duplicate a given state CSP_i we need to duplicate the relevant information that is stored in CSP_i. Given the initial state CSP_0 and the state CSP_i, the relevant data of CSP_i are all the changes that have occurred from CSP_0 to CSP_i. These changes represent operations that have been performed, which can include the domain reduction of decision variables and the addition of new constraints. In order to duplicate the state of CSP_i we need to duplicate the changes brought on by the operations. Specifically, if we are given copies $CSP_{01},...,CSP_{0n}$ of an initial state CSP_0 and a subproblem CSP_i whose state we wish to duplicate in $CSP_{01},...,CSP_{0n}$, we must copy to each CSP_{0j} all the changes that have occurred from CSP_0 to CSP_i. This can be done by exploiting the fixed point semantics of constraint propagation (Section 2.1.2.1) by using recomputation methods.

In the case where the changes were the cause of operations that do not have side effects, we are only required to copy the results of the operations. This is the case if domain reductions have occurred. In this case, the changes are represented in the domains of the decision variables. Thus, one possibility is to explicitly copy the domains of each decision variable in parent CSP_i to each CSP_{0j}. Clearly, this will duplicate any domain reductions that may have occurred. In the case where we have binary variables, the process is trivial, since all we need to copy is one value for each variable. However, multi-value decision variables pose more difficulty. In the multi-value case, the domain of a decision variable can contain many values, with several gaps. This makes the copying of domains tedious and inefficient. Thus, we use an implicit method. Instead of explicitly copying all the domains of the decision variables, we exploit the fixed point semantics of constraint propagation by using domain recomputation. This is accomplished by only copying domain information that can be used to efficiently recompute the domains of all the variables.

The actual method used for domain recomputation is dependent on the method used for subproblem generation. Recall from Section 2.1.2.4, that one method for creating a subproblem is labeling, fixing a selected branching variable at one of its possible values, and then propagating the consequences. In this case, recomputation is performed by only copying the domains of the decision variables that have been bound. Recall, these are decision variables whose domains have a cardinality of one. Their values are then explicitly set in each CSP_{0j} . Thereafter, we propagate the effects of those bindings, which in turn will recompute the domains of the unbound decision variables.

Although not previously mentioned, another method for creating subproblems from a parent node is to perform a dichotomizing domain reduction on a selected branching variable. This is often done when the given problem has variables whose domains contain many values or are continuous. In this case, subproblems are generated by dividing the domain of the branching variable in half. This creates two subproblems, one with the upper half of the domain and one with the lower half. In this case, recomputation is performed by copying the *range* of each decision variable, the minimum and maximum value of its domain, to each CSP_{0j} . The effects of the copying are then propagated.

In the case, where the changes from CSP_0 to CSP_i are the cause of operations that have side effects, only copying the results of the operations will not suffice. The fact that the operations have side effects warrants that they must be copied also. This is the case if new constraints are added for or as part of node generation. In this case, we also copy the new constraints to $CSP_{01},...,CSP_{0n}$ and then propagate for each CSP_{0j}. Note, the order in which the constraints are copied is irrelevant. The fixed point semantics of constraint propagation guarantees that the domains of decision variables are always reduced in the same way regardless of the order that the constraints are considered. Therefore, the copying of the new constraints to $CSP_{01},...,CSP_{0n}$ will result in the same fixed point being reached as CSP_i.

5.5.2.3 Partitioning

After state copying, $CSP_{01},...,CSP_{0n}$ represent functional copies of parent CSP_i. To create subproblems of CSP_i we use variable-based partitioning on $CSP_{01},...,CSP_{0n}$. Specifically, we choose a decision variable from CSP_i that is not bound and use variable-based partitioning to distribute the sub-domains of that variable to $CSP_{01},...,CSP_{0n}$. Thereafter, we propagate the consequences, thus creating child nodes (subproblems) of CSP_i. It should be noted that in practice for the purpose of performance, we wait until the partitioning stage before we execute the propagation engine. That is, we do not execute the propagation engine after we copy the state of the parent to its children. Instead, we wait until the problem is partitioned and then execute the propagation engine. Thus, propagation is only performed once.

5.5.2.4 Explicit Node Representation Benefit

If we use the explicit node representation presented in Section 5.5.1.2 we can further improve the performance of subproblem generation. This is because with explicit node representation a parent node can be transformed to one of its children. In theory, after a node is used to generate its children, it can be deleted, since it no longer needs to be considered. However, with explicit node representation, a parent node still contains useful state information in its natural structure. Thus, with an additional constraint the parent can be easily transformed into one its children. This saves computational time required for subproblem generation as in a backtracking search. Furthermore, in an application where the branching factor is low such as zero-one integer problem, only one new node would be required. Thus copying would only be required to create one child.

Chapter 6

Two New Multithreaded Search Strategies

6.1 Introduction

In this chapter, we present two new multithreaded search strategies and their applications. In addition, we test the performance of the new strategies on benchmark problems from the literature and report empirical results. For completeness, we provided a description of the test bed.

The experiments were performed on an Ultra Enterprise 4000, a SUN SMP with fourteen 250MHz SUN UltraSparc processors, each with 4 megabytes of L2 cache. The system contained 1.8 gigabytes of RAM and operated under the Solaris operating system Version 2.6. The applications were programmed in C++ using ILOG's Optimization Suite. The source code was linked with the Solaris thread library and was compiled using Sun's C++ Compiler Version 4.0, with options:

-xtarget=ultra -xarch=v8plus -xcache=16/32/1:4096/64/1 -O5

For a comparison of how this platform performs in relation to other platforms see the Linpack Benchmark Report [Dongarra,92]. An updated version of the report can be found at *http://www.netlib.org/benchmark/performance.ps.*

It should be noted that although the test bed had fourteen processors, we conducted experiments with at most twelve active threads. This is because we did not have access to all the processors of the test bed for an extended period of time. However, we were able to monopolize twelve processors for extended periods. Thus, the experiments were conducted with at most twelve active threads.

6.2 Multithreaded Best-First With Backtracking Search

6.2.1 Search Strategy

There are many problems where global optimization is attractive. The Multithreaded Best-First With Backtracking Search (MT-BFWBS) is a hybrid approach for solving such problems. Specifically, the algorithm uses threads to perform a parallel best-first with backtracking search. The idea behind the best-first feature is to expand the nodes that are most likely to lead to an optimal solution first. However, this requires the use of a queue for node storage and thus makes memory a limitation. The MT-BFWBS accounts for memory by using backtracking to control the size of the queue. Specifically, when the queue grows to a certain size, nodes are no longer expanded in a best-first manner. Instead, they are expanded using backtracking. Thereafter, when the size of the queue is reduced, the MT-BFWBS reverts to a best-first search. The benefit of this design is that many of the best nodes will be expanded first while controlling the memory requirements of the search.

6.2.2 Implementation

The MT-BFWBS uses a centralized concurrent priority queue implemented with a heap [HMPS,96] to store nodes. In general, the order in which the nodes are arranged on the queue is determined by whether the objective is a minimization or maximization function. Again to fix ideas, we assume a minimization problem. In this case, the nodes are arranged in ascending order according to a given heuristic evaluation function.

The use of a queue to store nodes requires the concurrent existence of nodes. Section 5.5 presented two methods for this. However, through experimentation we have discovered that the best performance of the MT-BFWBS is achieved by using explicit node representation. We attribute the success of explicit node representation to two key factors of the MT-BFWBS. First, global optimization requires a great deal of computational power and efficiency, since theoretically the whole search tree must be accounted for. Thus, the execution time for the creation and copying of nodes must be minimized. Clearly, explicit node representation is better in this regard. Secondly, the MT-BFWBS incorporates backtracking to control the size of the priority queue. Therefore, nodes that use larger amounts of memory are not problematic. Hence, the trade-off of using more memory to produce quicker results was ideal for the MT-BFWBS.

For simplicity, we begin by discussing the search as a single threaded application. The algorithm begins by placing the root node in the priority queue and then creating a thread. A thread's task is to remove the best node from the queue to expand it. The method of node expansion and the control of the search are determined by the size of the queue. If the size of the queue is within a given limit, the thread expands the node in a breath-first fashion, creating the children of the node. If these newly created nodes are feasible and do not violate any bounds, they are inserted into the priority queue. Of course, this increases the size of the queue. However, if the size of the queue exceeds the given limit, the thread will not expand a node by normal means. Instead, the thread will use chronological backtracking to fully expand the node. Once fully expanded and bounds updated (if required), the node no longer needs consideration and can be deleted, reducing the size of the queue. This keeps the queue to a controllable size while assuring that many of the best nodes will be expanded first. The process terminates after the thread terminates. The thread terminates when the queue is empty or when the best node on the queue has a lower bound greater than or equal to the current best solution.

To perform the algorithm in parallel more threads are required. The Dynamic Thread Creation scheme presented in Section 4.3 can be easily adapted for this purpose. The parallel version begins just as the sequential version, with only one thread. However, during the expansion of a node, if the number of concurrently executing threads is less than the active thread limit (Section 4.3.3), the thread creates a new thread and assigns it the expansion of half the children, using variable-based partitioning. This is true for both methods of node expansion. The threads then continue in the same manner, removing the best nodes from the queue and expanding according to the size of the queue. In addition, all threads are functionally equivalent, and thus can create other threads if the number of concurrently executing threads is less than the active thread limit. A thread terminates when the queue is empty or when the best node on the queue has a lower bound greater than or equal to the current best solution. The process terminates when all threads terminate. The complete algorithm is shown in Figures 6.1-

6.4. Figure 6.1 shows the flowchart of “main”, Figure 6.2 shows the flowchart of a thread, Figure 6.3 shows the flowchart of the breath-first node expansion and Figure 6.4 shows the flowchart of the depth-first node expansion. For simplicity, we illustrate the algorithm using labeling (Section 2.1.2.4) for subproblem generation.

6.2.3 Applications

The MT-BFWBS is applicable to optimization problems that have a good heuristic for measuring bounds, since these bounds will be used to order the nodes in the priority queue. The accuracy of the heuristic will directly affect the performance of the overall algorithm. Clearly, a more accurate heuristic will provide better performance.

Another consideration in using the MT-BFWBS is the branching strategy to be used. Branching strategies that create a smaller number of child nodes (e.g. labeling) have shown greater performance than branching strategies that create a larger number of child nodes. Experiments have shown that branching strategies that create many child nodes cause the priority queue to grow full at a faster rate. This fills the priority queue with nodes that are relatively high in the search tree. Consequently, the priority queue may become full with many less promising nodes. Once the priority queue is filled, the search will switch to backtracking to reduce the size of the priority queue. However, after the queue size is reduced, it will again quickly grow full, since the branching strategy creates many child nodes. As a result, the overall search will spend more time backtracking than performing best-first node expansions. This makes the overall search perform more like a parallel backtracking search. However, in this case, the performance is not comparable to a parallel backtracking search, since best-first node expansion adds additional overhead from creating and copying nodes. Therefore, branching strategies that create fewer child nodes should be used.

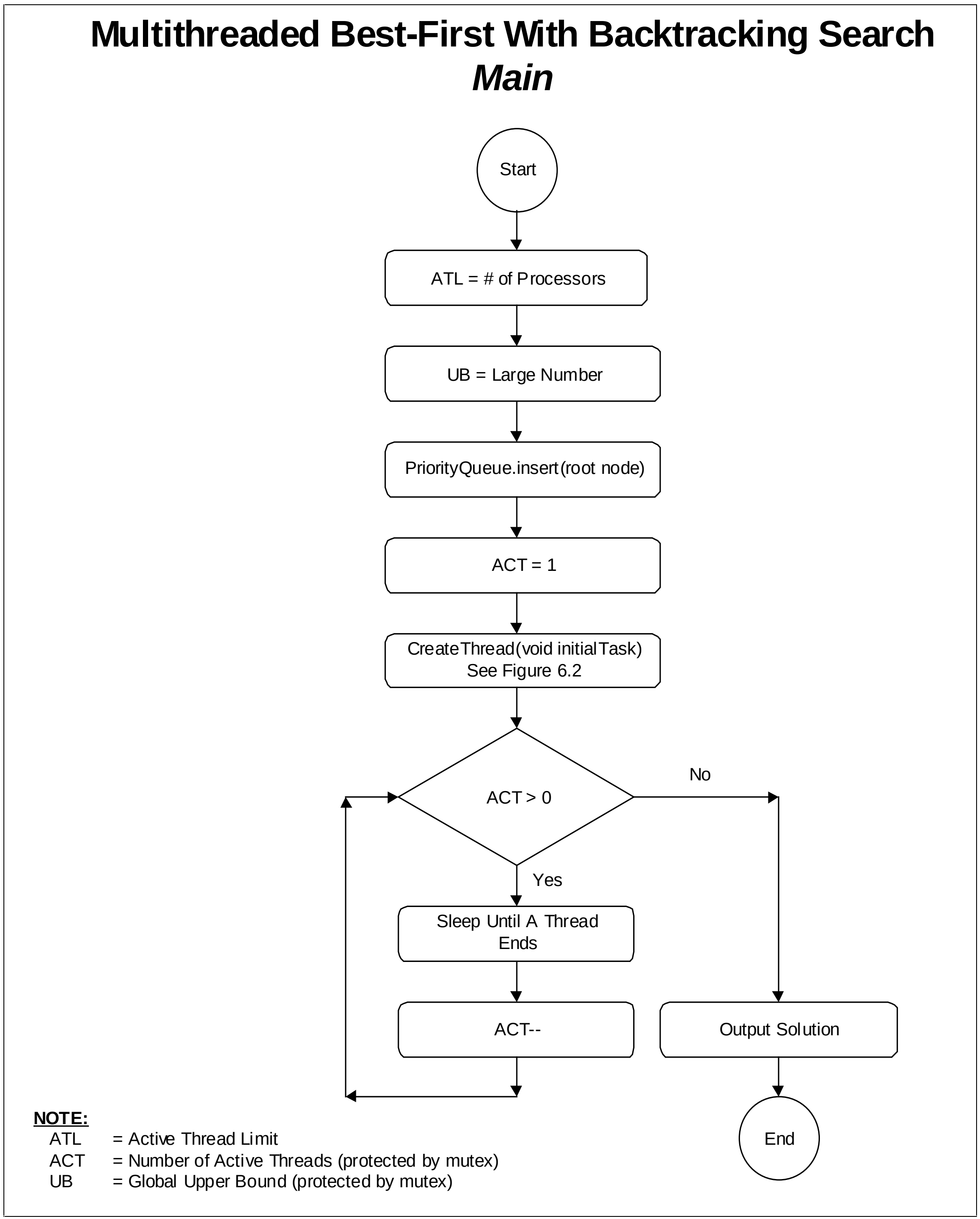

Figure 6.1: Flowchart of "main" for the MT-BFWBS.

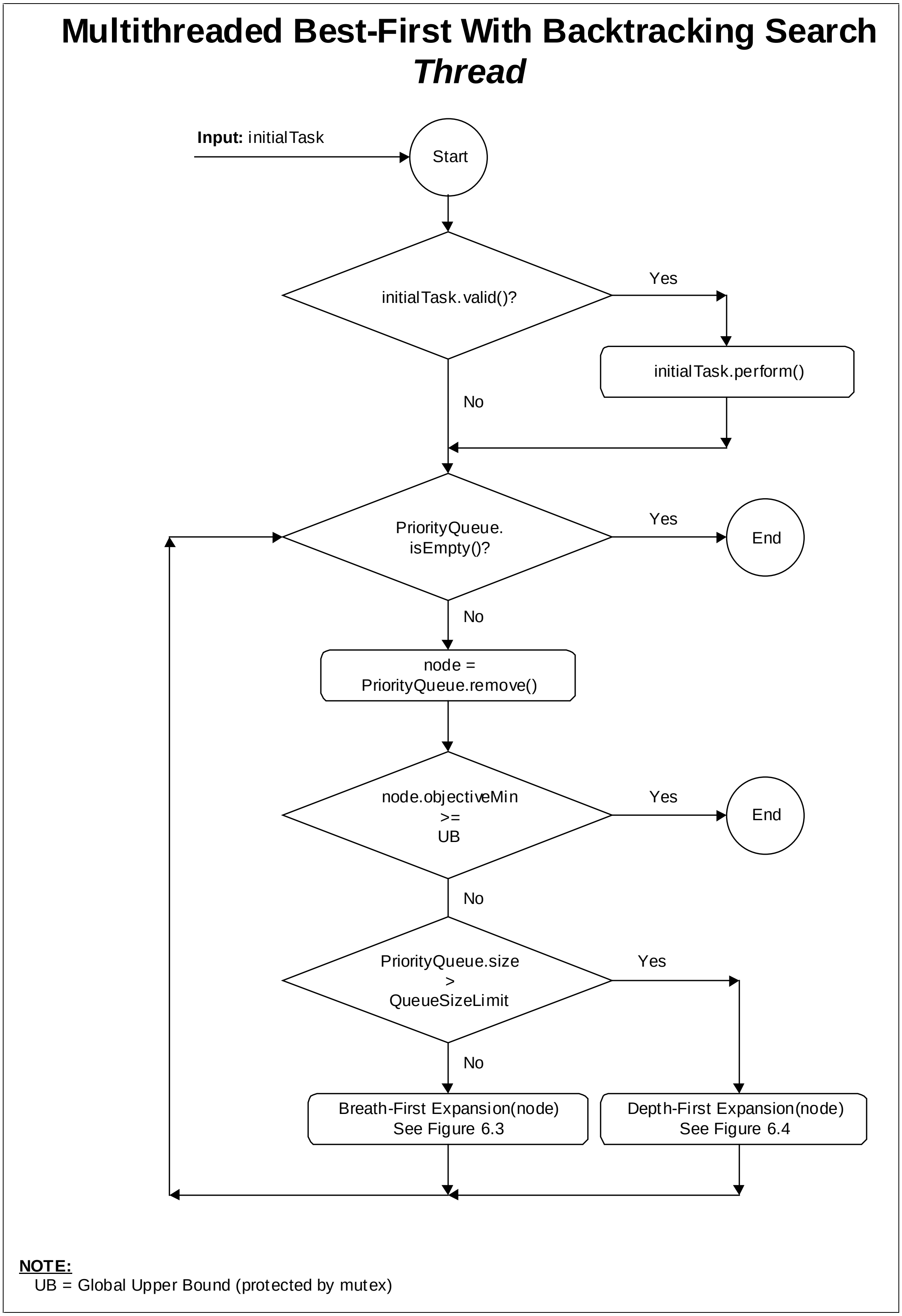

Figure 6.2: Flowchart of a thread for the MT-BFWBS.

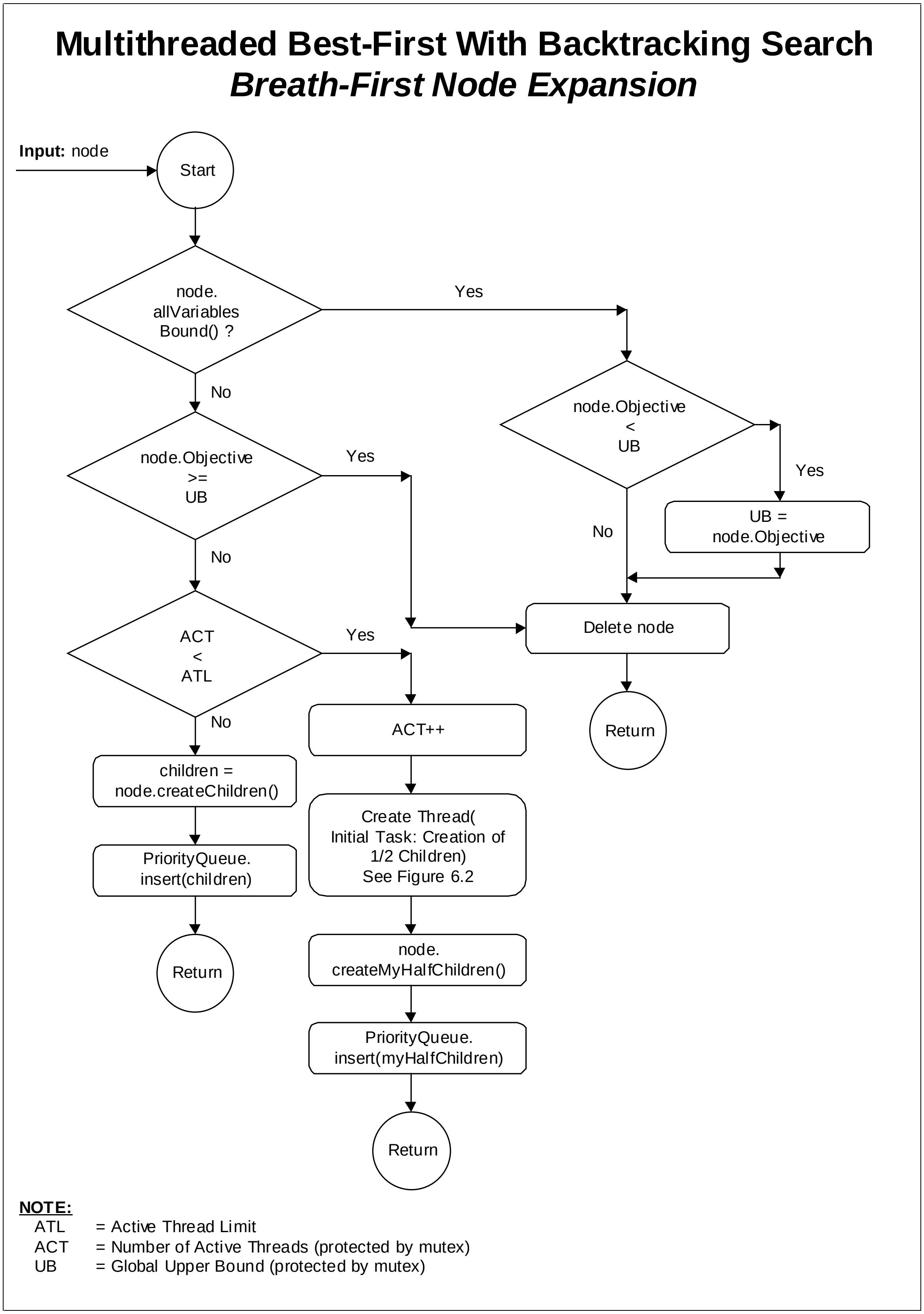

Figure 6.3: Flowchart of the breath-first expansion for the MT-BFWBS.

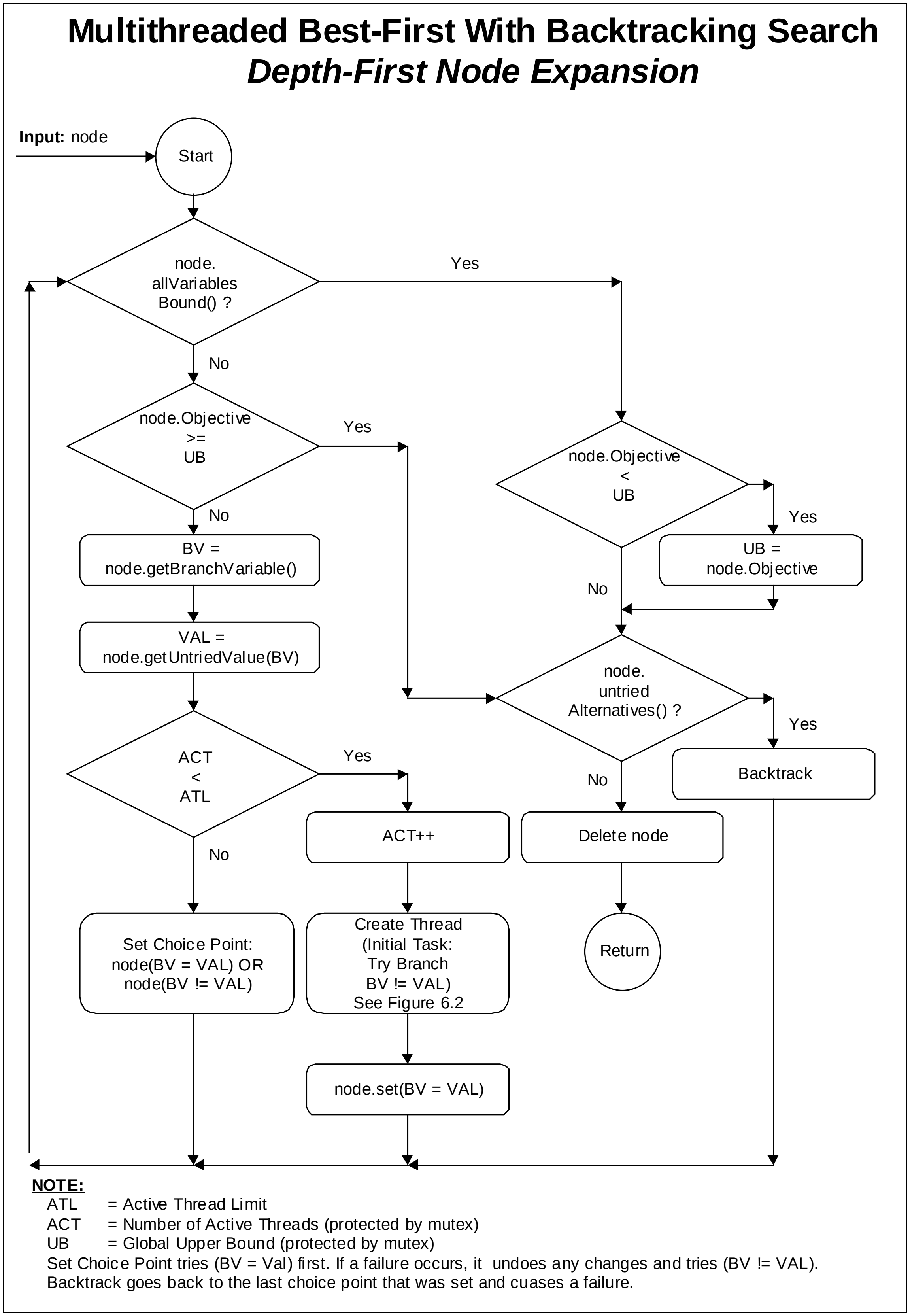

Figure 6.4: Flowchart of the depth-first expansion for the MT-BFWBS.

6.2.3.1 The Set Covering Problem

The Set Covering Problem (SCP) is a well-known and well-studied problem upon which the MT-BFWBS can be applied. It is the problem of covering the rows of a *m*-row, *n*-column, zero-one matrix (a_{ij}) by a subset of the columns at a minimal cost. The *density* of an instance of the SCP is defined as the percentage of ones in the matrix a_{ij}. The problem can be formally defined as follows:

Let

x_j = 1 if column *j* is in the solution

x_j = 0 otherwise

c_j = the cost of column *j*

then the problem is to

minimize $$\sum_{j=1}^{n} c_j x_j \tag{6.1}$$

subject to $$\sum_{j=1}^{n} a_{ij} x_j \geq 1, \quad i = 1,...,m, \tag{6.2}$$

$$x_j \in \{0,1\}, \qquad j = 1,...,n. \tag{6.3}$$

Constraint 6.2 insures that each row is at least covered by one column and Constraint 6.3 is the integrality constraint. If the inequality in 6.2 is changed to an equality, the resulting problem is called the *Set Partitioning Problem*. If in the SCP the costs of the columns are equal then the SCP becomes a special case known as the *unicost SCP*. Here we will consider the general case where column costs may differ.

The SCP has many real applications that include crew scheduling [Rubin,73], facilities location [TSRB,71], and Boolean expression simplification [Breuer,70]. Although easy to formulate, solving an instance of the SCP is not trivial. The size of its search space is 2^n. In fact, the SCP is NP-hard [GJ,79].

Various methods have been proposed for solving the SCP, including exact methods and heuristics. Most of the early algorithms proposed were exact methods based on various branch and bound techniques, and improvements on such methods by obtaining tighter bounds [Etcheberry,77] [BH,80] [Beasley,87]. As of late, the focus has shifted from exact methods to heuristic algorithms using Lagrangean-based approaches such as in [Beasley1,90] [CNS,96].

6.2.3.2 Empirical Results

The SCP was modeled as an integer programming problem. It was implemented with ILOG Solver and Planner such that two identical integer formulations were maintained for each. However, each had its specific duty. ILOG Solver was used to manage the local search strategies, while Planner managed all the linear constraints. In addition, communication of variable bounds was automatically carried out by the engines. This allowed the each to help the other solve its problem. This is similar to the cooperating solvers work in [DB,95].

The application used a small amount of preprocessing such as column reductions by using column domination rules as defined in [Beasley,87]. The goal here was not to get the maximum number of reductions. Instead, it was to obtain a more manageable problem quickly. In addition, during preprocessing, a global upper bound was obtained by using a progressive round-off algorithm. For more information on progressive round-off see [MT,96].

The application was implemented with the MT-BFWBS. The concurrent priority queue was ordered by the lower bounds of the nodes, which were simply their linear relaxations given by Planner. Two distinct branching strategies were used for the different expansion methods. The backtracking expansion method used one criterion for selecting a branching variable, the commonly used "most fractional variable". The breath-first expansion used two criteria. First, the best 10% of the most fractional variables are chosen and from that 10% the variable with the highest cost was chosen. If there was a tie based on cost, the "most fractional variable" from the two was chosen. Other ties were arbitrarily broken. The intuition behind this branching strategy is that hardest decisions are choosing which of the higher costing columns should be equal to one, since those columns will more dramatically increase the objective. Thus, we should try to make the hardest decisions first.

The application was tested on a range of SCP instances obtained from the OR-Library at Imperial College [Beasley1,90]. Table 6.1 shows the details for the data sets tested. Each data set has 5 data instances. Table 6.2 shows the details for the data instances, including the number of columns after column reductions, the upper bound obtained by progressive round-off, and the optimal solution. The number of columns after column reductions, gives the size of each data instance's search space. From Table 6.2, we can see that this ranges from 2^{632} to 2^{937}. It is clear, even with reductions, that the tested data instances produce very large search spaces.

Data Set	Rows	Columns	Density
D	400	4000	5%
E	500	5000	10%
F	500	5000	20%

Table 6.1: SCP data set details.

Data Instance	Reduced Columns	Upper Bound	Optimal Solution
D.1	632	63	60
D.2	672	69	66
D.3	686	74	72
D.4	643	67	62
D.5	606	61	61
E.1	830	32	29
E.2	949	33	30
E.3	871	28	27
E.4	930	30	28
E.5	957	31	28
F.1	729	15	14
F.2	671	16	15
F.3	752	15	14
F.4	697	15	14
F.5	655	14	13

Table 6.2: SCP data instance details.

Table 6.3 shows the average execution times (in seconds) and speedups for the data instances. Each data instance was tested with the active thread limit (ATL) set at 1, 2, 4, 8, and 12. Each ATL instance was executed three times and averaged. Recall, the ATL determines the concurrency limit. Thus, by varying the ATL we are varying the concurrency of the MT-BFWBS. In addition, a time limit of 16 hours was imposed for each trial. The rationale for limiting the execution time to 16 hours was that most computationally demanding applications are run overnight: from 5 P.M. to 9 A.M.. If an optimal solution was not found within the 16-hour time limit, no solution (NS) is reported. In these cases, speedups are not reported, since they are not applicable (NA). In addition, when an optimal solution was not found with ATL = 1, but was found with ATL > 1, the speedup calculation was prorated.

Data Instance	Active Thread Limit									
	1		2		4		8		12	
	Time	Speedup	Time	Speedup	Time	SpeedUp	Time	Speedup	Time	Speedup
D.1	182.3	1.0	93.3	2.0	44.0	4.1	18.7	9.8	17.3	10.5
D.2	438.3	1.0	259.7	1.7	122.0	3.6	60.0	7.3	47.3	9.3
D.3	705.3	1.0	379.0	1.9	200.3	3.5	105.7	6.7	79.0	8.9
D.4	978.7	1.0	544.0	1.8	207.3	4.7	115.3	8.5	70.3	13.9
D.5	9.0	1.0	5.0	1.8	3.0	3.0	3.0	3.0	3.0	3.0
E.1	48918.7	1.0	24739.0	2.0	12717.7	3.8	6633.7	7.4	4457.3	11.0
E.2	NS	NA	NS	NA	NS	NA	28362.0	1.0	20683.7	1.4
E.3	NS	NA	31869.7	1.0	15950.7	2.0	6064.0	5.3	4630.0	6.9
E.4	47278.0	1.0	17862.7	2.6	9386.3	5.0	4617.7	10.2	3075.3	15.4
E.5	16409.7	1.0	8446.7	1.9	3738.0	4.4	1173.0	14.0	797.0	20.6
F.1	30417.0	1.0	15268.3	2.0	6848.7	4.4	2845.3	10.7	2014.3	15.1
F.2	8450.0	1.0	4393.0	1.9	2663.7	3.2	1684.7	5.0	1411.3	6.0
F.3	13849.7	1.0	7281.7	1.9	3780.0	3.7	2207.3	6.3	1798.0	7.7
F.4	28899.3	1.0	15152.3	1.9	7093.0	4.1	4154.7	7.0	3257.0	8.9
F.5	NS	NA	18942.3	1.0	7051.0	2.7	3664.7	5.2	2944.7	6.4

Table 6.3: Average execution times (seconds) and speedups for the SCP using the MT-BFWBS.

The results in Table 6.3 show that in the case of data instances where optimality was not proven in 16 hours (e.g. data instance E.3, where ATL = 1) increasing the ATL proved optimality. Thus, we can conclude in the case of the MT-BFWBS, increasing the number of active threads can increase the size of solvable problems. In addition, the results in Table 6.3 show that increasing the ATL leads to significant decreases in execution time. Although, it should be noted that the actual performance gained from increasing the ATL varied from data instance to data instance. However, it should also be noted, that on average, the speedup achieved using the MT-BFWBS is very close to being linear to the ATL, as shown in Figure 6.5.

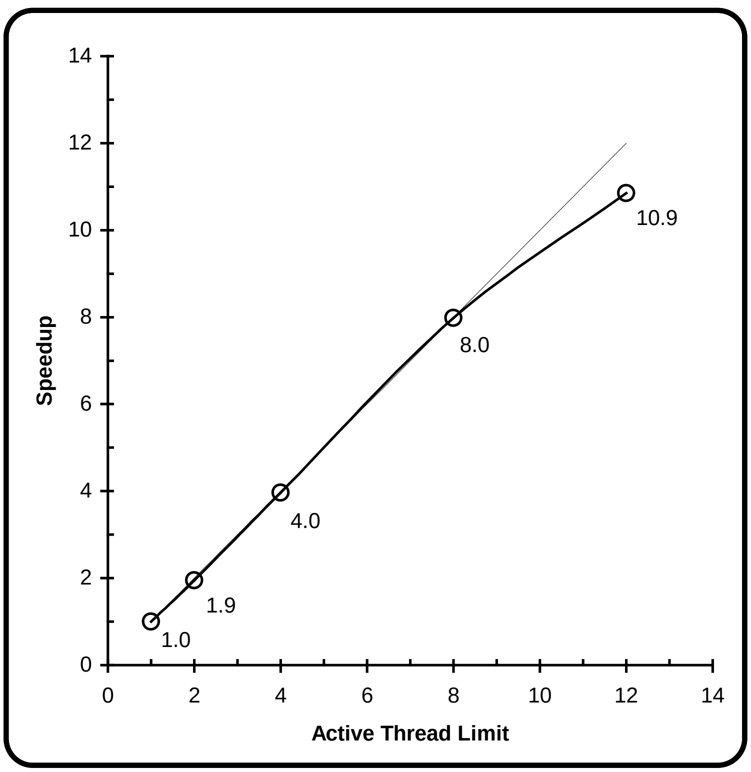

Figure 6.5: Graph of the average speedup for the SCP using the MT-BFWBS.

To gain better insight into the role of Dynamic Thread Creation in the MT-BFWBS, we investigated the number of threads that were created and terminated for each trial. Our first observation was that the number of threads created and terminated grew with the execution time of each data instance. Secondly, in most cases, the number of threads created and terminated exceeded the active thread limit. In some cases, the number of threads created and terminated was greater than 300. Thus, it was clear that in the case of the MT-BFWBS the low cost of thread creation was being fully exploited. Although it should be noted that in the case of data instance D.5 the maximum number of threads created (8) was less than the ATL, when ATL = 12. This was because that data instance was solved before the problem could be completely partitioned among 12 threads. However, it should also be noted that that data instance took only three seconds to solve.

In addition to testing multithreaded performance, we compared the execution times and the number of nodes visited by the MT-BFWBS with ATL = 1 to two sequential algorithms, ILOG Planner MIP and ILOG Planner & Solver depth-first branch and bound (DFBB). Planner MIP is the CPLEX mixed integer algorithm provided in ILOG Planner. In actuality, Planner MIP is CPLEX MIP, one of the best mixed integer engines available, since the underlying engine for Planner is CPLEX. ILOG Planner & Solver DFBB uses the same cooperating solvers implementation as the MT-BFWBS, except it uses a sequential depth-first branch and bound search as in [DB,95]. Table 6.4 shows the comparison of the different algorithms on data set D. It should be noted that in all cases, the same starting bounds and columns reductions were used. Thus, all algorithms had the same starting points and when possible used the same branching rules.

Data Instance	Planner MIP		Planner & Solver DFBB		MT-BFWBS (ATL = 1)	
	Time	Nodes	Time	Nodes	Time	Nodes
D.1	41.0	767	137.0	826	182.3	731
D.2	289.3	5828	1331.3	8850	438.3	1585
D.3	928.0	16760	2704.7	15352	705.3	2875
D.4	457.0	8298	908.0	5918	978.7	4619
D.5	6.7	78	7.3	28	9.0	15

Table 6.4: Comparison of the average execution times (seconds) and node counts for the SCP.

Table 6.4 shows that Planner MIP had faster execution times than the other algorithms on the tested data instances. On average, it was 2.8 times faster than Planner & Solver DFBB and 2 times faster than the MT-BFWBS. However, the MT-BFWBS visited 3 times fewer nodes than the Planner & Solver DFBB and 3.5 times fewer nodes than Planner MIP. From the node count results, we can conclude that the disparity in performance of the MT-BFWBS as compared to the Planner CPLEX MIP algorithm is due to the highly optimized implementation of the latter, since the MT-BFWBS visited fewer nodes.

In addition to comparing, execution times and the number of nodes visited, we also compared the solution quality on data set D, after a 3-minute time limit. In this case, we also tested the MT-BFWBS with ATL = 12. The results are shown in Table 6.5 where solutions in bold represent that the solution found was proven optimal within the time limit.

Data Instance	Planner MIP	Planner & Solver DFBB	MT-BFWBS (ATL = 1)	MT-BFWBS (ATL = 12)
D.1	**60**	**60**	60	**60**
D.2	68	68	66	**66**
D.3	75	73	72	72
D.4	66	64	63	62
D.5	**61**	**61**	**61**	**61**

Table 6.5: Comparison of the SCP solution qualities after a 3-minute time limit.

Table 6.5 shows that the MT-BFWBS provided better quality solutions than the other algorithms, using the same amount of time. In addition, it shows that increasing the number of active threads in the MT-BFWBS with a fixed time limit improves the quality of solutions.

6.3 Multithreaded Limited Discrepancy Search

6.3.1 Search Strategy

There are many problems that have search spaces so large, that they effectively prohibit the use of exhaustive search techniques, even with the additional power of parallelization. In these cases, a heuristic is often used to guide the search through a small fraction of the search tree. However, a heuristic can fail. A Limited Discrepancy Search (LDS) [HG,1996] deals with this issue by assuming that the heuristic may have had succeeded if not for a few wrong turns, most likely made at the top of the search tree. Thus, we may want to deviate from the heuristic from time to time. These deviations are called *discrepancies.* The LDS works by setting a discrepancy limit *d* that restricts the search to paths that do not diverge more than *d* times from the choices recommended by the heuristic. The LDS is most often implemented with chronological backtracking, where *d* increases on each iteration and all paths that have a discrepancy less than *d* are repeated. The algorithm starts with *d* = 0, i.e. exactly following the heuristic. In this case, only the left most branch of the search tree is explored. If a solution is not found *d* is incremented to 1 and the search is tried again, this time allowing at most one deviation from the heuristic for every path. Thus, all paths with more than *d* decrepancies are pruned. If this fails, *d* is incremented again and the process is iterated, until either a solution is found or it is proven that there is no solution. This is much like an Iterative Broadening Search (IBS) [GH,91], where imposed breath limits are increased until a solution is found and an Iterative Deepening Search (IDS) [Korf,85], where imposed depth limits are increased until a solution is found.

The LDS has been used successfully for solving scheduling problems, such as Job Shop [HG,95], Resource Constrained Project Scheduling [Crawford,96], and Preemptive Job Shop [LB,98]. In all these cases, the LDS was implemented sequentially, using chronological backtracking. Here we will present the Multithreaded LDS (MT-LDS), a multithreaded version of the LDS that uses a queue and thus, does not require backtracking. In addition, this version of the LDS is geared to CSOPs that have many easily found solutions such that the density of solutions makes optimality difficult to prove. In these cases, the MT-LDS can be used to provide good solutions in a reasonable time by not guaranteeing optimality.

6.3.2 Implementation

The MT-LDS uses a two lock concurrent queue [MSL,96] to store nodes. The use of a queue requires the programmer to represent the nodes. In Section 6.2, we saw a case where the explicit method of node representation provided more benefits than the external method. This is not the case here. In this case, external node representation was found more beneficial: the benefits of memory efficiency outweighed the benefits of greater node copying performance. This is because the version of the LDS presented here does not use chronological backtracking. As previously noted, nodes are stored in a queue. This places a critical importance on the number of nodes the queue can hold, since there is a direct correlation between the number of nodes the queue can hold and the number of discrepancies that can be tried. Therefore, we must minimize the memory requirements of a node. One can argue that memory limitations may force the number of discrepancies to be low or may cause the search to be incomplete. However, another argument can also be made that the LDS assumes a good heuristic. Clearly, if more than a few discrepancies are required the heuristic used may be inappropriate. In these cases, a new heuristic should be sought or an alternative search should be used, since even the standard LDS is of little help with a poor heuristic.

We begin describing the search as a single threaded application, assuming we have a minimization CSOP and we are using labeling (Section 2.1.2.4) for subproblem generation. The algorithm begins by initializing the upper bound of the problem. Unlike other versions of the LDS, the MT-LDS will use the upper bound to prune paths that can not produce better solutions, just as in a branch and bound search. The algorithm then creates the root node and places it in the queue. Since external node representation is used, a node is simply a data structure and thus requires an instance of the solving engine to actually perform the search. This is done in a thread by allowing it to contain the original problem formulation consisting of a manager, decision variables, and constraints.

The algorithm then sets the discrepancy limit as determined by the user. This limit controls the number of variations from the heuristic that must be tried. Consequently, it also controls the size of the search space to be explored, quantified as *C(n,d)*, the number of combinations of size *d* that can be derived from *n* variables, where *d* is the discrepancy limit, as shown in Equation 6.4.

$$C(n,d)=\frac{n!}{(n-d)!\,d!} \tag{6.4}$$

After setting the discrepancy limit, the algorithm creates a single thread. The task of a thread is to remove the node at the head of the queue and perform an LDS-Probe on it. Before performing the LDS-Probe, the state of the solver is stored and the state information of the node is copied into the thread's problem formulation.

The LDS-Probe begins by selecting a branching variable from the set of unbound variables. Thereafter, a heuristic is used to select a value for the branching variable. This creates a choice point. Either the variable is equal to the value or it is not equal to the value. The path where the variable is equal to the value is the one the thread will follow, since the LDS assumes the heuristic is correct. The other path is a discrepancy. Before following the heuristic, if the number of current discrepancies is less than the discrepancy limit, the thread will use forward checking to assure that the node created by the discrepancy is feasible, since we do not want to store nodes that are not feasible. If the discrepancy is deemed feasible, the thread creates a new node, and copies its current state information and discrepancy information, along with the new discrepancy to be tried later. Thus, the new node will have one more discrepancy than the current node. This new node is then inserted at the end of the queue. The thread will then bind the branching variable to the value chosen by the heuristic and then select another branching variable, repeating the process until all variables are bound or a failure occurs. If a failure occurs, the thread will restore the state of the solving engine to undo any of the variable bindings. If a solution is found it will be compared against the upper bound, and if found better, the upper bound will be updated. Thereafter, just as with a failure, the thread will restore the state of the solving engine to undo any of the variable bindings. The thread will then repeat the whole process, by removing the node from the head of the queue and performing an LDS-Probe on it. This will continue until the queue is empty.

To perform the algorithm in parallel we use the Dynamic Thread Creation scheme, as in Section 6.2. The parallel version begins just as the sequential version, with only one thread. However, in the parallel version the upper bound of the problem must be stored in shared memory to provide access to all threads. In addition, during an LDS-Probe after the insertion of the discrepancy node, if the number of concurrently executing threads is less than the active thread limit (Section 4.3.3), the initial thread creates a new thread. The task of the new thread is the same as the initial thread, to remove nodes from the head of the queue to perform LDS-Probes until a solution is reached. In addition, all threads are functionally equivalent, and thus can create other threads if the number of concurrently executing threads is less than the active thread limit. A thread terminates when the queue is empty. The process terminates when all threads terminate. The algorithm is illustrated in Figures 6.6-6.9. Figure 6.6 shows the flowchart of "main", Figure 6.7 shows the flowchart of a thread, Figure 6.8 shows the flowchart of the LDS-Probe function, and Figure 6.9 shows the function for creating nodes with discrepancies.

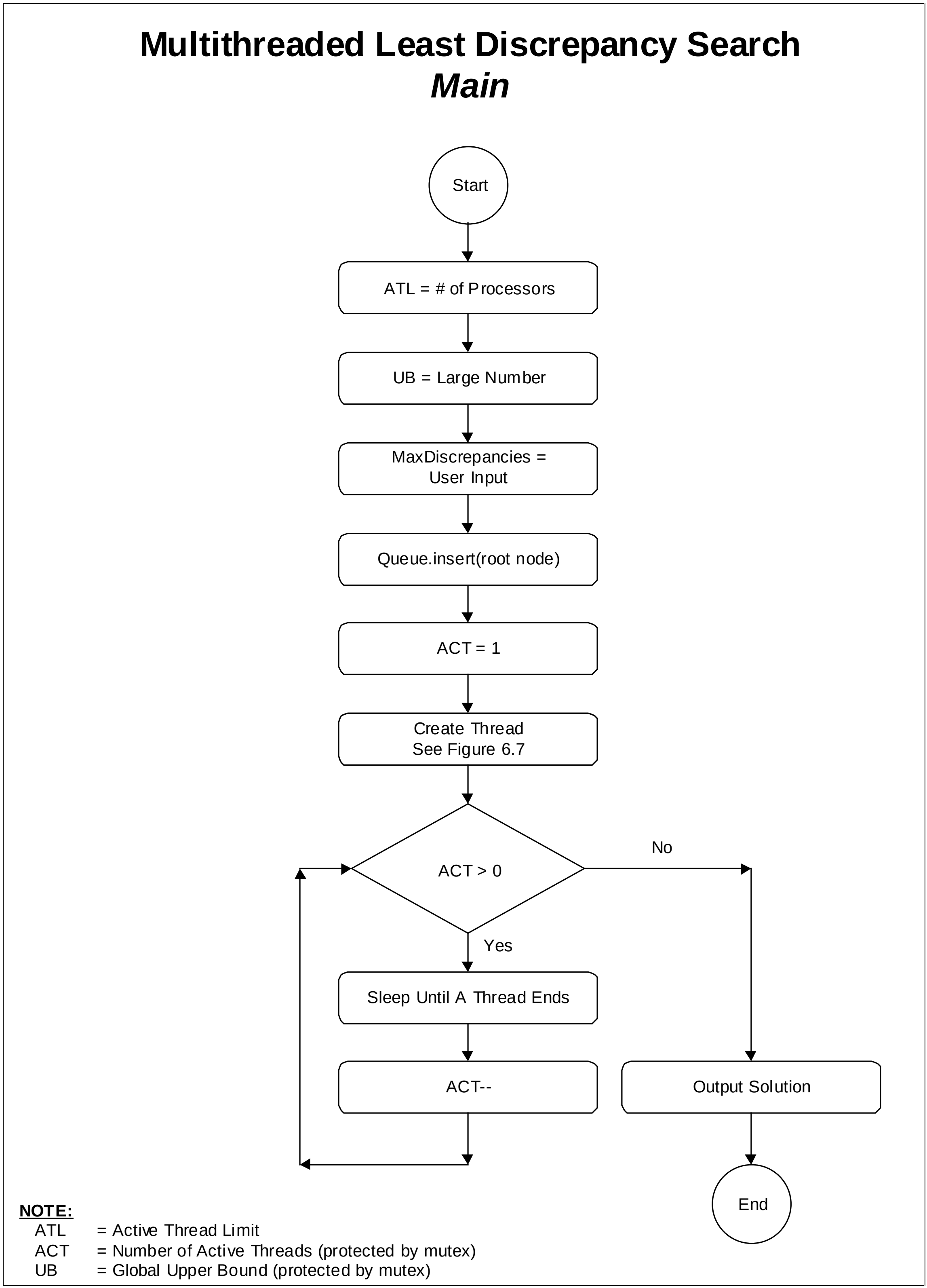

Figure 6.6: Flowchart of "main" for the MT-LDS.

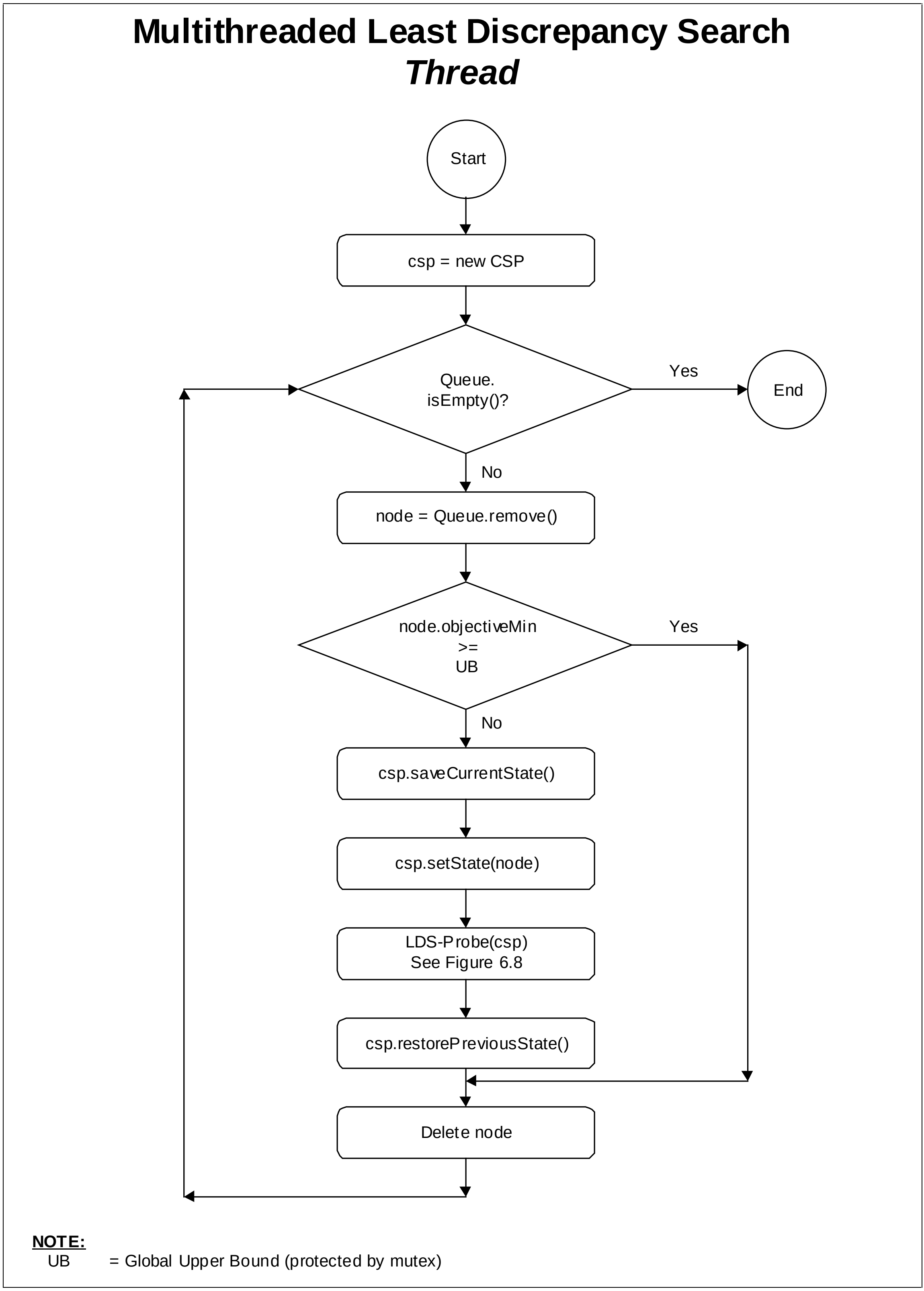

Figure 6.7: Flowchart of a thread for the MT-LDS.

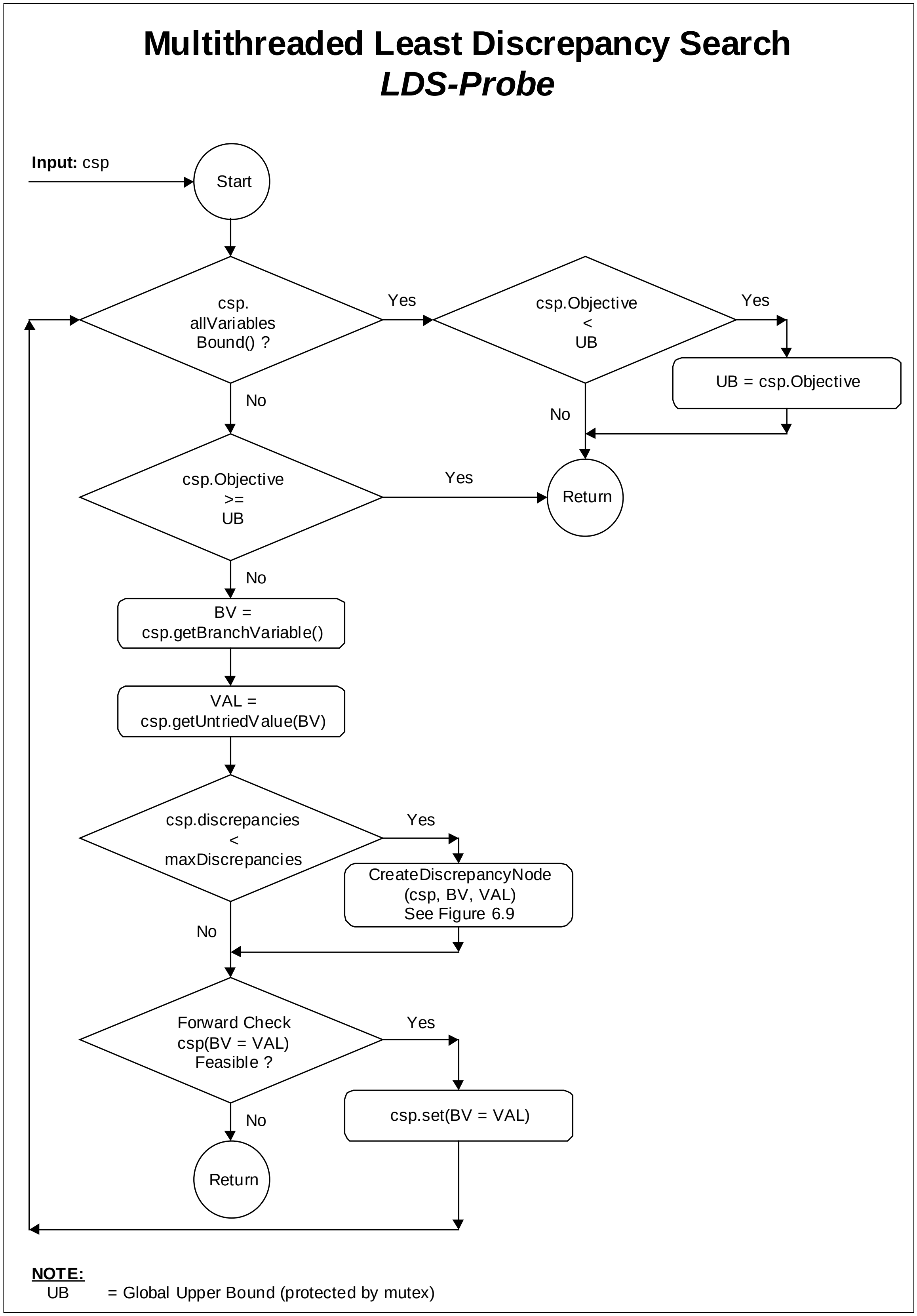

Figure 6.8: Flowchart of the LDS-Probe function for the MT-LDS.

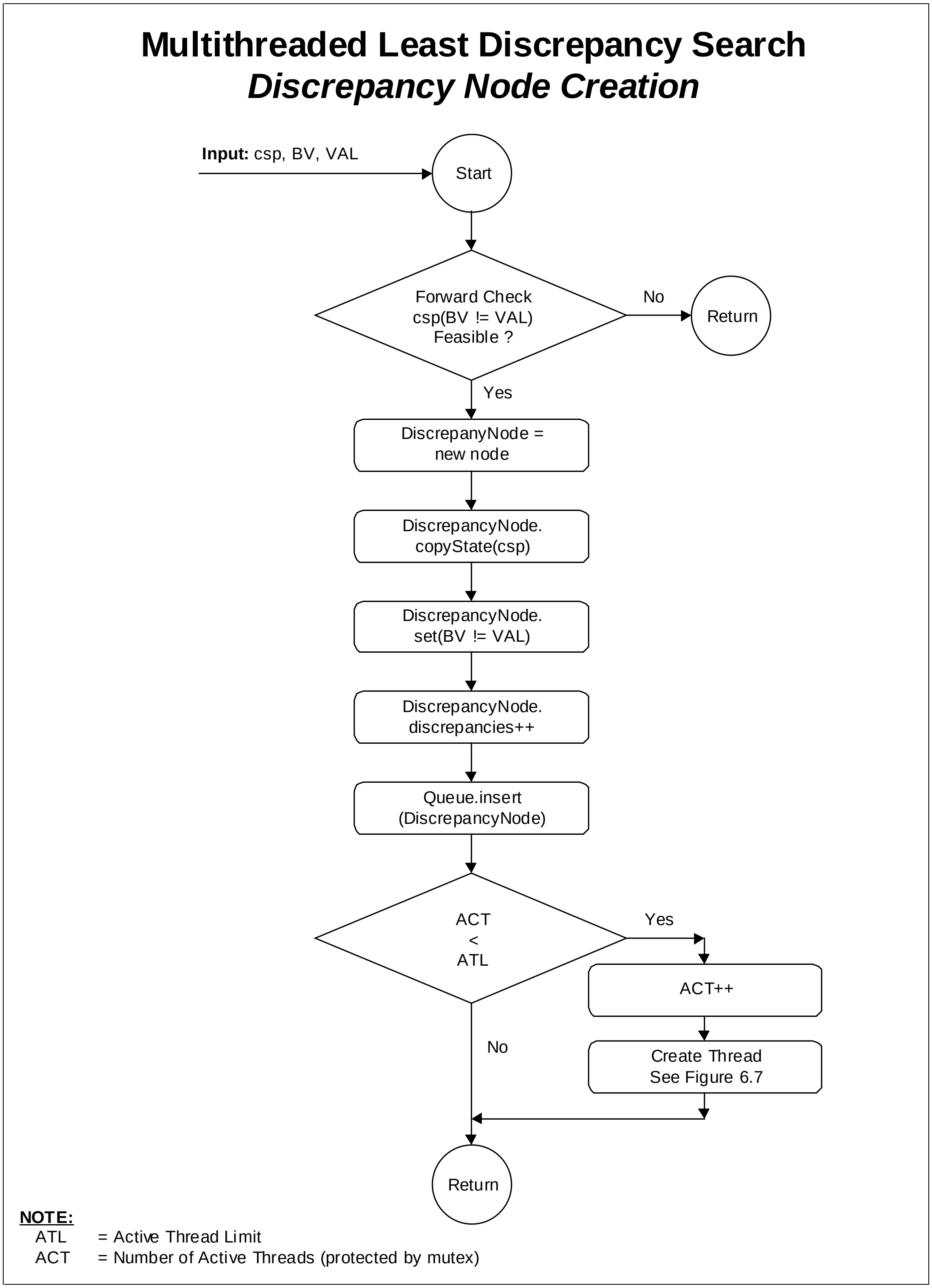

Figure 6.9: Flowchart of the creation of nodes with discrepancies for the MT-LDS.

6.3.3 Applications

The MT-LDS is applicable to large CSOPs, where search space size prohibits the use of exhaustive search techniques. In these cases, guaranteeing optimality is often impractical. Thus, the MT-LDS can be used to provide relatively good solutions in a limited time. However, the quality of the heuristic will directly affect the quality of solutions. Thus, a candidate problem should have a good heuristic. One such problem is Resource Constrained Project Scheduling.

6.3.3.1 Resource Constrained Project Scheduling

A Resource Constrained Project Scheduling (RCPS) problem consists of a finite set of tasks and resources with the following properties and constraints:

- A task has a fixed duration.
- A task may require one or more resources at a time.
- A resource has a finite capacity that can not be exceeded.
- Some resources are interchangeable, i.e. parallel resources.
- Each task may have a deadline, such that it must end before a given time.
- A partial ordering of the tasks exists such that one task may be required to precede another.

The goal of a RCPS problem is to find a schedule that satisfies the problem constraints and minimizes the *makespan*, the time it takes to complete all tasks. Finding an optimal makespan is quite difficult, since theoretically it requires accounting for the complete search space. If we consider the problem of scheduling *n* tasks as a permutation of the tasks, the complete search space can be as great as the factorial of *n*. However, heuristics can be used to guide the search through only the portions of the search tree that may lead to good solutions. This makes the RCPS a candidate for the LDS.

6.3.3.2 Empirical Results

The Multithreaded LDS was applied to a series of benchmark RCPS problems made available at *http://www.neosoft.com/~benchmrx* by Barry Fox of Boeing and Mark Ringer of Honeywell. The web site also provides a single data set that is applicable to all the problems and is based on large-scale assembly. The data set contains 575 tasks and 17 resources. There are two types of resources: zone and labor. There are 13 zone resources and 4 labor resources. Both zone and labor resources have finite capacities that can not be exceeded. However, the labor capacities can vary from shift to shift. The physical problem, which the abstract data set represents, is the process of creating a large assembly consisting of a large number of discrete tasks. Each task entails the performance of specific work documented in formal process plans. The general goal is to minimize completion time of all the tasks. However, each problem instance has various constraints that either simply or further constrain the general problem. Here we will consider problems 2-4, which can be described as follows:

Problem 2: If we had unlimited labor resources, how long would it take to schedule all the tasks? In this case, we can ignore any labor constraints, but must respect the precedence and zone constraints. This problem is the simplest case. However, the lack of labor resources and the looseness of constraints create the largest search space of all the problems.

Problem 3: If we had limited labor resources and the additional constraint that each task must continue from start to finish without stopping, how long would it take to schedule all the tasks? In this case, we must respect labor, precedence, and zone constraints while assuring a task finishing within the same shift as it started. This is the most difficult case. However, the tightness of constraints yields a smaller search space than Problem 2 or Problem 4.

Problem 4: If we had limited labor resources and each task could stop between shifts, how long would it take to schedule all the tasks? In this case, we must respect labor, precedence and zone constraints while allow tasks the freedom to finish in the next shift of the one it started. This is a relaxation of Problem 3, but like Problem 2, the looseness of constraints provides a greater search space than Problem 3.

The three problems were modeled with ILOG Solver and Scheduler, using the provided objects. Each problem was modeled as a permutation of tasks and thus, used a heuristic to select the next activity to be scheduled. It was found that picking the unscheduled task with the smallest maximum start time and smallest minimum start time was best. Once an activity was selected, following the heuristic meant scheduling the activity to start at its earliest available start time and a discrepancy of the heuristic was to "postpone" the activity, stating it could not start until another activity was started before it. This pushed the starting time of the postponed activity upward and made each decision binary.

We applied the MT-LDS to the three problems varying the number of discrepancies and the active thread limit (ATL). Tables 6.6-6.8 show the solutions achieved for the different problems varying the number of discrepancies. The format of the solutions is DD/SS + HH:MM where DD is the day number, SS is the shift number and HH:MM gives the hours and minutes into that shift. Tables 6.9-6.11 show the average execution times (in seconds) and speedups for the different problems varying the number of discrepancies and threads. Each discrepancy instance greater than 0 was tested with the ATL set at 1, 2, 4, 8, and 12. Each ATL instance was executed three times and averaged. In addition, as with the SCP application a time limit of 16 hours was set for each trial. In this case, if a search tree provided by the discrepancy limit was not fully explored within the time limit, no solution (NS) was reported. In these cases, speedups are not reported, since they are not applicable (NA). In addition, when a search tree provided by the discrepancy limit could not be fully explored with ATL = 1, but was with ATL > 1, the speedup calculation was prorated. Figures 6.11-6.13 show graphs for the average speedup of all the discrepancy instances that completed within the 16 hour time limit.

Discrepancies	Solution
0	39/1 + 2:19
1	39/1 + 0:07
2	38/2 + 6:37
3	38/2 + 6:34

Table 6.6: RCPS Problem 2 solutions using the MT-LDS with various discrepancies.

Discrepancies	Solution
0	46/1 + 6:25
1	44/2 + 3:55
2	44/2 + 3:55
3	44/2 + 2:55

Table 6.7: RCPS Problem 3 solutions using the MT-LDS with various discrepancies.

Discrepancies	Solution
0	45/2 + 2:51
1	43/2 + 0:10
2	43/1 + 5:18
3	43/1 + 1:06

Table 6.8: RCPS Problem 4 solutions using the MT-LDS with various discrepancies.

Discrepancies	Active Thread Limit									
	1		2		4		8		12	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1	104.7	1.0	50.7	2.1	25.3	4.1	13.7	7.7	9.3	11.2
2	3780.0	1.0	1891.0	2.0	970.0	3.9	495.7	7.6	332.3	11.4
3	NS	NA	43133.7	1.0	22151.3	1.9	11265.0	3.8	7570.0	5.7

Table 6.9: Average execution times (seconds) and speedups for RCPS Problem 2 using the MT-LDS.

Discrepancies	Active Thread Limit									
	1		2		4		8		12	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1	100.7	1.0	47.7	2.1	25.0	4.0	13.0	7.7	9.7	10.4
2	2366.0	1.0	1186.0	2.0	608.7	3.9	311.0	7.6	210.0	11.3
3	31449.3	1.0	15771.3	2.0	8236.3	3.8	4255.7	7.4	2694.7	11.7

Table 6.10: Average execution times (seconds) and speedups for RCPS Problem 3 using the MT-LDS.

Discrepancies	Active Thread Limit									
	1		2		4		8		12	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1	124.3	1.0	60.3	2.1	30.3	4.1	16.0	7.8	11.7	10.7
2	3166.3	1.0	1582.3	2.0	813.0	3.9	416.7	7.6	283.0	11.2
3	55388.0	1.0	27793.7	2.0	14256.3	3.9	7245.0	7.6	4737.7	11.7

Table 6.11: Average execution times (seconds) and speedups for RCPS Problem 4 using the MT-LDS.

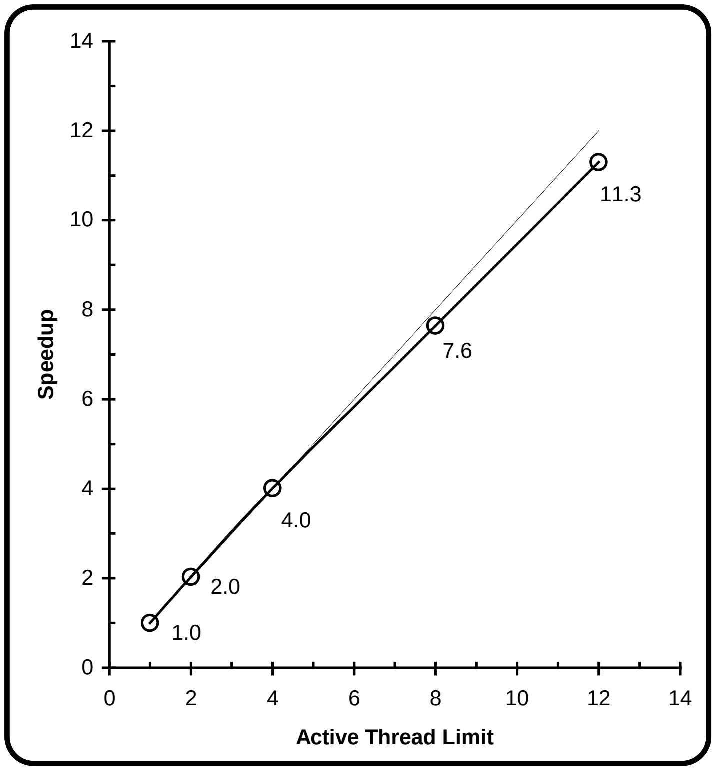

Figure 6.10: Graph of the average speedup for RCPS Problem 2 using the MT-LDS.

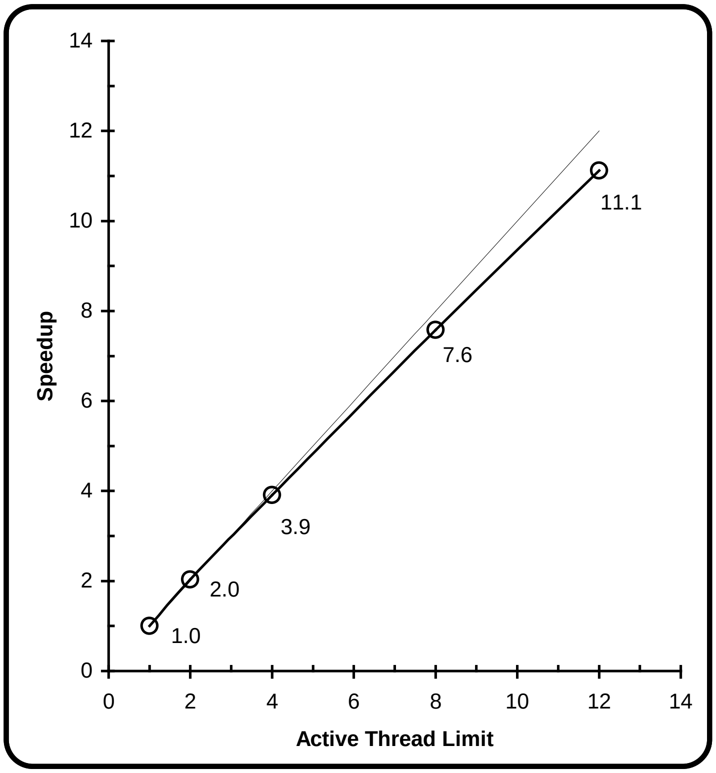

Figure 6.11: Graph of the average speedup for RCPS Problem 3 using the MT-LDS.

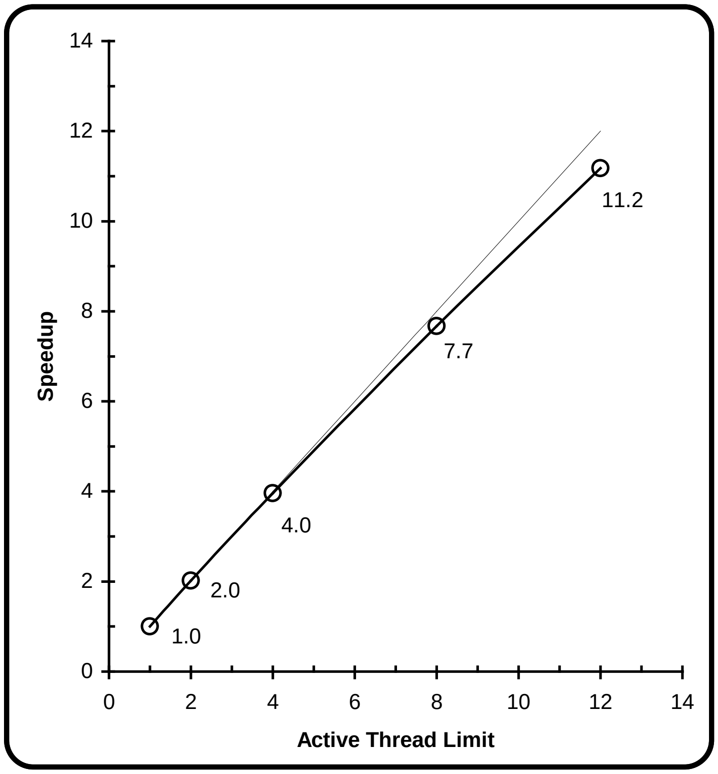

Figure 6.12: Graph of the average speedup for RCPS Problem 4 using the MT-LDS.

The results in Tables 6.6-6.8 show that increasing the number of discrepancies can generally improve the quality of solutions. However, Tables 6.9-6.11 show that increasing the discrepancy limit of a problem also increases its execution time. Recall, the search space of a problem with n variables and a discrepancy limit of d is $C(n,d) = \frac{n!}{(n-d)!d!}$. In addition, Figures 6.10-6.12 show that increasing the number of active threads decreases the execution time of the MT-LDS.

To gain better insight into the role of Dynamic Thread Creation in the MT-LDS we investigated the number of threads that were created and terminated for each trial. We observed that the number of threads created and terminated was always equal to the active thread limit. Therefore, after a thread terminated new threads were not being created. This was very different from the MT-BFWBS. However, it was not surprising since in the MT-LDS the number of nodes to be explored is somewhat fixed. This removes many non-deterministic properties that would otherwise cause a thread to terminate or create new threads. To further understand this point, let us take a closer look at thread termination in the MT-LDS. Let us begin by recalling that a thread terminates when the queue is empty. When this occurs, there are two possibilities:

- There are no active threads and thus the search is complete.
- There are active threads and thus the search is still in progress.

In the case where the search is still in progress, the active threads are most likely expanding nodes whose discrepancy counts are at the discrepancy limit. This is because the nodes with smallest discrepancies are explored first. Consequently, since the nodes being expanded are at the discrepancy limit, the active threads will not create any new nodes with greater discrepancies, i.e. no new work will be created. In fact, the only tasks of the active threads will be to expand their respective nodes by exactly following the given heuristic. Furthermore, after expanding their nodes, the active threads will terminate since the queue is empty. Thus, once one thread terminates, it is highly probable that the search is near completion and thus the other active threads will terminate shortly. This point is supported in our experiments, where there was less than a 1 second difference in time from when the first thread of a trial terminated to when the last thread of the trial terminated. Thus, it becomes clear that the MT-LDS can not fully exploit the low cost of thread creation, as does the MT-BFWBS. Nonetheless, Dynamic

Thread Creation plays an important role in the MT-LDS. It provides a method to dynamically partition the LDS among several threads such that full parallel capacity can be accomplished in logarithmic time (see Section 4.3.4).

In addition to testing for scalability and solution quality per discrepancy limit, we also tested for solution quality with ATL = 1 and ATL = 12, after a 3-minute time limit. In this experiment, the problems were executed without a fixed discrepancy limit. The discrepancy limit started at 0 and incrementally grew until the time limit was reached. It should be noted that in all cases the discrepancy limit never grew greater than 2. We then compared these solutions to solutions from using chronological backtracking with the same heuristic with a 3-minute and a 16-hour time limit. This is shown in Table 6.12.

Problem Number	Chronological Bactracking 3-Minute Time Limit	Chronological Bactracking 16-Hour Time Limit	MT-LDS (ATL = 1) 3-Minute Time Limit	MT-LDS (ATL = 12) 3-Minute Time Limit
2	39/1 + 1:56	39/1 + 1:56	39/1 + 0:07	38/2 + 6:37
3	45/2 + 3:55	45/2 + 3:55	44/2 + 3:55	44/2 + 3:55
4	45/2 + 1:38	44/1 + 1:05	43/2 + 0:10	43/1 + 5:18

Table 6.12: Comparison of the RCPS solution qualities after fixed time limits.

The results in Table 6.12 shows that the MT-LDS provides better quality solutions than chronological backtracking using the same heuristic. In fact, after 16 hours chronological backtracking could not prove optimality and moreover, had a solution that was worse than MT-LDS with a 3-minute time limit. In addition, when comparing the MT-LDS with ATL = 1 to the MT-LDS with ATL = 12, the results show that increasing the number of active threads generally leads to better quality solutions in the same amount of time.

Chapter 7

Conclusions

7.1 Contributions

Constraint programming is an effective method for solving combinatorial problems. However, as problems grow in scale and complexity, they become computationally demanding. This thesis proposed application-based multithreaded parallelization for improving the performance of such problems. It has contributed to the constraint programming field by presenting methods for achieving application-based parallelism. This has included methods for partitioning a CP application to allow parallelism, methods for efficient state duplication, and methods to create non-backtracking searches in a constraint programming environment. Other contributions include the presentation of new parallel algorithms such as Dynamic Thread Creation, an asynchronous load balancing scheme that is specific to multithreaded environments. Dynamic Thread Creation reconfigures traditional load balancing approaches by exploiting the low cost of thread creation. In Dynamic Thread Creation, all threads are functionally equivalent and can create other threads freely as long as the number of active threads does not exceed the available resources. In addition, unlike traditional approaches, when a thread is finished with its task, it simply terminates. This freedom to create and terminate threads as needed minimizes the management of computational resources and provides an intuitive programming model.

In addition to Dynamic Thread Creation, this thesis also presented two new multithreaded search strategies, the MT-BFWBS and the MT-LDS. Both search strategies use Dynamic Thread Creation as the underlying scheme for parallelization and both break away from the classic backtracking schemes that are normally associated with constraint programming. These non-traditional searches are clear examples of the power and flexibility that application-based parallelism provides.

To review, the MT-BFWBS is a hybrid search that combines the benefits of a best-first approach with chronological backtracking. The best-first nature of the search allows the many of the most promising nodes to be expanded first while its backtracking nature controls its memory requirements. The MT-BFWBS was successfully applied to the SCP, where it was found that multithreading improved performance, and in the case where time was limited, it improved solution quality.

The MT-LDS expanded the previous work of [HG,96] [Crawford,96] by presenting a multithreaded version of the LDS that does not use backtracking. In the case of the MT-LDS, a queue is used to store nodes and thus, unlike previous work, paths are never repeated and priority is given to nodes with discrepancies that are higher in the search tree. The MT-LDS was successfully applied to a set of benchmark RCPS problems, where it was found that multithreading improved performance and solution quality.

7.2 Extensions and Future Work

We see two possible directions for future work. The first possibility is to stay completely within the confines of this research. In this case, future work can include applying the provided algorithms to other problems, designing and implementing new multithreaded search strategies, and improving the efficiency of the provided algorithms. In particular, we see an opportunity to improve efficiency by working with the developers of CP tools to provide general methods for the copying of state information. At this time, we are unaware of any CP tool that provides a general method for state copying. Without internal support from CP tools, state copying is computationally expensive and inefficient. An efficient general means for state copying may not only further the development of multithreaded constraint programming, but it may also further the general constraint programming field by allowing programmers to create new search strategies using efficient means for node generation.

The other direction we see for future work is the creation of an open parallel framework for application-based multithreaded parallelism. Through the research for this thesis, we have found that many software components are reusable from one multithreaded application to the next. These reusable components are the parallelizing algorithms such as Dynamic Thread Creation. In addition, we found that the differences from one multithreaded application to the next are based on the CSP model and the type of search strategy that is performed. We believe that high-level instructions can capture these differences. Thus, we seek to create a framework, using the algorithms provided in this thesis, that will allow the application programmer to use high-level instructions to describe the type of parallel search strategy that is to be used. The system will then parallelize the application with threads.

The idea of using high-level instructions for the specification of algorithms is not new. For example, [Frühwirth,95] presented a method to allow user-defined constraints through rules. However, in our case, we are not defining how a constraint is handled, instead we are defining the rules of a parallel search.

Currently, there are two systems, BOB [LRP,95] and Portable Parallel Branch-and-Bound Library [TP,96] that provide an open parallel framework such as the type we are describing. However, both these systems are for the parallelization of general branch and bound algorithms. They are not specific to constraint programming or threads. Thus, using these systems for multithreaded parallelization of constraint programming is not trivial. We believe a framework that is specific to constraint programming will simplify the parallel model.

Let us consider a primitive model for a system that provides parallel search rules for CP. In particular, let us only consider the case of a system for CSPs that contain only binary decision variables. In addition, for further simplicity, the model is first described sequentially.

Let us begin by stating that all nodes are stored in concurrent list L and the model only contains four user-defined rules:

- ***WR:*** Which node should be selected next for expansion from list *L*.

- ***WI:*** Where should a given node *n* should be inserted in list *L*.

- ***WB:*** Which branching variable is to be selected from the current node.

- ***WV:*** Which value should be tried first for a given branching variable.

Given a CSP and the user-defined rules, the model will create the search strategy by using the following algorithm:

① Place the root node in *L*

② While *L* is not empty use *WR* to select the next node *n*

③ If *n* is a goal node end

④ Apply *WB* on *n* to get branching variable *b*

⑤ Apply *WV* on *b* to get a binary value *v*

⑥ Use the algorithms in Section 4.5 to create new nodes *c*1 and *c*2 from *n*, such that *c*1 has branch *b* = *v* and *c*2 has branch *b* != *v*

⑦ Use *WI to* insert *c*1 and *c*2 in *L*

⑧ Goto Step ②

Although the above algorithm is sequential, it can be multithreaded by using Dynamic Thread Creation. In this case, we begin with one thread executing the algorithm. However, if at any time during the algorithm the active thread limit is not reached, the currently executing thread will create another thread that will start at step ②. In addition, all threads are functionally equivalent and thus can create other threads during the algorithm if the active thread limit is not reached.

Given the simple model outlined above, a programmer can define the parallel search strategy required for a given CSP. For example, to create a MT-LDS the programmer would define *WR* as being the first node in *L*, and *WI* as the head of *L* for *c*1 and the tail of *L* for *c*2. To create a multithreaded depth-first search, the programmer would define *WR* as being the first node in *L*, and *WI* as the head of *L* for *c*1 and the head + 1 of *L* for *c*2. The possibilities are inspiring.

7.3 Final Remarks

Application-based multithreading is an effective means for improving the performance of constraint programming on difficult combinatorial problems. In particular, through benchmark problems, we have shown that application-based multithreading can improve performance in three ways:

1. It can decrease the execution time of an application.

2. It can increase the size of problems that an application can solve.

3. It can improve the quality of solutions for an application that is limited by time.

In addition, application-based multithreading provides a flexible programming model that is realistic for industrial environments. It allows a programmer to implement the best parallel techniques for a given problem. What's more, the advent of relatively inexpensive symmetric multiprocessors, the adoption of threads into mainstream operating systems, and the commercial acceptance of constraint programming make application-based multithreaded constraint programming readily applicable, thus furthering its attractiveness.

Bibliography

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

[AMT,95] Arnow, D.; McAloon, K.; Tretkoff, C.: Parallel integer goal programming, *Proceedings of the 23rd Annual ACM Computer Science Conference*, Nashville, TN, 1995.

[Arnow,95] Arnow, D.: DP: A library for building portable, reliable distributed applications, *Proceedings of the USENIX Winter '95 Technical Conference*, pages 235-247, New Orleans, LA, 1995.

[Atay,92] Atay, C.: *A Parallelization of the Constraint Logic Programming Language 2LP*, Ph.D. Thesis, City University of New York, 1992.

[Baptiste,95] Baptiste, P.: Resource Constraints for Preemptive and Non-Preemptive Scheduling, MSc Thesis, University Paris VI, 1995.

[BB,87] Biswas, J.; Browne, J.C.: Simultaneous Update of Priority Structures, *Proceedings of the International Conference on Parallel Processing*, pages 124-131, 1987.

[BC,96] Beasley, J.E.; Chu, P.C.: A genetic algorithm for the set covering problem, *European Journal of Operational Research*, 94:392-404, 1996.

[BDM,94] Baker, S.; Donnelly, A.; McHale, C.; Walsh, B.: Synchronization Variables, Technical Report, Department of Computer Science, Trinity College, Number TCD-CS-94-01, 1994.

[Beasley,87] Beasley, J.E.: An algorithm for set covering problems, In: *European Journal of Operational Research*, 31:85-93, 1987.

[Beasley1,90] Beasley, J.E.: OR-Library: distributing test problems by electronic mail, *Journal of the Operational Research Society*, 41(11): 1069-1072, 1990. See also: http://mscmga.ms.ic.ac.uk/info.html

[Beasley2,90] Beasley, J.E.: A Lagrangian Heuristic for Set Covering Problems, *Naval Research Logistics*, 37(1):151-164, 1990.

[BH,80] Balas, E.; Ho, A.: Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study, *Mathematical Programming Study*, 12:37-60, 1980.

[BJ,92] Beasley, J.E.; Jörnsten, K.: Enhancing an algorithm for set covering problems, *European Journal of Operational Research*, 58:293-300, 1992.

[BL,96] Berg, D.J.; Lewis, B.: *Threads Primer: A Guide to Multithreaded Programming*, SunSoft Press, Upper Saddle River, NJ, 1996.

[BLL,94] Blumofe, R.D.; Leiserson, C.E.: Scheduling Multithread Computtaions by Work Stealing, *Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science*, San Diego, CA, 1994.

[BLPN,95] Basbtiste, P.; Le Pape, C.; Nuijten, W.: Incorporating Efficient Operations Research Algorithms in Constraint-Based Scheduling, In: *Proceedings of the First International Joint Workshop on Artificial Intelligence and Operations Research*, Timberline Lodge, OR, 1995.

[Borning,81] Borning, A.: The programming language aspects of ThingLab, a constraint oriented simulation laboratory, *ACM Transaction on Programming Languages and Systems*, 3(4):353-387, 1981.

[BS,81] Burton, F.W.; Sleep, M.R.: Executing functional programs on a virtual tree of processors, *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architectures*, 1981.

[Breuer,70] Breuer, M.A.: Simplification of the set covering problem with an application to boolean expressions, *Journal for the Association of Computing Machinery*, 17:166-181, 1970.

[Caseau,91] Caseau, Y.: An Object-Oriented Deductive Language**,** *Annals of Mathematics and Artificial Intelligence*, 3:211-258, 1991.

[CL1,96] Caseau, Y.; Laburthe, F.: Improving Branch and Bound for Jobshop Scheduling with Constraint Propagation, *Proceedings of the 8th Franco-Japanese 4th Franco-Chinese Conference CCS'95*, 1996.

[CL2,96] Caseau, Y.; Laburthe, F.: CLAIRE: Combining Objects and Rules for Problem Solving, *Proceedings of the JICSLP'96 workshop on multi-paradigm logic programming*, 1996.

[CNS,96] Ceria, S; Nobili, P.; Sassano, A.: A Lagrangian-Base Heuristic for Large-Scale Set Covering Problems, *Mathematical Programming*, 1996.

[Colmerauer,87] Colmerauer, A.: Opening the Prolog III Universe: a new generation of Prolog promise some powerful capabilities, *Byte*, August, 1987.

[CS,91] Mellor-Crummey, J. M.; Scott, M.L: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Transactions on Computer Systems*, February, 1991.

[Dantzig,63] Dantzig, G.B.: *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[da Silva,98] da Silva, A.: Using ILOG Solver for column generation: a successful marriage of constraint Programming and linear programming, *Proceedings of the 1998 ILOG Users' Conference*, Paris, France, 1998.

[DB,95] DeBacker, B.; Beringer, H.: Cooperating Solvers and Global Constraints: The Case of Linear Arithmetic Constraints, *Proceedings of ILPS' 95 Workshop, 1995.*

[DC,93] Diaz, D.; Codognet, P.: A Minimal Extension of the WAM for clp(FD), *Proceedings of the Tenth International Conference on Logic Programming*, pages 774-790, Budapest, Hungary, 1993.

[Diaz,91] Diaz, D.: Compilation de Prolog: étude et réalisation d'un outil extensible, Mémoire de D.E.A., Université d'Orléans, 1991.

[Dongarra,92] Dongarra, J.J.: Performance of various computers using standard linear equation software, *Computer Architecture News*, pages 22-44, 1992.

[DRKP,91] De Boer, F. S.; Rutten, J.J.M.M.; Kok, J.N.; Palamidessi, C.: Semantic models for concurrent logic languages, *Theoretical Computer Science*, 86(1):3-33, 1991.

[DVSAGB,88] Dincas, M.; Van Hentenryck, P.; Simonis, H.; Aggoun, A.; Graf, T.; Berthier, F.: The Constraint Logic Programming Language CHIP, *Proceedings of International Conference on Fifth Generation Computing Systems*, 1988.

[Etcheberry,77] Etcheberry, J.: The set covering problem: A new implicit enumeration algorithm, *Operations Research*, 13:760-772, 1977.

[EL,92] El-Rewini, H.; Lewis, T.G.: *Introduction to Parallel Computing*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[EK,91] Ecker, J.G.; Kupferschmid, M.: *Introduction to Operations Research*, Krieger Publishing Company, Malabar, FL, 1991.

[Frühwirth,95] Frühwirth, T.: Constraint Handling Rules, *Constraint Programming: Basic and Trends*, pages 90-107, Springer-Verlag, New York, NY, 1995.

[GBDJMS,94] Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R.; Sunderam, V.: *PVM: Parallel Virtual Machine-A User's Guide and Tutorial for Network Parallel Computing*, MIT Press, Cambridge, MA, 1994.

[GGKK,94] Grama, A.; Gupta, A.; Karypis, G.; Kumar, V.: *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings, Redwood City, CA, 1994.

[GH,91] Ginsberg, M.L.; Harvey, W.D.: Iterative Broadening, *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 210-215, 1991.

[GJ,79] Garey, M.R.; Johnson, S.J.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, NY, 1979.

[GKP,93] Grama, A.; Kumar, V.; Pardalos, P.: Parallel Processing of Discrete Optimization Problems, *Encyclopedia of Microcomputers*, Wiley, New York, NY, 1993.

[Haridi,97] Haridi, S.: Tutorial of OZ 2 and the DFKI OZ System, DFKI Documentation Series, Available at: http://www.sics.se/~seif, 1997.

[HE,80] Haralick, R.; Elliott, G.: Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence*, 1980.

[Henz,98] Henz, M.: Constraint based round robin tournament planning, Technical Report, Department of Information Systems and Computer Science, National University of Singapore, 1998.

[HG,96] Harvey, W.D.; Ginsberg, M.L.: Limited Discrepancy Search, *Proceedings of the Fourteenth International Conference on Artificial Intelligence*, pages 607-615, 1995.

[HMPS,96] Hunt, G. C.; Michael, M. M.; Parthasarathy, S.M.; Scott, M.L.: An Efficient Algorithm for Concurrent Priority Queue Heaps, *Information Processing Letters*, November, 1996.

[HSW,93] Henz, M.; Smolka, G.; Würtz, J.: OZ-A programming language for multi-agent systems, *13th International Joint Conference on Artificial Intelligence*, pages 404-409, Chambery, France, 1993.

[ILOG-CPLEX,98] ILOG CPLEX Reference Manual, 1998.

[ILOG-OPT,98] ILOG Optimization Suite, White Paper, ILOG 1998.

[ILOG-Plan,98] ILOG Planner Reference Manual, ILOG 1998.

[ILOG-Sched,98] ILOG Scheduler Reference Manual, ILOG 1998.

[ILOG-Sol,98] ILOG Solver Reference Manual, ILOG 1998.

[JL,87] Jaffar, J.; Lassez, J.-L.: Constraint Logic Programming, *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111-119, Munich, Germany, 1987.

[Jones,89] Jones, D.W.: Concurrent Operations on Priority Queues, *Communications of the ACM*, 32(1):132-137, 1989.

[JM,87] Jaffar, J.; Michaylov, S.: Methodology and Implementation of a CLP System, *Proceedings of the 4th International Conference on Logic Programming*, pages 196-218, Melbourne, Australia, 1987.

[KGR,94] Kumar, V.; Grama, A.; Rao, V. N.: Scalable load balancing techniques for parallel computers, *Journal of Distributed Computing*, 22(1):60-79, March, 1994.

[Korf,85] Korf, R.E.: Depth First Iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, 27:97-109, 1985.

[KRR,94] Kumar, V.; Rao, V. N,; Ramesh, K.: Parallel Best-First Search of State-Space Graphs: A Summary of Results, *Proceedings of the 1988 National Conference on Artificial Intelligence*, 1988.

[KSS,96] Kleiman, S.; Shah, D.; Smaalders, B.: *Programming With Threads*, SunSoft Press, Upper Saddle River, NJ, 1996.

[Kumar,92] Kumar, V.: Algorithms for Constraint Satisfaction Problems: A Survey, *AI Magazine*, 13(1):32-44, 1992.

[KW,90] Kuchen, H.; Wagener, A.: Comparison of Dynamic Load Balancing Strategies, Technical Report, Technical University of Aachen (RWTH Aachen), Number 90-05, 1990.

[Laurière,78] Laurière, J.L.: Alice: A Language and a Program for Stating and Solving Combinatorial Problems, *Artificial Intelligence,* 10:29-127, 1978.

[LB,97] Le Pape, C., Baptiste, P.: Heuristic Control of a Constraint-Based Algorithm for the Preemptive Job-Shop Scheduling Problem, To appear in: *Journal of Heuristics*, 1998.

[LBD,88] Lusk,E; Butler, R.; Disz, T.; Olson, R.; Overbeek, R.; Stevens, R.; Warren, D.H. D.; Calderwood, A.; Szeredi, P.; Haridi, S.; Brand, P.; Carlsson, M.; Ciepielewski, A.; Hausman, B.; The Aurora Or-Parallel Prolog System, *Proceedings of the International Conference on Fifth Generation Computer Systems*, 3:819-830, Springer-Verlag, New York, 1988.

[Lemke,96] Lemke, J.: A production planning system for a steel plant, *Proceedings of ILOG Solver and ILOG Schedule 2nd International Users Conference*, Paris, France, 1996.

[LePape,94] Le Pape, C.: Constraint-Based Programming for Scheduling: An Historical Perspective, *Working Notes of the Operations Research Society Seminar on Constraint Handling Techniques*, 1994.

[LRP,95] Le CUN, B., Roucairol, C.; The PNN Team: BOB: a Unified Platform for Implementing Branch-and-Bound like Algorithms, University of Versailles, Research Report, Number 96/16, 1996.

[LS,84] Lai, T.H.; Sahni, S.: Anomalies in parallel branch and bound algorithms, *Communications of the ACM*, pages 594-602, 1984.

[Maher,87] Maher, M.: Logic Semantics for a class of committed-choice programs, *Logic Programming: Proceedings of the 4th International Conference*, pages 858-876, Melbourne, Australia, 1987.

[MAS,98] Marriot, K.; Stuckey, P.: *Programming with Constraints: An Introduction*, MIT Press, Cambridge, MA, 1998.

[MD,92] Mahanti, A.; Daniels, C.: SIMD parallel heuristic search, *Artificial Intelligence*, 1992.

[MF,93] Mackworth, A.; Freuder, E.: The complexity of constraint satisfaction revisited, *Artificial Intelligence*, 25:65-74, 1993.

[MOZART] The Mozart Programming System, URL: http://www.mozart-oz.org/

[MP,93] Morton, T.E.; Pentico, D.W.: *Heuristic Scheduling Systems with Applications to production Systems and Project Management*, Wiley, New York, NY, 1993.

[MPI,94] Message Passing Interface Forum, MPI: A message passing interface standard, *The International Journal of Super Computer Applications and High Performance Computing*, 8(3-4):159-416, 1994.

[MR,95] Montanari, U.; Rossi, F.: A Concurrent Semantics for Concurrent Constraint Programs via Contextual Nets, *Principles and Practices of Constraint Programming*, pages 3-27, MIT Press, Cambridge, MA, 1995.

[MRS,94] Montanari, U.; Rossi, F.; Saraswat, V.: CC Programs with both In- and Non-Determinism: A Concurrent Semantics, *Lecture Notes in Computer Science*, 1994.

[MS,94] Mudambi, S.; Schimpf, J.: Parallel CLP on Heterogeneous Networks, *Proceedings of the 11th International Conference on Logic Programming*, pages 124-141, Ligure, Italy, 1994.

[MSDN] Microsoft: Microsoft Developer Network, http://www.microsoft.com/msdn

[MSL,96] Michael, M.M.; Scott, M.L.: Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, Proceedings of the *15th ACM Symposium on Principles of Distributed Computing*, pages 267-275, 1996.

[MSP,95] Moreira, J.E.; Schouten, D.A.; Polychronopoulos, C. D.: The Performance Impact of Granularity Control and Functional Parallelism, IBM Research Report, 1995.

[MT,95] McAloon, K.; Tretkoff, C.: 2LP: Linear Programming and Logic, *Principles and Practices of Constraint Programming*, pages 101-116, MIT Press, Cambridge, MA, 1995.

[MT,96] McAloon, K.; Tretkoff, C.: *Optimization and Computational Logic*, Wiley, New York, NY, 1996.

[MTW,97] McAloon, K.; Tretkoff, C.; Wetzel, G.: Sports League Scheduling, *Proceedings of the 1997 ILOG Optimization Suite International Users' Conference*, Paris, France, 1997.

[MTW,98] McAloon, K.; Tretkoff, C.; Wetzel, G.: Disjunctive Programming and Cooperating Solvers, *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search*, pages 75-96, Kluwer, Boston, MA, 1998.

[Muller,97] Muller, L.G.: *Parallelism Through Multithreaded Programming In A Constraint-Based System*, Ph.D. Thesis, City University of New York, 1997.

[NA,95] Nuijten, W.; Aarts, E.: A Computational Study of Constraint Satisfaction for Multiple Capacitated Job Shop Scheduling, *European Journal of Operational Research*, 90(2):269-284, 1996.

[Nuijten,94] Nuijten, W.P.M.: Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach, Ph.D. Thesis, Eindhoven University of Technology, 1994.

[Pai,95] Pai, D.K.: Robot Programming and Constraints, *Principles and Practices of Constraint Programming*, pages 71-83, MIT Press, Cambridge, MA, 1995.

[PB,90] Patil, S.; Banerjee, P.: A parallel branch and bound algorithm for test generation, *IEEE Transactions on Computer Aided Design*, 9(3):313-332, 1990.

[Pountain,95] Pountain, D.: Constraint Logic Programming, *Byte*, February, 1995.

[Prasad,96] Prasad, S.: Weaving a Thread, *Byte*, October, 1996.

[PS,82] Papadimitriou, C. H.; Steiglitz, K.: *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Puget,94] Puget, J.F.: A C++ implementation of CLP, *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore, 1994.

[Richter,94] Richter, J.: *Advanced Windows NT: The Developer's Guide to the Win32 Application Programming Interface*, Microsoft Press, Redmond, WA, 1994.

[RK,87] Rao, V.N.; Kumar, V.: Parallel Depth-First Search on Multiprocessors Part I: Implementation, *International Journal of Parallel Programming*, 16(6):479:499, 1987.

[RK,88] Rao, V.N.; Kumar, V.: Concurrent Access of Priority Queues, *IEEE Transactions on Computers*, 37(12):1657-1665, 1988.

[RK,93] Rao, V.N.; Kumar, V.: On the Efficiency of Parallel Backtracking, *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427-437, 1993.

[RogueWave] RogueWave Software, Writing Multithreaded Applications with Threads.h++ , White Paper, See Also: http://www.roguewave.com/products/threads/

[Rubin,73] Rubin, J.: A technique for the solution of massive set-covering problems with applications to airline crew scheduling, *Transportation Science*, 7:34-48, 1973.

[RW,98] Rodosek, R.; Wallace, M.: One model and different solvers for hoist scheduling problems, IC-PARC Technical Report, 1998.

[Saraswat,93] Saraswat, V.A.: *Concurrent Constraint Programming*, The MIT Press, Cambridge, MA, 1993.

[Selman,96] Selman, B.: Computational Challenges in Artificial Intelligence, *ACM Computing Surveys*, 1996.

[Schmitdt,94] Schmidt, D.C.: The ADAPTIVE Communication Environment Object-Oriented Network Programming components for Developing Client/Server Applications, *Proceedings of the 12th Annual SUN Users Group Conference*, pages 214-225, 1994.

[Schmidt,95] Schmidt, D.C.: A OO Encapsulation of Lightweight Concurrency Mechanisms in the ACE Tool Kit, Technical Report, Washington University, Computer Science, Number WU-CS-95-31, 1995.

[SR,90] Saraswat, V.A.; Rinard, M.: Concurrent constraint programming, *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 232-245, San Francisco, CA, 1990.

[SRP,91] Saraswat, V.A.; Rinard, M.; Panangaden, P.: Semantic Foundations of Concurrent Constraint Programming, *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 333-352, Orlando, FL, 1991.

[Steele,80] Steele, G.: *The Definition and Implementation of a Computer Programming Language Based on Constraints*, Ph.D. Thesis, Massachusetts Institute of Technology, 1980.

[SUN-MA,91] SunSoft: Solaris SunOS 5.0 Multithread Architecture, A White Paper, 1991.

[SUN-MPG,94] SunSoft: Multithread Programming Guide, A User Manual, 1994.

[SUN-MIC,96] SunSoft: Multithreaded Implementations and Comparisons, A White Paper, 1996.

[Sutherland,63] Sutherland, I.: Sketchpad: a man-machine graphical communication system, *Proceedings of the IFIP Spring Joint Conference*, pages 329-346, Detroit, Michigan, 1963.

[SW,96] Stein, C.; Wein, J.: On the Existence of Schedules that are Near-Optimal for both Makespan and Total Weighted Completion time, Technical Report, Dartmouth College, Computer Science, Number PCS-TR96-295, July 1996.

[Thompson,96] Thompson, T.: The World's Fastest Computers, *Byte*, January, 1996.

[TP,96] Tschöke, S.; Polzer, T.: Portable Parallel Branch-and-Bound Library: User Manual Library Version 2.0, University of Paderborn, 1996.

[TSRB,71] Toregas, C.; Swain, R.; Revelle, C.; Bergman, L.: The location of emergency service facilities, *Operations Research*, 19:1363-1373, 1971.

[Tsang,93] Tsang, E.: *Foundations of Constraint Satisfaction*, Academic Press, San Diego, CA, 1993.

[Tucker,93] Tucker, A.: *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*, Ph.D. Thesis, Stanford University, 1993.

[VanHen1,89] Van Hentenryck, P.: Parallel constraint satisfaction in logic programming: Preliminary results of CHIP within PEPSys, *Proceedings of the Sixth International Conference on Logic Programming*, pages 165-180, Lisbon, Portugal, 1989.

[VanHen2,89] Van Hentenryck, P.: *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA, 1989.

[VSD,91] Van Hentenryck, P.; Saraswat, V.; Deville, Y.: Constraint Processing in cc(FD), Technical Report, Brown University, 1991.

[VSRL,93] Véron, A.; Schuerman, K.; Reeve, M.; Li, L.: Why and How in the ElipSys OR-Parallel CLP System, *Proceedings of Parallel Architectures and Languages Europe*, pages 291-304, Munich, Germany, 1993.

[Wallace,96] Wallace, M.G.: Practical Applications of Constraint Programming, *Constraints: An International Journal*, 1(1-2)*:*139-168, 1996.

[Warren,83] Warren, D.H.D.: An Abstract Prolog Instruction Set, Artificial Intelligence Center, Computer Science and Technology Division, Number 309, 1983.

[Wetzel,97] Wetzel, G.: *Abductive and Constraint Logic Programming*, Ph.D. Thesis, Imperial College, 1997.

[WLY,85] Wah, B.W.; Li, G.J.; Yu, C.F.: Multiprocessing of combinatorial search problems, *IEEE Computers*, 18(6):93-108, 1985.

[WNS,97] Wallace, M.G.; Novello, S; Schimpf, J.: ECLiPSe : A Platform for Constraint Logic Programming, *ICL Systems Journal*, 12(1), May 1997.

[WZ,98] Wetzel, G.; Zabatta, F.: A Constraint Programming Approach to Portfolio Selection, *Proceeding of The 13th biennial European Conference on Artificial Intelligence*, Brighton, United Kingdom, 1998.

[ZY1,98] Zabatta, F., Ying, K.: A Thread Performance Comparison: Windows NT and Solaris On a Symmetric Multiprocessor, *Proceedings of The 2nd USENIX Windows NT Symposium*, Seattle, WA, 1998.

[ZY2,98] Zabatta, F., Ying, K.: Dynamic Thread Creation: An Asynchronous Load Balancing Scheme For Parallel Searches, *Proceedings of 10th International Conference on Parallel and Distributed Computing and Systems*, pages 20-24, Las Vegas, NV, 1998.