UNIVERSITY OF CINCINNATI

Date: 11/13/2006

I, Michal Kouril,

hereby submit this work as part of the requirements for the degree of:

Doctor of Philosophy (Ph.D.)

in:

Computer Science and Engineering

It is entitled:

A Backtracking Framework for Beowulf Clusters with an Extension to Multi-cluster Computation and SAT Benchmark Problem Implementation

This work and its defense approved by:

Chair: Dr. Jerome Paul

Dr. Kenneth Berman

Dr. John Franco

Dr. John Schlipf

Dr. Frank Pinski

A Backtracking Framework for Beowulf Clusters with an Extension to Multi-cluster Computation and SAT Benchmark Problem Implementation.

A dissertation submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in the Department of
Computer Science
of the College of Engineering

17th November, 2006
by

Michal Kouril

Bachelor (Information Technology)
University of Economics,
Prague, Czech Republic, 1997

Thesis Advisor and Committee Chair: Dr. Jerome Paul

Abstract

The main topic of this dissertation involves cluster-based computing, specifically relating to computations performed on Beowulf clusters. I have developed a light-weight library for dynamic interoperable message passing, called the InterCluster Interface (ICI). This library not only supports computations performed over multiple clusters that are running different Message Passing Interface (MPI) implementations, but also can be used independently of MPI. In addition I developed the Backtracking Framework (BkFr) that simplifies implementations of the parallel backtracking paradigm in the single cluster environment, and supports the extension of computations over multiple clusters. BkFr uses MPI for the intra-cluster communication and ICI for the inter-cluster communication. I have also developed a template-based library of programming modules that facilitate the introduction of the rapidly emerging message passing parallel computing paradigm in upper-division undergraduate courses.

An important application of ICI and the backtracking framework discussed in this dissertation is the computation of Van der Waerden numbers, which involve the existence of arithmetic progressions in arbitrary partitions of {1, 2, …, n} for sufficiently large n. The computations of these numbers utilized a special SAT solver that I developed. For example, I almost doubled the previously known lower bound for the Van der Waerden number W(2, 6), and I am running a calculation whether the lower bound is actually the exact value. Among other reductions, I was able to drastically reduce the search for potential solutions with the aid of a tunneling technique based on an aggressive addition of uninferred constraints. The tunneling technique has the likely potential to be used in a number of other satisfiability settings. I also developed an FPGA version of my SAT solver for Van der Waerden numbers that improved the search speed up to 230 times or more compared to its sequential equivalent. Utilizing this special SAT solver has resulted in significant progress in the search to determine bounds or exact values for Van der Waerden numbers and, more generally, for Van der Waerden subnumbers. I have found the exact values for several of these subnumbers, specifically W(2; 5,6)=206, W(2; 4,8)=146, W(2; 3,16)=238 and W(3; 2,4,7)=119.

Acknowledgements

I would like to thank my advisor and thesis chair Professor Jerry Paul for his guidance, inspiration, patience, and support during my research. It would be hard to imagine my degree and dissertation without his input and help. I'm very grateful to him.

I've also been fortunate in working with John Franco, who introduced me to the SAT community and put some of my ideas into perspective. His door was always open. (We also had a lot of fun running the ACM contest in the past 7 years.)

Professors John Schlipf, Kenneth Berman, Fred Annexstein, and Karen Tomko also contributed to my work through our insightful discussions and encouragement throughout my graduate years.

I'm also indebted to the funding agencies that kept me afloat—the Ohio Board of Regents for the fellowship that gave me the means to finish my research, as well as the Department of Defense, specifically Mark Vanfleet and Michael Dransfield, for allowing me to work on the SAT solver SBSAT, where I first became aware of Van der Waerden numbers.

The Department of ECECS and now CS has been a great place to conduct my research, one that is singular in its friendly environment, and one that I hope to find again. I'm grateful to the technical staff, Rob Montjoy and Chris Isbell, for their help and dedication to keeping the technology running 24/7.

Thanks to my fellow graduate student Sean Weaver for helping me with the equivalence check of the FPGA code using the cryptol tools, as well as acting as a sounding board for my ideas.

Finally, I'd like to thank my wife for her editorial support, and enduring and sharing the life of a graduate student—and although not fully understanding what I'm doing, trusting me that it is important.

Contents

List of Figures

List of Tables

Chapter 1 - Introduction

In recent years a new paradigm of computing has emerged, namely *cluster computing*. In this paradigm, off-the-shelf processors, typically rack mounted, are connected using generic network components such as Ethernet[1], or specialized equipment such as Myrinet[2] or Infiniband[3]. The name for such a configuration is a *Beowulf cluster*[4], and it is economically feasible to build such clusters containing hundreds of processors. Moreover, by connecting together multiple clusters even more computing power can be harnessed.

To provide a common environment for programs developed for not only Beowulf clusters but also other multiprocessor machines, the message passing paradigm has been standardized into the message passing interface (MPI)[5, 6]. MPI is implemented by a number of commercial vendors, as well as open-source projects such as MPICH[7], OpenMPI[8] and LAM[9]. MPI provides a library of functions augmenting existing high level languages such as Fortran, C, or C++. Communication between processors is implemented by point-to-point send and receive instructions, and global operations such as

broadcast, scatter, and so forth, are also supported. In addition to these basic communication instructions, MPI provides a large library of instructions that provide the user with a programming environment supporting cluster-based computations. With this new paradigm problems can be attacked that heretofore were impractical due to their computation time complexity.

To bring even more computational power into play, it would be very advantageous to be able to connect multiple clusters together in a dynamic fashion. By *dynamic* is meant the possibility to connect and disconnect two independently running clusters. However, the MPI-1 Standard[5] does not define any interface for dynamic communication between two clusters. The MPI-2 Standard[6] does define dynamic communication between two clusters, but the communication protocol is not standardized, thereby rendering dynamic communication impossible between MPI libraries of different origin. In other words, the MPI-2 Standard is not *interoperable*, and only allows communication between clusters using the same version of MPI. The Interoperable MPI (IMPI) Standard[10] does define the communication protocol for connecting multiple clusters running different versions of MPI. However, the participating clusters must be captured at the outset of the computation, thereby creating a static environment with the inability to connect and disconnect cluster during the computation.

One of the main topics in this dissertation is my development of a communication library (InterCluster Interface, or ICI)[11, 12]) that allows multiple clusters to be connected in a dynamic and interoperable manner. More precisely, ICI allows clusters running different versions of MPI (interoperability) to dynamically join an ongoing computation. Of course, this most naturally supports problems amenable to asynchronous computation, such as parallel depth-first search or parallel backtracking.

The problem of maintaining efficient load balancing when performing cluster-based computation is becoming a major issue. The ever-increasing number of nodes puts a high demand on the way applications are parallelized. In response I developed a tool that supports efficient parallelization of the various problems. This tool, which is a second major topic of this dissertation, is a backtracking

framework (BkFr)[13] that simplifies implementation of the parallel backtracking paradigm in the single cluster environment, and supports the extension of computations over multiple clusters. Problems implemented in this framework can not only make use of all nodes in a single cluster, but also can dynamically span multiple clusters. Communication within a single cluster is provided using MPI and communication between clusters utilizes the above-mentioned ICI library.

The message passing paradigm can be considered to be a fundamental component of the CS curriculum. However, often the undergraduate CS curriculum is too constrained to devote an entire course to this topic. A third major topic of this dissertation is my development of a template library of problems[14] amenable to parallel backtracking as supported by MPI and ICI. This template library facilitates a rapid introduction to the message passing paradigm and allows an instructor to introduce these important ideas as one topic amongst many in a typical algorithms course. Indeed, my belief that all CS undergraduates should be exposed to the message passing paradigm provided the motivation for developing this template library.

Satisfiability (or *SAT*)[15]) refers to the problem of determining the existence of an assignment of values to the variables occurring in a given Boolean expression that makes the expression true (called a *satisfying assignment*). Satisfiability is a fundamental problem in computer science, since solutions to many important problems, such as circuit testing, combinatorial search, and so forth, can be formulated as satisfiability questions. Satisfiability is also fundamental to the theory of computation, since in 1971 Cook proved that SAT is NP-complete[16]. Programs for solving Satisfiability problems are called *SAT solvers*, and there is a very active research community working on improving their efficiency.

A fourth major topic of this dissertation is my design of a highly efficient SAT solver that uses parallel backtracking to address the problem of computing Van der Waerden numbers W(K,L)[17]. More specifically, given positive integers K, L, Van der Waerden proved that there exists a (smallest) integer n = W(K,L) such that whenever {1,2, …, n} is K-colored, there exists a monochromatic arithmetic progression of length L in {1, 2, …, n}. An arithmetic progression is defined as a sequence of numbers

such that the difference between any two successive numbers of the sequence is a constant[18]. These numbers were shown to exist by B.L. Van der Waerden in the 1920s, but their exact values are known for a handful of cases.

By showing that $W(2,6) \geq 1132$, I have almost doubled the previously known lower bound of 696. This calculation was performed using my tunneling technique[19] described in more detail in Chapter 5 of this dissertation. I hope to complete the computation for the number W(2,6) (W(2,K) only currently known for K = 2, 3, 4 and 5). Determining the exact value of W(2,6) will be a breakthrough, since the last Van der Waerden number was found more than two decades ago in 1979.

A Van der Waerden *sub number* $W(K; L_1, L_2,\ldots L_K)$ is defined analogously to the Van der Waerden number W(K,L), but a different L is given for each color. Using my SAT solver I have also improved known lower bounds for several sub numbers.

In addition to Beowulf clusters, reconfigurable hardware is also gaining popularity. I call reconfigurable hardware refers to a device (typically an integrated circuit) that can be programmed on the gate level at run-time[20]. While there are several types of reconfigurable hardware, I will focus on Field Programmable Gate Arrays (FPGAs). I have employed FPGAs to aid in the completion of the computation of W(2,6) with significant speedup over available clusters.

The dissertation is organized into five chapters: The second chapter contains the background for various concepts used in the dissertation. The third chapter contains the details of the Backtracking Framework (BkFr), including how it is applied, and how it is used for teaching MPI in the classroom. The fourth chapter includes a description and discussion of the main properties of the InterCluster Interface library, which provides the dynamic interoperable connection for BkFr. The fifth chapter focuses on the problem of computing Van der Waerden numbers, finding lower bounds for these numbers, and specifically describing the search for W(2,6). Chapter six contains a description of how I extend the computer-based Van der Waerden number-specific SAT solver to reconfigurable hardware – that is, to FPGAs. In chapter seven I summarize my research and discuss future directions.

Chapter 2 - Background

In this chapter I describe some of the key concepts used throughout the dissertation. I begin by expanding my description given in the previous chapter of Beowulf clusters and associated programming environment. Both single cluster and multiple cluster environments are discussed. In particular, I describe the limitations imposed by the current communication protocols for connecting clusters together. I then describe the parallel backtracking design strategy, and how I utilize it to attack difficult combinatorial problems such as testing the satisfiability of CNF formulas, and determining lower bounds or exact values for Van der Waerden numbers.

2.1 Beowulf Clusters

A *Beowulf cluster* refers to a collection of processors (nodes) connected in a network usually using Ethernet, but new protocols and systems such as Myrinet are beginning to see widespread use. Communication between nodes in the clusters is typically done via message passing. The nodes in the

cluster are off-the-shelf general purpose processors, each having their own local memory. Message passing parallel languages such as MPI (Message Passing Interface) contain a library of functions, which extend ordinary sequential languages such as Fortran, C, or C++, and allow the programmer to regard the processors as connected in a complete graph topology. In other words, commands in the language exist for sending a message between any pair of processors in the cluster, and no routing needs to be explicitly coded, in contrast to interconnection network parallel machines such as meshes or hypercubes. Of course, while communication between clusters is straightforward in code, nevertheless it has high latency compared to local computations performed by the processors. Thus, the design of efficient implementation of parallel algorithms on clusters requires careful attention to achieving a good balance between computation and communication. I am concerned with efficient utilization of a cluster or set of clusters for problems that are amenable to a solution using a generalized backtracking algorithm, and for which it is possible to efficiently control computation and communication phases in order to achieve good speedup that also scales well.

General purpose CPUs are getting faster and faster. According Moore's Law[21], the component density and related performance of integrated circuits roughly doubles every 18 months. The continuation of Moore's law beyond a decade or so is questionable[22], since it requires miniaturization approaching the atomic level. Hence, my best hope for continued growth in computation power seems to lie in utilizing hundreds or thousands of processors executing some form of parallelism. Among the fastest supercomputers today are also clusters built from general purpose CPUs – Beowulf clusters. The first Beowulf cluster was built in 1994[4] when the interconnection network evolved, allowing for adequate communication having sufficient bandwidth[1] and low latency[2]. Connecting hundreds or even thousands of CPUs accumulates unprecedented raw power.

[1] Communication bandwidth measures the amount of data transferred per second (e.g.: 100 megabits per second)
[2] Communication latency measures the time it takes to transfer data. It is measured from the time the application starts the transfer to the time the receiving application gets the data.

2.2 Parallel vs. Distributed Processing

Parallel processing typically involves connecting sequential CPUs (processors) under tight synchronization managed by a central control to solve a given task. Often one processor cannot proceed with its assigned computation until it obtains a result from another processor, requiring frequent synchronization among the processors. For some tasks, such as simulations of physical processes, synchronization constraints are necessary and cannot be dropped. Due to the architecture of the general processor, and the significant difference between computation speed and communication speed, the longer a computation can proceed without waiting for synchronization with other processors, the greater utilization of all the processors is achieved.

In the distributed computing scenario, processors are not tightly synchronized, but rather can proceed relatively asynchronously working on that part of a given problem assigned to them. Synchronization, when it does occur, can often be limited to relatively short intervals between computation phases.

2.3 The Single Cluster

A single cluster, being homogenous, possesses certain properties that can be integrated into a computational scheme. For example, the number of nodes stays the same for the entire time a task is running. The nodes within the cluster usually have the same or similar speed, memory, and hardware. Depending on the communication library, every node has a unique ID, and nodes communicate in a fairly simple way using send and receive instructions. Some of the libraries offer special functions for global operations such as scatter, gather, and so forth.

A cluster usually has a standard way of running applications. For example, a batch system might be implemented using one or more job queues, where jobs sit and wait for the scheduler to assign it free resources (e.g., a subset of the nodes in the cluster). If there are no free resources available immediately,

the job is put into a queue. Batch systems also implement restrictions on the number of resources the job can require, including the maximum time it can occupy the cluster. Batch systems also can manage computers with different hardware configurations and make sure that the job requirements, which are submitted with the job, match with the actual computer the job runs on. There also might be different queues with different priorities and different access controls. Jobs of different users might be entitled to different treatment. Batch systems also offer additional services such as accounting, e-mail notification, and so forth. They also ensure that the application has dedicated access to the resources.

Another way of running an application on a cluster is simple sharing without a scheduler. A user can run an application anytime, but there is no guarantee that other applications won't be running at the same time.

2.4 The Cluster Hardware/Software Environment

The Beowulf cluster, as previously mentioned, consists of off-the-shelf computers connected together. These computers are dedicated to running user applications typically without monitor and keyboard, and it is common practice to store them in the rack cabinets for space efficiency. The cluster is usually homogeneous – consisting of the same or similar computers, i.e., the same number of CPUs per computer, the same CPU speed, the same model of CPU, the same amount of memory, and so forth. It is possible to build a cluster from heterogeneous components, but this results in decreased predictability of behavior in disparate parallel applications.

Typically the clusters run Unix operating system such as Linux, AIX, or Solaris. Clustering using Windows is possible as well. TCP/IP is the underlying communication protocol for most of the communication libraries. The vast majority of clusters connect to the Internet also using TCP/IP protocol. From the programmer's point of view, several communication libraries support the cluster environment, i.e., communication among the nodes. The most commonly used communication library for clusters is probably MPI (Message Passing Interface). As the name indicates, the underlying technique relies on

passing messages between nodes. The current standard is MPI 1.1, and the newly created version—and in some packages already supported—is MPI 2.0. Older than MPI is the PVM (Parallel Virtual Machine) standard, which treats every node as a virtual machine. Other methods not closely related to the clusters, but allowing communication between computers, are RPC (Remote Procedure Call), as a standard part of every Unix implementation, Java-only implementation of RMI (Remote Method Invocation), Microsoft's Distributed COM, Corba, etc.

2.5 Multiple Clusters

Sometimes a single cluster is not big enough or powerful enough to finish a task in a reasonable time, or there are not enough resources (such as memory) to start the task. Access to several clusters then would be desirable. But clusters usually have different properties and are not available at exactly the same time. Although several available systems do assume tight synchronization and simultaneous availability, they have certain limitations. The fact that the user has to reserve these resources in advance when they are likely to be geographically separated is causing serious challenges for cluster scheduling. Another reason for concern is that the failure of a single node in a cluster that might contain a thousand nodes causes the simulation to fail. The development of my framework does not count on such systems being available. The framework targets clusters that are available at arbitrary times – essentially a more unfriendly environment. There are also different communication aspects of inter-cluster and intra-cluster communication. The available bandwidth between clusters is much smaller than the bandwidth of the interconnection network within the cluster, and the latency is much higher. Finally, there is a need for fault tolerance and security. Interruption of communication between two nodes in the Internet is more likely than inside the cluster. In the cluster, interruptions are usually caused by a hardware failure, as opposed to the Internet, where routing tables change dynamically and frequently, and disconnections or timeouts are likely.

2.6 Connecting Clusters

The idea of connecting clusters is not as well researched as intra-cluster communication. Although it's popular to create a computation grid that might consist of several clusters, most of the time it only refers to how the jobs are submitted and not to the actual implementation of inter-cluster communication. For example, one might submit code to a computation grid asking for a cluster of 64 nodes. But if there is no such cluster as a part of that grid, the code will not get executed even if the total number of nodes is more than 64. If, for example, 32 node clusters are available, then of course it is possible to submit two smaller jobs requesting 32 nodes each. However, since most of the grids operate under batch systems, one has no way of predicting when the two jobs get executed—much less expecting that they will execute at the same time. The only jobs that would properly execute under this scenario are jobs whose work can be divided into parts that don't require communication between the parts. If communication is required between the two jobs during execution, usually the only available way of communicating between such clusters is Berkeley TCP/IP API. Even when API is available on the entire system (and this is not always the case), a new API function for the communication between clusters will have to be introduced that will deviate from the MPI standard. One goal of this project is to design an extension to the MPI standard extension so that this 'special' API won't be necessary.

Libraries such as Globus can be used to create a global network of computers called a *computational grid*. Every computer is uniquely identified and public/private key certificates insure the security of the system. The main parts of Globus are GRAM (Globus Resource Allocation Manager), GIS (Grid Information Service) and GSI (Grid Security Infrastructure). One of the MPI packages, MPICH, has an extension for Globus called MPICH-G2 where it is possible to run MPI programs over Globus nodes. However, there seems to be a problem in running the programs (MPI) using a homogeneous API interface (MPI interface) over a non-homogeneous underlying hardware (made transparent for MPI due to Globus). Unless the assignment of nodes between physical nodes and MPI logical nodes is given, it is difficult to tune the global cluster accordingly. However, this assignment is not available to the programmer.

Another system for implementing a computational grid is called Condor[23] with an extension for Globus called Condor-G. “Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.”[24] However, Condor suffers from the same type of limitations that were discussed in connection with MPICH-G2.

Other frameworks are available for scientific computation. One of the most successful is Cactus[25]. It can run over Globus, but exhibits the problem with heterogeneous underlying architecture mentioned earlier in this chapter[26]. This framework does not efficiently support the multi-cluster functionality that is a feature of my *Backtracking Framework*.

2.7 Backtracking

Many important problems are amenable to the backtracking algorithm, and are good candidates for parallelization as implemented in the cluster environment.

A solution to these problems involves making a sequence of decisions, and the associated *state space tree* for the problem simply models all possible decision sequences. Backtracking refers to a systematic algorithm for searching for a solution in a state space tree modeling a given problem. One starts at the root of the tree (the empty decision), and generates a path in the tree consisting of a decision sequence. If a leaf is hit without finding a solution, or the path is bounded through some constraint, the algorithm would *backtrack* to the first node in the tree where an alternate decision could be made, and continues the search from there. Backtracking is particularly well suited for parallelization since the state space tree can be divided into a *covering set of initial paths* starting at the root, and each initial path can be assigned to a different computing node (distributed backtracking). A *covering set of initial paths* is

simply a set of paths starting at the root, such that an initial segment of any path from the root to a leaf agrees with (exactly) one of the initial paths in the covering set. The *state space tree* is divided into disjoint subtrees rooted at the endpoints of the paths in the covering set, and the computing nodes would then search their own subtree of the tree independently of other nodes. However, it is important that information be shared among the nodes which might help restrict (bound) the search, or when a node has exhausted its part of the search, and can then help another node in its search by splitting the node's subtree.

In Figure 1 a covering set of six initial paths for a binary state-space tree is shown, together with the associated subtrees assigned to computing node. Figure 2 illustrates the current path generated by node 2, and the subtree that would be sent to a helper node if node 2's subtree would be split. It is assumed that a left child is visited first in the backtracking search. Since the path with the backtrack point (BkPt) determines the subtree, the phrases "splitting the path" and "splitting the subtree" will be interchangeably.

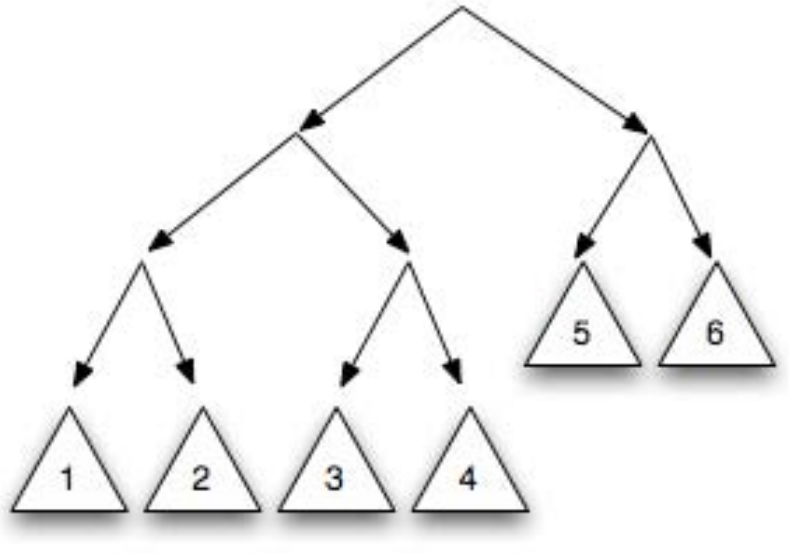

Figure 1. A covering set of six initial paths in a binary state-space tree

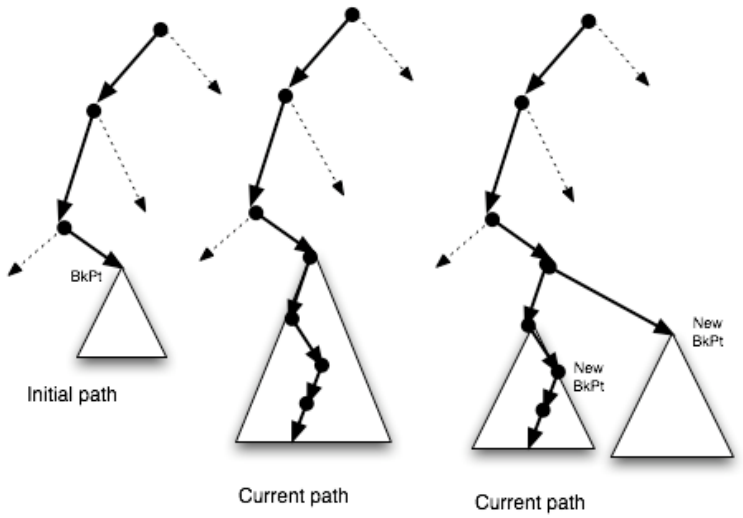

Figure 2. Splitting node 2's subtree

2.8 Speedup

When comparing the performance of parallel versus sequential backtracking search, two extreme situations arise. One extreme is when it takes a long time for the sequential algorithm to find a solution but the parallel algorithm finds it very quickly. This extreme situation arises when the solution is in a portion of the *state space tree* toward the end of a sequential backtracking search, and the parallel algorithm assigns this portion to a node that finds this solution quickly. Such a situation results in super-linear speedup. On the other hand, a solution might be located in a part of the state space tree searched early on by the sequential backtracking search, resulting in a situation where the sequential and parallel algorithm take the same amount of time (in fact, the parallel algorithm might even take more time because of the need to communicate the fact that a solution has been found). In the latter situation, I don't achieve any speedup at all, and at the cost of using *n* processors. Clearly, these two extremes are measuring the speedup of somewhat anomalous inputs, and the more important measure is the speedup on average. It is reasonable to expect that my backtracking framework will achieve the average linear speedup. Moreover,

if attention is restricted to those inputs of a given problem having no solution, such as searching for a satisfying assignment to the variables occurring in an unsatisfiable Boolean expression, it seems that linear speedup or even super-linear speedup will be achieved (on average) by my backtracking framework.

2.9 The Class NP / NP-complete

There is a large class of important decision problems, known as NP-complete problems, for which polynomial complexity algorithms solving any input instance for a given problem have not been found, nor has it been shown that any algorithm solving all inputs for the problem will necessarily require a super-polynomial time in the worst case. Moreover, if a polynomial solution can be found (for all input instances) for any NP-complete problem, then all NP-complete problems can be solved in polynomial time. Recall that the class NP is the set of decision problems for which a potential solution (called a "witness") to yes instances can be guessed in polynomial time, and the validity of the solution can be verified in polynomial time. The class P is the class of problems solvable in polynomial time, so that P $\subseteq$ NP. The question of whether P = NP (generally believe to be false) has remained open since it was posed over thirty years ago.

A problem X in NP is called *NP-complete* if every other problem Y in NP can be *polynomially reduced* to X, meaning that there is a mapping τ from the inputs of size n to Y to the inputs of at most size $p(n)$ to X, for a suitable polynomial $p(n)$, such τ is computable in polynomial time, and such that an input I to Y is a yes instance, if, and only if, $\tau(I)$ is a yes instance to X. It is far from obvious that NP-complete problems exist. However, in 1971, Cook proved that SAT is NP-complete. SAT refers to the problem of determining whether or not there is an assignment of values to the variables occurring in a given Boolean expression that makes the expression true (called a *satisfying assignment*). Recall that a Boolean expression is an expression among Boolean variables using the connectives "and", "or" and "not" (and

possibly others such as “xor”, “implies”, and so forth). Since Cook’s result appeared, literally thousands of problems have been shown to be NP-complete.

Since problems in NP are decision problems, they are suitable for the *Backtracking Framework*. As mentioned above, for NP-complete problems, there is no known algorithm that would solve the problem in polynomial time, and backtracking is often the only known algorithmic approach to the problem. As part of this project, a library of NP-complete problems will be created that are suitable for this framework, and a template will also be created that will help potentially implement any such problem. The problems currently implemented and tested so far are sum of subsets, knapsack, hitting set and SAT. A major goal of this project is to significantly enhance my implementation of a solver for SAT.

2.10 CNF

The most common SAT input is in the CNF (Conjunction Normal Form). Each CNF instance contains n variables. Each variable can have either a positive or negative literal (variable *x* – literal +*x* and -*x*). The variable can be either true or false. The literals form clauses that contain boolean ‘or’ operator only (e.g, (x_1 v -x_2 v x_3)). Clauses are then joined with boolean ‘and’ operator forcing each clause to be satisfied in order for the whole instance to be satisfied.

Example:

For variable x_1, x_2 and x_3 could be literals $x_1, \overline{x_1}, x_2, \overline{x_2}, x_3, \overline{x_3}$ and set of clauses

$(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2)$

Where one possible satisfying assignment could be

x_1 = *true*

x_2 = *true*

x_3 = *false*

CNF is a very common format for SAT in computing, and it is the one we use here. The standard DIMACS defines the actual file format.

2.11 SAT Solvers

There are two categories of SAT solvers, those which use a systematic search algorithm, and those which use a stochastic local search (also called incomplete). Solvers in the first category will give the solution if there is one and will give the answer 'unsatisfiable' if there is no solution. The stochastic local search algorithm will give a solution if there is one and will not finish if there is no solution. In this dissertation the discussion is limited to solvers in the first category that are based on Davis-Putnam-Loveland-Logemann backtrack search.

2.12 Van der Waerden numbers

In the 1920s, B. L. Van der Waerden proved the existence of a combinatorial property corresponding to partitioning (coloring) sufficiently large initial intervals of integers {1,2, …, n} [17]. Specifically, given positive integers K, L, Van der Waerden proved that there exists a (smallest) integer n = W(K,L) such that whenever {1,2, …, n} is K-colored, there exists a monochromatic arithmetic progression of length L in {1, 2, …, n}. Ever since this result was announced it has been a problem of interest to determine the exact value of these numbers, or at least improve lower bound estimates (Van der Waerden's existence proof gives truly enormous upper bounds). This problem has proved extremely difficult, and only a handful of these numbers are known. For example, for K = 2, the only known numbers are W(2,3) = 3, W(2,3)=9, W(2,4)=35 and W(2,5)=178. Using a specialized version of my SAT solver, I have determined a number of improved lower bounds for some Van der Waerden numbers. In particular, I have shown that $W(2,6) \geq 1132$, and I hope to soon finish a calculation showing that equality obtains.

2.13 FPGAs

FPGAs (Field Programmable Gate Arrays) or reconfigurable chips are integrated circuits that can be dynamically programmed to perform various functions. FPGAs are a relatively new technology that is a very active area of research today. The first FPGAs appeared in the mid 80s having around 1000 gates. The latest FPGAs have over tens of millions of gates. The basic building block in a typical FPGA is the LUT (LookUp Table). A typical FPGA has LUTs consisting of 4 inputs and 1 output, although some of the latest chips have started to use LUTs with more inputs. Accompanying the LUTs is a customizable interconnect that allows connecting virtually any LUT output to any LUT input. In addition to LUTs and the interconnect, FPGAs have external input/outputs and possibly additional logic such as RAMs, DSPs, registers, flip-flops, latches, adders, and so forth. By programming and combining logical components one can have a FPGA behave like a CPU or several CPUs, a specialized MPEG compression circuit, or even a SAT solver as I show in this dissertation.

Chapter 3 - Backtracking Framework

3.1 Acknowledgment

Parts of this chapter were taken from the following paper(s):

M. Kouril and J. L. Paul, *A parallel backtracking framework (BkFr) for single and multiple clusters, Proceedings of the first conference on computing frontiers on Computing frontiers,* ACM Press, Ischia, Italy, 2004, pp. 302--312.

J. L. Paul, M. Kouril and K. A. Berman, *A template library to facilitate teaching message passing parallel computing, SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* ACM Press, Houston, Texas, USA, 2006, pp. 464-468.

I was jointly responsible for writing the manuscript with Prof. Jerry Paul and I received substantial editing assistance from him. Prof. Jerry Paul and prof. Kenneth Berman applied the teaching framework in the classroom. The remainder of the work is mine. The co-author(s) acknowledge that majority of the work was done by me and grant their consent to use the material in this dissertation as it is.

3.2 Introduction

The backtracking paradigm as described in section 2.7 is particularly well suited to distributed and parallel processing, since the associated state-space tree modeling a given problem can be partitioned into subtrees that are assigned to different processes for independent processing. The resulting parallel backtracking strategy has been widely discussed in the literature. For example, Kumar et. al. [27] describe a general method where processes alternate between idle and active states in what they call *parallel depth-first search*. The flowchart in Figure 3 I illustrates how parallel depth-first search is implemented within my backtracking framework. In particular, when a process P_i finishes processing its assigned subtree and becomes idle, it then is available to help a working process P_j by a splitting of P_j's subtree into two subtrees that can then be worked on separately by P_i and P_j. In this way good load balancing can often be achieved. The BkFr backtracking framework described in this dissertation follows this depth-first search paradigm, and implements it in a single or multiple Beowulf cluster environment. Extending the computation over multiple clusters with good scalability is one of the principal aims of this research.

Beowulf clusters are becoming common today since they bring super-computer performance at a fraction of the cost. The main parallel communication library for Beowulf clusters is MPI [28], which is widely available for intra-cluster communication. Although MPI-2 [29], Globus [30] and other libraries offer support for inter-cluster message passing, they lack dynamic fault-tolerant capabilities, and require a high level of system support. In view of these limitations, I decided to develop ICI (see Chapter 4), which offers MPI-like functionality with only very basic system requirements such as Unix socket API. ICI provides a communication framework in which multiple clusters can be connected in a peer-to-peer scenario, thereby enabling clusters to dynamically join and leave complex computations. ICI also incorporates fault-tolerant features, including recovering from communication failures by transparently attempting to reconnect the resources. When reconnection is not possible, ICI informs my framework so that work can be reassigned to other resources. ICI allows my backtracking framework BkFr to be

implemented over multiple clusters, and uses functions that closely resemble a (small) subset of MPI, thereby facilitating programming by the user.

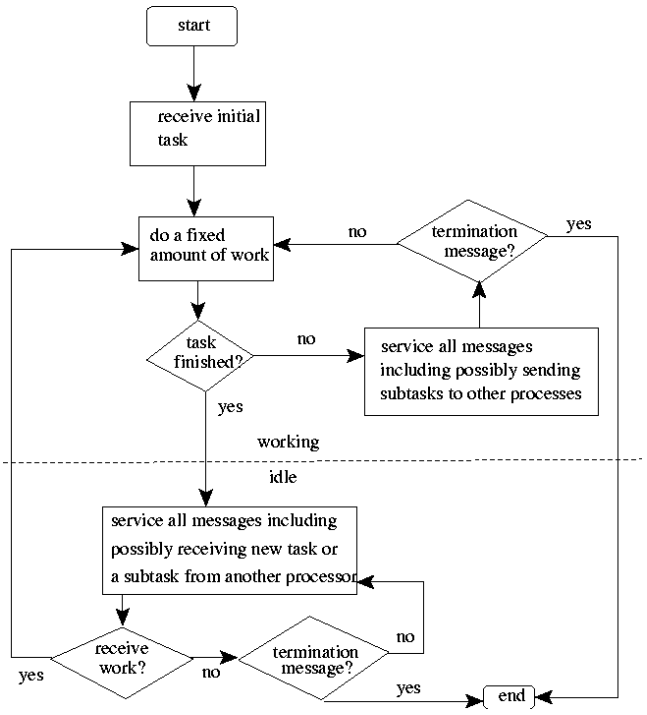

Figure 3. Implementation of parallel depth-first search within my backtracking framework

Other backtracking frameworks similar to BkFr have been built or described theoretically. For example, the ZRAM[31, 32] framework also uses a subset of the MPI functions for communication within a cluster, and additional features such as check-pointing and dynamic load balancing are implemented. The framework has several search engines: branch-and-bound, reverse search, backtracking and tree size estimation. However, the ZRAM framework is limited to a single cluster, and, in particular, does not provide for multiple clusters to dynamically enter and (possibly) leave the computation.

The paper by Sander[33] describes a model where a class of applications is modeled as a *tree shaped computation*. Various splitting strategies and several other strategies that help with efficient

parallelization are discussed. The author mentions random polling as a simple yet efficient method for work distribution. Karp & Zhang[34] show that random polling and randomized methods, in general, yield good results in a distributed environment. Here again, these implementations of the backtracking paradigm do not address support for multi-cluster computation and fault tolerance.

BkFr was created to provide a scalable platform for problems in the backtracking domain. The framework scales well from a small cluster or even a single computer computation to a multi-cluster environment where clusters can dynamically join and leave during the course of the computation. The intra-cluster communication provided by BkFr uses a small set of MPI functions, so other communication libraries could easily be used instead. Also, BkFr simplifies the API interface with compute modules facilitating effective utilization of any number of processors. The framework also imposes no restrictions on the work-splitting strategies or on the heuristics used by a particular problem implementation. Its generality makes it a useful tool for implementing the backtracking strategy in a variety of settings, from single computers to multiple clusters. Moreover, BkFr offers a simple interface (less than 10 functions) for new compute modules, making it relatively simple for the user to adapt existing code to the BkFr framework.

3.3 BkFr Overview

A general *Backtracking Framework* consists of three parts: (i) the communication part that handles (intra-cluster) communication between the nodes of the cluster and eventually (inter-cluster) communication between more than one cluster, (ii) the core part that creates an interface between the communication and the computation library in order to effectively handle the transparency of the platform for the problem implementation, and (iii) the problem implementation part that addresses the problem that is being solved.

When implementing BkFr in a cluster environment, nodes in a cluster are designated as either workers or the master. In the multi-cluster environment, one node in the initiating cluster is designated as the supermaster. BkFr has five main components:

- Communication
- Shared Variable Emulator and events
- Worker, master and supermaster framework functions
- Path Repository for multi-cluster computation
- Compute modules API

The relationship among these components is shown in Figure 4. When implementing a new problem in the BkFr framework, the user only needs to write code for the compute module. This code amounts to implementing the set of functions required by the compute module API as described in Section 3.12. As shown in Figure 4, when implementing the compute module API, the user utilizes the Shared Variable Emulation Layer, thereby obviating the necessity to explicitly include any MPI or ICI code in the compute module. In Section 3.12 I illustrate part of an actual compute module for the sum of subsets problem.

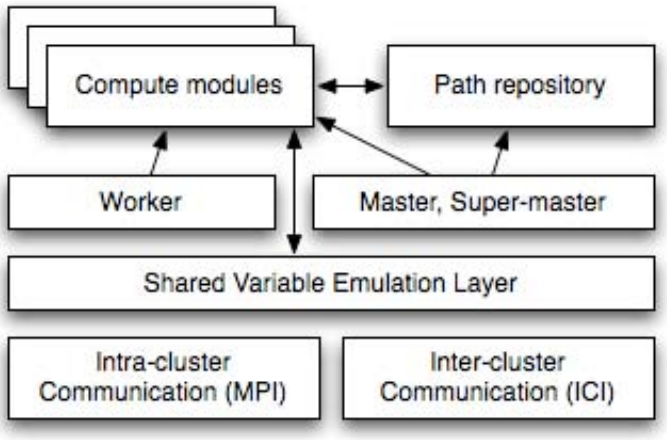

Figure 4. BkFr components

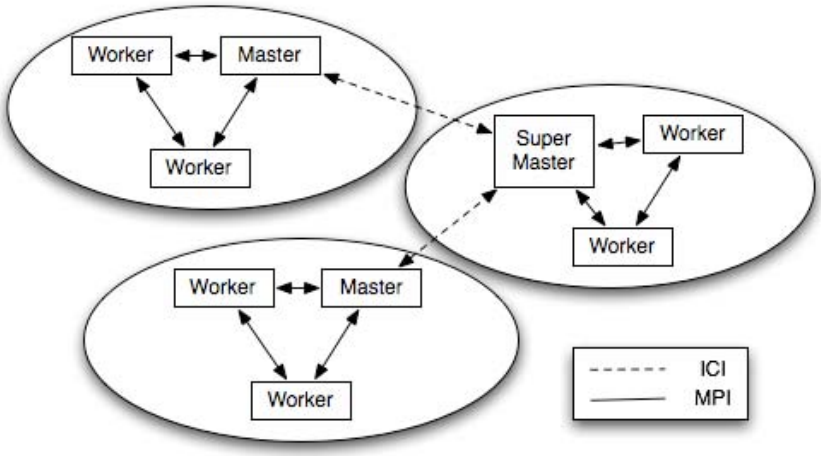

Figure 5. Communication options

3.4 Inter-Cluster and Intra-Cluster Communication Libraries

For intra-cluster communication, BkFr utilizes MPI, since it is available on a wide variety of clusters. However, BkFr uses only a small subset of the MPI library, so that it would not be difficult to develop alternate versions that use a different interface for intra-cluster communication. To better focus on framework functionality and to hide certain communication details, I developed a Shared Variable Emulation Layer (discussed further in the next section) that makes use of MPI Tags to identify a specific

variable or an event during send and receive operations. All MPI functions used by BkFr are non-blocking, such as MPI_Bsend, and MPI_Iprobe. MPI_Recv is executed only if the existence of a message is confirmed via MPI_Iprobe.

For applications where the preallocated buffer required by non-blocking communication presents a difficulty, future implementations of BkFr will provide the opportunity to utilize alternate communication functions such as MPI_Isend. Asynchronous broadcasting is important to the BkFr framework, and while not supported directly by MPI, is easy to implement with existing MPI commands and explicit broadcast trees.

For reasons mentioned at the beginning of this chapter, I decided to implement my own Inter-Cluster Interface library (ICI) for inter-cluster communication. ICI provides an MPI-like interface with functions corresponding to a subset of the MPI functions. Again I use non-blocking functions such as ICI_Bsend, ICI_Iprobe. In addition, functions for managing connections between clusters are required, such as ICI_Listen and ICI_Connect. The ICI environment allows for clusters to enter and possibly leave a computation dynamically (for example, due to a network link failure), without affecting the integrity of the computation. The initiating cluster for a computation contains a node that BkFr designates as a supermaster, and all inter-cluster communication goes through the supermaster. A node in each of the participating clusters is designated as the master node, which manages computations within the cluster, and which is the only node in the cluster that communicates with the supermaster. All other nodes are designated as workers (see Figure 5). If the cluster containing the supermaster node encounters a node failure, then (as usual in the MPI environment) the whole computation fails. However, this is not true of any of the other participating clusters, since the supermaster maintains a Path Repository, and will simply reassign the work of a failed cluster to another resource. This fault-tolerance feature was a principal motivation for the development of ICI.

The standard MPI process numbering scheme within each cluster is used to accommodate intra-cluster and inter-cluster communication. More precisely, the nodes inside the cluster have node number

assigned as they would using standard MPI_Comm_rank(MPI_COMM_WORLD). Node number 0 in any cluster is regarded as the master node for the cluster, and is the only node in the cluster directly participating in inter-cluster communication. Inter-cluster communication uses the negation of a connection number obtained from the ICI library. Hence, if the framework sends a message with a positive destination identifier, then the request is targeted inside the cluster, otherwise inter-cluster communication occurs.

3.5 The Shared Variable emulator and event functions

As mentioned above, in order to implement the higher-level framework functionality I decided to create an intermediate layer, called the Shared Variable Emulator, which hides the communication specific functions like MPI or ICI function calls. Not only did this allow us to better focus on the desired framework functionality, but it also greatly simplifies the implementation details required by users of BkFr. Variables are identified by a value of 0 to VAR_USER-1 for system variables and VAR_USER and up for user variables. Every shared variable has to be first registered with the Shared Variable Emulator. During the registration the user has to specify the identifier, text representation, variable type, and number of members (1 – single value, 0 – for auto-allocation during message receive and > 1 for a pre-allocated array). The user also has to specify the physical location of the variable identified by a pointer. The pointer is NULL for auto-allocated variables. A special type of temporary variable is used to send a value without retaining a local copy. The variable space is de-allocated as soon as the temporary variable is sent. Every variable has a set of rules attached to it that describe what to do with the value should the variable be updated. The special functions for updating the variables are, of course, available as well. This layer is not only available for the framework and its functionality, but also for the compute modules themselves. All of these features make it easier to implement new problems within the framework.

My Shared Variable Emulator recognizes three types of elements: variables, variable events and system events (see Table 1). One variable can have more than one event associated with it, so that the

variable can be sent with event-dependent purposes. Variables can contain a condition validation function that will update variable only if the condition is true (for example, if the new value is higher that the previous value). The set of rules directing the distribution of the variables is sensitive to a sender-receiver hierarchy (supermaster, master, worker). For example, it is possible to set the rules in such a way that a variable is automatically broadcast throughout the cluster when it is updated by the master.

Typically a function is called upon receiving an update to a variable/event. System events are also attached to variables and allow the framework to perform appropriate system actions when a node receives a variable. At the same time, a compute module attaches its own events to variables as appropriate to its problem-specific functionality (for example, see the PATH variable description in the section 8).

Table 1. Shared Variable Emulator elements

Type	**Description**
Variable	Shared variable identification, by default no function is attached.
VariableEvent	Function assigned to an earlier defined variable
SystemEvent	Function defined by the system and later assigned to a variable

Updating variables can trigger different actions depending on whether the variable is updated locally or from another node or cluster. Moreover, while the user can use `SendVariable` or `SendVariableAs` functions to send a variable to a specific node, these functions should be restricted to use by the framework in order to avoid portability problems. Some of the functions available to the user are shown in Figure 6.

To illustrate how the API works I now describe two of the functions – `RegisterVariable` and `LocalUpdateVariableInt`.

Every variable in the Shared Variable Emulator has to be registered, a task that is performed by the `RegisterVariable` function. The `RegisterVariable` function initializes internal structures for the variable and sets up the initial sharing rules. The format and parameters are listed below:

```
void RegisterVariable(char *name, int id, int num_elements, int datatype, void *ptr)
```

- `char *name` – variable name – string used to read the variable from a data file or to display the variable together with its value
- `int id` – variable id – integer with a value of greater or equal to VAR_USER. This is the identifier of the variable.
- `int num_elements` – 0 or positive integer. Identifies the number of elements in the variable. If the variable is not an array num_elements equals 1. If num_elements equals 0 the variable is auto-allocated whenever necessary.
- `int datatype` – BKFR_INT or BKFR_CHAR. Additional types will be added in future implementations.
- `void *ptr` – pointer pointing to the actual location of the variable. For auto-allocated variables this value is NULL.

```
void RegisterVariable(char *name, int id, int num_elements,
     int datatype, void *ptr);
void RegisterTempVariable(int *id, int num_elements,
     int datatype, int duplicate_id);
int SendVariable(pcompute_struct compute, int var_id, int dst);
int SendVariableAs(pcompute_struct compute, int var_id,
     int dst, int evt_id);
void FreeVariable(int var_id);
void ReallocateVariable(int i, int num_elements);
void LocalUpdateVariable(int var_id, void *ptr, int len);
void LocalUpdatedVariable(int var_id);
void LocalUpdateVariableInt(int var_id, int val, int pos);
void AddVariableFn(int var_id, proc_var_event proc, int bcast);
void AddVariableDst(int var_id, int dst, int dst_send_as);
void AddVariableCond(int var_id, proc_vars_cmp fn);
void AddVariableEvent(int var_id, int evt_id,
     proc_var_event proc);
void AddSystemEvent(int evt_id, proc_var_event proc, int bcast);
int GetVariableCount(int id);
int SetVariableCount(int id, int count);
void *GetVariablePtr(int id);
void SetVariablePtr(int id, void *ptr);
int GetElementSize(int id);
int GetVariableCount(int id);
int GetVariableSize(int id);
```

Figure 6. Selected Shared Variable Emulator API Functions

For example, one might invoke RegisterVariable(“SUM”, VAR_SUM, 1, BKFR_INT, &var_sum).

`LocalUpdateVariableInt` is function for updating the shared variable by the user program. Using this function instead of a direct update to the variable location ensures that any rules connected with the variable will be followed. Such rules could be communicating the variable value to all nodes (broadcast), conditional update, and so forth. The first argument is the variable id, the second is the new value and the third is the position in the array if the variable is defined as an array. For example, one might invoke `LocalUpdateVariableInt(VAR_SUM, 100, 0)`.

3.6 The Master-Worker Intra-Cluster Paradigm

Within a single cluster, the framework designates a master process (typically node 0), and all other processes are workers. If the MPI (intra-cluster communication environment) is not detected, or only one process is requested, the framework defaults to the so called *stand-alone* mode where the computation module is loaded on a single computer without any external connection. The main functions of a worker process are to accept work from the master, to eventually report back to the master when the work is finished, and to respond to the master's requests for splitting the work of other workers.

The master's responsibilities include managing the computation, and maintaining the worker status vector, where the entry for each worker is one of the values IDLE, WORKING, STUCK, SENT, REJECTING or DEAD. These values have the following interpretations:

- IDLE – process has not been sent the problem definition yet
- WORKING – process is working on its subtree
- STUCK – process has finished its part of the subtree and informed the master
- REJECTING – process is working has refused to split its subtree
- SENT – after a worker has reported being STUCK, the master has requested another (working) worker to accept help from this stuck worker. The status SENT is assigned to the STUCK worker so the master knows that it already dealt with this STUCK worker.
- DEAD – the worker is in the process of shutting down. The master will not try to further contact the worker.

From the communication standpoint, the framework uses the Shared Variable Emulator mentioned earlier, which hides the actual communication calls behind a list of rules attached to each variable. Table 2 lists the variables and their properties.

The main transition phases of a single-cluster computation are as follows:

Initialization.

1. The master starts, and waits for a status report from each worker.
2. Each worker reports status IDLE as soon as it starts.
3. When the master has received the idle report from each worker, it sends out the problem definition and the initial path for each worker.
4. After receiving the problem definition and its initial path, each worker begins executing the alternating computation/communication cycles.

A worker finishes searching its part of the state-space tree.

1. The worker informs the master that it is available as a helper.
2. The master will choose a working worker and sends message containing the id of the available helper whose status is now SENT.
3. The worker receives the id of the helper and makes an attempt to split its work.
4. If successful the split path is sent to the helper that then restarts its computation/communication cycle.
5. If unsuccessful the master is so informed and it then attempts to send the helper id to another working worker.
6. If no working workers are available to accept help, the helper's status is (re)set to STUCK.

Shutdown.

1. The master receives messages from all workers that they have finished searching their subtrees.
2. The master updates a variable and initiates a broadcast of this variable to every worker with the request to shutdown.

3. Upon receiving this update, workers forward the update to their children in the broadcast tree, and shutdown. The last worker (highest id) in the cluster sends a message back to the master for a more graceful shutdown.
4. The master also shuts down and the job terminates.

3.7 The SuperMaster-Master-Worker Inter-Cluster Paradigm

During a multi-cluster computation, some type of central control must manage the splitting and the distribution of the problem onto multiple clusters. Such functionality is a natural extension to the master role, and is given to a node that I call the supermaster. In addition to its responsibilities for managing the work of other clusters, the supermaster also performs the usual tasks of a master node with respect to its own cluster. The supermaster uses a subsystem called the Path Repository (see Section 3.11) where it maintains the status of every possible path in the state-space tree. The supermaster also maintains a clusters status vector, which contains a status of every cluster currently participating in the computation. Similar to the single cluster worker status vector, entries in the cluster status vector have one of the values IDLE, WORKING, STUCK, REJECTING or SENT. The interpretations of each of these values are as follows:

- IDLE – cluster has not been sent the problem definition yet
- WORKING – cluster is working on its subtree
- STUCK – cluster has finished its part of the subtree and informed the master
- REJECTING – cluster is working but refusing to split its subtree
- SENT – after a cluster reported being STUCK, the supermaster has requested another working cluster to accept help from this stuck cluster. The status SENT is assigned to the STUCK cluster so the supermaster knows that he already dealt with this STUCK cluster.

Initially, the supermaster reads a problem definition and initializes its own cluster. When a cluster joins the computation, the supermaster sends the joining cluster the problem definition and an initial path. If there are no more pre-split initial paths to be sent, the supermaster asks another cluster to split its path and send it back. The supermaster then records the split path in the Path Repository and forwards it to the idle cluster. The similar situation happens when any cluster finishes its assigned subtree. A cluster has an option to refuse to split its path, in which case it so informs the supermaster. The supermaster, as with a single cluster helper scheme, makes a note of this (changes the status of the cluster in the clusters status vector to REJECTING) and asks another working cluster to accept the split path. The computation ends when all clusters remain in the STUCK status. At this point the supermaster sends a request to every connected cluster to shut down.

3.8 BKFR Shared Variables

In this section I describe the shared variables that control the framework (see Table 2). The PATH variable plays a special role in the system. The compute module defines the PATH variable instead of the framework. Since every compute module expresses the PATH in a different way, this variable cannot be predefined by the framework. My solution is to let compute module define the PATH variable and attach framework events to it. Then the framework uses events rather than actual variable to perform such tasks as initializing a new worker, sending a helper a new path, and so forth.

In Table 2 only communication properties for variables that require them are listed. When the communication properties are left blank it means that no further communication actions are taken. The broadcast communication is always non-blocking and initiated by the master.

3.9 Single cluster shared variables

BkFr maintains the variable VAR_STATUS in the worker portion of the code, and uses the variable to send updates to the master whenever the status of the worker changes. Some examples of the status of a worker and the concomitant value taken by VAR_STATUS include:

- the worker was just initialized and is ready to accept the problem definition and the initial PATH (STATUS_IDLE)
- the worker has started a computation (STATUS_WORKING)
- the worker has finished the work associated with it's subtree (STATUS_STUCK)
- the worker has shutdown (STATUS_DEAD)

Table 2. BkFr Shared Variables

(* identify variable defined and assigned to the function by the compute module)

Worker:

Variable / direction	**Event**	**Communication properties**	**Attached Function**
VAR_STATUS / in		Broadcast	No function attached
VAR_STATUS / out		To master	--
VAR_HELPER / in	EMPLOY_HELPER		BkFr_EmployHelper
VAR_HELPER / out	REFUSE_HELPER	To master	--
* / in	INIT_PATH	Broadcast	InitPath
* / in	INIT_HELPER_PATH		InitHelperPath
-- / out	INIT_HELPER_PATH	To helper (worker/master)	--

Master:

Variable / direction	**Event**	**Comm. props**	**Attached Function**
VAR_STATUS / in			BkFr_WorkerStatusUpdate
VAR_STATUS / out		Broadcast	--
VAR_HELPER / in	REJECT_HELPER		BkFr_RejectHelper
VAR_HELPER / out	EMPLOY_HELPER		--
VAR_CLUSTER_STATUS / in			No function attached
VAR_CLUSTER_STATUS / out			--
* / in	INIT_PATH	Broadcast	No function attached
* / in	INIT_HELPER_PATH		BkFr_ClusterHelperPath
-- / out	INIT_HELPER_PATH		--

Multi-cluster (supermaster and non supermaster):

Variable / direction	**Event**	**Comm. Props**	**Attached Function**
NEW_CLUSTER_STATUS / in			BkFr_NewClusterStatus
* / in	INIT_CLUSTER_PATH		InitClusterPath

Supermaster:

Variable / direction	Event	Comm. Props	Attached Function
VAR_CLUSTERS_STATUS / in			No function attached
VAR_CLUSTER_STATUS / in			BkFr_ClusterStatusUpdate
-- / in	CLUSTER_HELPER_PATH		No function attached

Non supermaster:

Variable / direction	Event	Comm. Properties	Attached Function
VAR_CLUSTERS_STATUS / out		DST=uplink	--
VAR_HELPER / in	CLUSTER_SPLIT_TREE		BkFr_ClusterSplitTree

[1]variable defined and assigned to the function by the compute module

When the master determines that the entire computation has completed, it updates every worker's VAR_STATUS by initiating a broadcast of the value STATUS_DIE for this variable to all the processes.

The shared variable VAR_HELPER contains the id of a worker that is available to help other workers with their subtrees. The master uses the event called EMPLOY_HELPER with the attached variable VAR_HELPER to ask a working worker to split its subtree. If the worker accepts help, it splits its subtree and sends a portion of it to the worker identified by the VAR_HELPER variable. If the worker refuses to split its tree, it replies with an event REFUSE_HELPER, and an attached variable VAR_HELPER containing the previously received id contained in EMPLOY_HELPER.

The PATH variable is not directly registered by the framework, but relies instead on the compute module to register it. The framework registers events that the compute module is expected to attach to the PATH variable via the events INIT_PATH and INIT_HELPER_PATH. The event INIT_PATH is used to initialize the worker, which automatically creates its assigned path of the covering set based on its own id. The event INIT_HELPER_PATH initializes the worker's path without adding the covering set.

3.10 Multi-cluster extension of shared variables

In a multi-cluster computation, the status of the clusters is kept by the supermaster in the variable VAR_CLUSTERS_STATUS. The value of VAR_CLUSTERS_STATUS is initialized as STATUS_WORKING and is changed to STATUS_DEAD if all connected clusters are shutdown. The variable NEW_CLUSTER_STATUS is used by supermaster to inform a connected cluster to shutdown in the end of the computation.

In the multi-cluster setting, there are two additional events attached to the PATH variable registered by the compute module. These events are INIT_CLUSTER_PATH for initializing PATH for the whole cluster by the supermaster, and CLUSTER_HELPER_PATH for sending PATH from one cluster to the supermaster after the supermaster has requested to split the subtree associated with PATH.

3.11 Path Repository

The Path Repository is the critical component of multi-cluster computations, since it maintains the status of every possible path in the covering set. In the current version there is only one Path Repository, and it is maintained by the supermaster. The Path Repository allows the supermaster to find a path that is not assigned to anyone, or, for fault tolerant purposes, to find that portion of the state-space tree that was previously assigned to a failed cluster so that it can be assigned to another resource.

When a cluster gets stuck the supermaster has two options for assigning additional work to the cluster. First, if the Path Repository contains unsearched paths, the next available unsearched path is sent to the stuck cluster. If no unsearched paths exist, then the supermaster asks another cluster to split its subtree into two parts, one of which will be sent to the stuck cluster via the worker (helper) → master → supermaster → master (stuck cluster) communication chain.

3.12 Compute modules API

In order to create a flexible environment in which new problems can be very easily added to the library by the user, I have designed BkFr so that the modules are required to implement only a small set of functions (see below). The compute modules are loaded in the form of a shared library, so that there is no need to recompile the framework itself when adding a new module.

The functions are split into three groups. The first group (basic computation) is mandatory the other two are required only if additional functionality is required. For example, if no multi-cluster computation is expected, then the user does not have to implement Path Repository or any other functions connected with the multi-cluster computation.

Basic computation

- `proc_init_compute` – initializes the compute module.
- `proc_search_tree` – hands the control over to the alternating compute/communication cycle. It is expected that the compute module will return control after reasonable amount of time so the framework could check for messages from other processes.
- `proc_get_path_var` – returns the id of the variable in which the module stores the PATH variable.
- The compute module must register the path-description variable and attach it to the event INIT_PATH using the function `AddVariableEvent`. The third parameter of `AddVariableEvent` is a name of a function that will be called upon receiving the INIT_PATH event.

Helpers

- `proc_employ_helper` – this function is called from the framework to request the compute module to split its current subtree, update PATH, and hand over a portion of the split subtree to another (idle) worker. The new path is expected to be in another registered variable.
- The compute module must attach a path-description variable to the INIT_HELPER_PATH event using the function `AddVariableEvent`. The third parameter of `AddVariableEvent` is a name of a function that will be called upon receiving the INIT_HELPER_PATH event.

Multi-cluster computation with Path Repository

- `proc_get_init_path` – the compute module is expected to return an initial path, that is, a path representing the entire subtree to be searched.
- `proc_split_path` – the compute module will split the subtree associated with the supplied path into two subtrees.
- `proc_compare_path` – by comparing two paths the compute module will return one of 5 possible answers – 1) the paths are equal, 2) path 1 is subsumed by path 2, 3) path 2 is subsumed by path 1, 4) path 1 is further in the tree than path 2 and 5) path 2 is further in the tree than path 1.
- The compute muddle must attach path-description variable to the INIT_CLUSTER_PATH event using the `AddVariableEvent` function[3]. The third parameter of `AddVariableEvent` is a name of a function that will be called upon receiving the INIT_CLUSTER_PATH event. The framework itself initiates a broadcast of the PATH variable automatically to all nodes.
- The compute module must attach a path-description variable to the CLUSTERS_HELPER_PATH event using the `AddVariableEvent` function[3]. The third parameter of `AddVariableEvent` is a name of a function that will be called upon receiving the CLUSTERS_HELPER_PATH event.

[3] the compute module is not required to implement these functions.

3.13 Teaching component of BkFr

It is important that the CS undergraduate be exposed to the message passing parallel computing paradigm as supported by MPI. However, often the undergraduate curriculum is too tightly constrained to allow for a full course devoted to this topic. Also, since learning MPI imposes an additional requirement on students and teachers alike, the message passing parallel computing concept might be skipped due to its complexity and background requirements. With these things in mind I have created a template library based on BkFr to facilitate teaching message passing parallel computing modules that can be easily integrated into the standard upper-division undergraduate (or first year graduate) course in algorithms. My focus has been on developing MPI programming modules that can be generated using a general parallel backtracking framework that I have developed, but I have also generated modules outside of this framework. The template library not only simplifies the material for the students, but also eases the burden on the instructor without compromising the concepts themselves. This library will eventually be available on-line.

Most of my MPI programming templates involve the backtracking design strategy. There are several reasons for focusing on the backtracking (and, more generally, on parallel depth-first search) for my template library. First of all, backtracking is one of the major design strategies that are typically included in an algorithms course. Second, backtracking problems are well suited to parallelization using the fundamental master-worker parallel design strategy. Moreover, parallel backtracking illustrates nicely yet another fundamental concept, namely, the alternating computation/communication cycle paradigm. Third, there are important problems with many applications, such as the satisfiability (SAT) problem for Boolean expressions, for which backtracking is the most commonly used design strategy. Indeed, there is a very active research community devoted to designing ever more efficient SAT solvers, and a large suite of benchmark problem instances is available on which to test solutions. Thus, the student can be exposed and gain insight into leading-edge research on SAT

I have tested my template library in the two quarter upper-division course in algorithms that is required for all CS majors. The students were very enthusiastic about including this introduction to the message passing parallel programming environment as part of the algorithms course. In the first quarter, with the aid of a template the students developed a sequential SAT solver using various heuristics to bound the search, and tested their results against a given standard suite of benchmark problems. Then in the second quarter, using my template library they developed a parallel SAT solver and compared it to the results obtained with their sequential solvers. In a semester course, this scenario could still be followed by assigning the sequential SAT solver problem early (or another problem of the instructor's choice), and the parallel SAT solver problem toward the end of the term. In any case, with the aid of the template library, it is quite feasible to give the student a solid introduction to the message passing parallel paradigm, but at the same time not spending more time than typically devoted to one of the many usual topics included in the standard upper-division algorithms course.

In reference [35] there is a chapter devoted to message passing parallel algorithms, including high-level pseudocode for basic process communication functions such as synchronous and asynchronous send and receive instructions. An appendix in [35] includes actual MPI code for some of the algorithms given in pseudocode within the text. Based on in-class experience, the treatment in [35] is adequate for those wishing to introduce MPI programming as one topic amongst many in an algorithms course. For those giving an entire course on the subject, there are now some good books devoted to teaching these topics ([36],[37]).

Template-based learning approaches in other programming contexts have been used in the past with success. To give some examples, in [38] a dynamic analysis framework for analyzing "fill-in-the-gap" Java exercises is discussed. The concept of "example-based" (Pascal) programming is presented in [39], and in [40] a project involving template-based programming in chemical engineering is described.

To illustrate my template-based MPI programming library, I focus on one template module, namely SAT. However, I have also developed modules for a number of other backtracking problems (sum of subsets, n-Queens, and so forth), as well as numerical problems such as matrix multiplication.

Table 3. Description of the provided functions

Provided function	Description
`read_cnf(FILE *fin);`	Read in CNF file in DIMACS format
`init_variables();`	Initialize all variables to unknown
`print_solution();`	If a solution was found this function prints it in the standard format
`print_unsatisfiable();`	If no solution exists this function reports it in the standard format
`check_all_clauses();`	Check all the clauses given the current assignment Returns SAT if the current assignment satisfies the formula, or UNSAT if a contradiction was found or UNDECIDED if some unsatisfied clauses are left.

3.13.1 Sequential Version

As mentioned in the introduction, students are first assigned the problem of generating a sequential SAT solver. To accelerate the student's ability to create their own SAT solver, I generated a C program template for the sequential version that included the functions described in the Table 3. An example of the template code given to the students is given in Figure 7.

```
typedef enum {
   NOT_ASSIGNED,
   TRUE,
   FALSE
} t_variable;
```

```
typedef enum {
   UNDECIDED,
   SAT,
   UNSAT
} t_sat;

/* remember where each clause starts */
int clauses_array[MAX_CLAUSES];
/* all clauses delineated by zeros */
int all_literals[MAX_LITERALS];
/* variable assignment */
t_variable variables_array[MAX_VARIABLES+1];
/* number of variables, clauses */
int variables, clauses;

t_sat backtrack() { /* return SAT if the satisfiable assignment is found, UNSAT otherwise */
/* FIXME */
}

int main() {
    int err = read_cnf(stdin);
    /* check for error during read_cnf */
    if (err != 0) return (err);
    /* assign all variables NOT_ASSIGNED */
    init_variables();
    if (backtrack() == SAT) print_solution();
    else print_unsatisfiable();
    return 0;
}
```

Figure 7. Example of part of the template function in the sequential version

The student's main responsibility is to write the backtracking portion utilizing various heuristics. In order to aid a student with the development environment, the following "quick start" is given to the student (see Figure 8).

As indicated by Figure 8, the students will then complete their SAT solvers by fixing the template, compiling, executing, and checking their results using the verifier. Fixing the template involves writing a backtracking function, as well as implementing a heuristic to determine the branching order in the associated state space tree. The student is asked to implement three different versions based on the following orderings of the variables.

a) Order the variables by their indices. This is a static ordering, that does not depend on the particular problem instance.
b) Order the variables based on the number of clauses they appear in at the start of the program. This heuristic is problem dependent, but the ordering is fixed (static) throughout the computation.
c) Order the variables dynamically by choosing the next variable based on the number of remaining unsatisfied clauses each variable is in.

The reader is referred to Chapter 18 in [35] for further details.

Quick start:

1) log in to gatekeeper (or your favorite UNIX machine)

gatekeeper ~>

2) download the skeleton source code

gatekeeper ~> wget http://www.ececs.uc.edu/~mkouril/class_sat/testsat.c

3) download the verifier

gatekeeper ~> wget http://www.ececs.uc.edu/~mkouril/class_sat/verifier.c

4) download a few cnf examples

gatekeeper ~> wget http://www.ececs.uc.edu/~mkouril/class_sat/uf20/uf20-01.cnf

gatekeeper ~> wget http://www.ececs.uc.edu/~mkouril/class_sat/uf20/uf20-02.cnf

5) compile the skeleton

gatekeeper ~> gcc testsat.c -o testsat -Wall -g

6) compile the verifier

gatekeeper ~> gcc verifier.c -o verifier

7) test the skeleton

gatekeeper ~> ./testsat < uf20-01.cnf

s SATISFIABLE

v 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 0

8) test the skeleton with the verifier

gatekeeper ~> ./testsat < uf20-01.cnf | ./verifier uf20-01.cnf

c~~ verifier: ERROR some clauses left (v lines don't define an implicant)

c~~ verifier: ERROR: Original clause left: c~~ verifier: -5 -8 -15 0

As you can see the skeleton returns the wrong solution due to the missing backtracking function. Please come up with your own backtracking function.

Once your backtracking function works correctly the output from the verifier will change to

c~~ verifier: OK: SATISFIABLE claimed, and implicant is correct

Figure 8. Quick start given to the students

3.13.2 Parallel Version

In the parallel version I utilize the master-worker paradigm, where the master process hands out tasks (initial paths in the state space tree) to the workers. The master (node with MPI rank=0)

precomputes a fixed depth covering set of initial paths, (see Figure 9), and hands out the next unused path to any worker node currently idle. Typically the covering set has more leaf nodes than the number of nodes involved in the parallel computation.

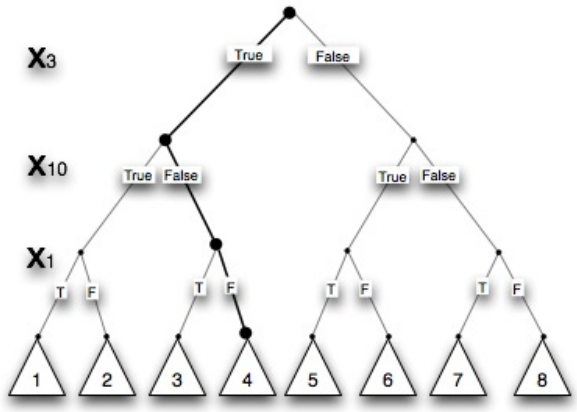

Figure 9. Typical covering set of fixed depth 3 for a suitable heuristic. An initial path $x_3, \overline{x}_{10}, \overline{x}_1$ is indicated.

A worker (node with non-zero MPI rank) upon receiving an initial path from the master starts the backtracking function that alternates between computation (backtracking) and communication. The communication portion is simply checking for a message sent by master indicating that someone already found a solution and the worker can shutdown. Upon receiving such a "die" message, the worker then propagates this message to its children in a user-defined broadcast tree so that eventually the message is received by all processes. This user-defined asynchronous broadcast is used instead of the MPI provided synchronous broadcast that locks all nodes. The MPI synchronous broadcast function cannot be used in this context, since the state (idle or working) of individual workers is not known in advance. The master initiates the user-defined asynchronous broadcast of a "die" message after exhausting the covering set and receiving an "I'm done" message from everyone, or upon receiving an "I found a solution" from a worker. In order to focus on parallelization issues, the template for the parallel version includes code for

the heuristic that orders variables before each decision based on the product of the number of remaining clauses the variable is in positively and negatively. However, the code for this heuristic is straightforward, and was actually part of the student's responsibility in the sequential version. Figures 4 and 5 show portions of the code from the parallel version of the template.

The template provided to the students also includes a "stand alone" configuration that allows the student to debug the basic backtracking portion of their code in a purely sequential setting (i.e., independent of any MPI code). This stand alone feature allows them to verify their programming logic before introducing the parallelism provided by MPI. When proceeding to the parallel version, the template is missing MPI calls that they are required to fill in. The places where MPI is missing are clearly marked with a comment (FIXME), and even the number of recommended MPI function calls is provided (e.g., `/* FIXME MPI, 2 */` means that two MPI statements are required). Similar to the quick start given in Figure 2 for the sequential version, I provide the student with a quick start for the parallel version that reflects the implementation details of my current MPI environment.

```
. . .
while(running==1) {
  if (idle == 0) {
     /* worker received task to work on */
     t_sat result=backtrack(myid, p);
     idle=1;
     switch (result) {
     case SAT: {
       /* SAT solution was found */
       /* get solution from the SAT solver */
       /* FIXME MPI, 0 */

       /* send it to the master */
       /* FIXME MPI, 1 */

       /* send idle message to the master */
       /* FIXME MPI, 1 */
     } break;
. . .
```

Figure 10. A portion of the worker code template with missing parallel code

Remarks: The template I generated for extending the sequential SAT version to the parallel version actually used a simplified version of a general backtracking framework (BkFr). In particular, BkFr includes the possibility for idle workers to request more work by splitting the state space subtree assigned to another worker.

```
t_sat backtrack(int myid, int p) /* return SAT if the satisfiable assignment is found, UNSAT
otherwise */
{
 while(1) {
  /* communication cycle:
   * check to see whether it is possible to end the computation
   * prematurely – i.e., whether an MPI message with
   * MPI_TAG = DIE has been received */
#ifndef STANDALONE  //run parallel code
  int flag=0;
  MPI_Status status;
  MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
  if (flag == 1) {
     if (status.MPI_TAG != DIE) {
       fprintf(stderr, "Worker(%d): Unexpected message\n", myid);
       exit(1);
     }
     MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

     /* tree like continuation of
        broadcast of DIE message */
     /* FIXME MPI, 2 */

     /* if I'm the last processor
        send message back to the master */
     /* FIXME MPI, 1 */
     return UNDECIDED;
  }
#endif
 /* computation cycle: purely sequential code */
 /* FIXME MPI, 0 */
 }
}
```

Figure 11. Template version of the parallel backtracking function executed by workers

3.14 Evaluation

3.14.1 The Sum of Subsets Problem module

In order to test the functionality of BkFr, I first ran a straightforward implementation of the optimization version of the sum of subsets problem. Recall that in the sum of subsets problem a set of numbers *S* and a number *sum* are given, and a smallest subset *A* of *S* whose elements add up to *sum* is sought. The input data to the sum of subsets problem module is an array of numbers representing *S*, and the number *sum*. The output (PATH) corresponding to a solution *A* is a 0/1 array representing the characteristic vector of *A*.

The sum of subsets compute module reads the numbers from a file on the master/supermaster and distributes them to the workers/masters. The module registers variables called NUMBERS, SUM, PATH and bounding variable VAR_BOUND. Variable PATH could be updated by two events – initial path setup (in the beginning of the computation where the whole cluster is stuck) or by receiving path from another computer after previously finishing its subtree.

The bounding variable is expressed as the length of the best solution found so far. It is a controlled variable with a special function attached that checks whether the local copy is better that the copy received from the external source (worker, master) or updated by the local computation. If the value is indeed better, then the framework will initiate the sharing of this variable with the rest of the cluster.

The compute module registers the following variables:

- VAR_PATH – array of integers (0/1)
- VAR_NUMBERS – array of integers
- VAR_SUM - integer
- VAR_BOUND (with MIN_cmp_fn) - integer

Basic computation

- `proc_init_compute` – allocates necessary internal variables
- `proc_search_tree` – in steps searches sequentially through the tree keeping the initial path intact.
- `proc_get_path_var` – returns VAR_PATH
- INIT_PATH function – creates initial covering set for all nodes depending on the number of the nodes in the cluster. Each node will have path as the starting point to search tree function.

Helpers

- `proc_employ_helper` – splits the path effectively dividing the search space by two and extending the initial path by one. It returns the the newly created second half of the path
- INIT_HELPER_PATH function – Receives VAR_PATH and sets up the starting point for the compute cycle.

```
int module_init (pcompute_struct _compute, int *argc, char **argv[])
{
   compute->proc_init_compute =   sumset_InitCompute;
   compute->proc_search_tree =    sumset_SearchTree;
   compute->proc_get_path_var =   sumset_GetPathVar;
   compute->proc_employ_helper =  sumset_EmployHelper;

   /* Path Repository */
   compute->proc_get_init_path =  sumset_GetInitPath;
   compute->proc_split_path =     sumset_SplitPath;
   compute->proc_compare_path =   sumset_ComparePaths;

   compute->proc_register_variable(
                "PATH", VAR_PATH, 0, BKFR_INT, NULL);
   AddVariableEvent(VAR_PATH,
          INIT_PATH, sumset_InitialSetupTree);
   AddVariableEvent(VAR_PATH,
```

```
           INIT_HELPER_PATH, sumset_HelperSetupTree);
   AddVariableEvent(VAR_PATH,
           INIT_CLUSTER_PATH, sumset_HelperSetupTree);
   AddVariableEvent(VAR_PATH,
           CLUSTERS_HELPER_PATH, sumset_HelperSetupTree);


   compute->proc_register_variable(
       "BOUND", VAR_BOUND, 1, BKFR_INT, &c_wrk_opt_len);
   AddVariableCond(VAR_BOUND, MIN_cmp_fn);
   *(int*)GetVariablePtr(VAR_BOUND) = INT_MAX;


   compute->proc_register_variable(
       "NUMBERS", VAR_NUMBERS, 0, BKFR_INT, NULL);
   AddVariableFn(VAR_NUMBERS, sumset_ReceivedNumbers, 1);
   compute->proc_register_variable(
       "SUM", VAR_SUM, 1, BKFR_INT, &c_sum);
}


/* PATH REPOSITORY */
int sumset_GetInitPath(int *var_id)
{
   int i = GetVariableCount(VAR_PATH);
   compute->proc_register_temp_variable(
         var_id, i, BKFR_INT, VAR_PATH);
   return 0;
}


int sumset_SplitPath(int var_id, int *dst_var_id, int sup_var_id)
{
   int i = GetVariableCount(var_id);
   int *current_path, *new_path;
   compute->proc_register_temp_variable(
           dst_var_id, i, BKFR_INT, var_id);
   i++;
   SetVariableCount(var_id, i);
   SetVariableCount(*dst_var_id, i);
```

```
   current_path = GetVariablePtr(var_id);
   new_path = GetVariablePtr(*dst_var_id);

   /* current node gets 1
    * helper node gets 0
    */
   new_path[i-1]      = 0;
   current_path[i-1]  = 1;
   return 0;
}

int sumset_ComparePaths(int var_id1, int var_id2)
{
   int *path1 = GetVariablePtr(var_id1);
   int path1_len = GetVariableCount(var_id1);
   int *path2 = GetVariablePtr(var_id2);
   int path2_len = GetVariableCount(var_id2);
   int i = 0;

   while (1) {
      if (i==path1_len) {
        if (i==path2_len) return 0; /* equal */
        return 1; /* path 2 within path 1 */
      }
      if (i==path2_len) return -1; /* path 1 within path 2 */
      if (path1[i] > path2[i]) return 2;
                              /* path 1 comes after path 2 */
      if (path1[i] < path2[i]) return -2;
                              /* path 1 comes before path 2 */
      i++;
   }
   return 0;
}
```

Figure 12. Example of the actual code for the sum of subsets compute module

Multi-cluster computation with Path Repository

- `proc_get_init_path` – returns an empty array variable
- `proc_split_path` – splits the VAR_PATH into VAR_PATH and a new variable
- `proc_compare_path` – compares two variables (paths)
- INIT_CLUSTER_PATH function – the same as INIT_HELPER_PATH
- CLUSTERS_HELPER_PATH function – the same as INIT_HELPER_PATH

3.14.2 SAT modules implementation

In view of its fundamental position in the domain of NP-complete problems, the SAT problem has received much attention in the literature. In fact, there is an annual conference *International Conference on Theory and Applications of Satisfiability Testing* devoted entirely to issues relating to the SAT problem, including the development of SAT solvers. The best SAT solvers use a variety of heuristics, resolution methods such as Davis-Putnam [41], lemmas, and complicated structures in order to speedup the search. Two SAT solvers, SBSAT [42] and zChaff[43], were added to BkFr as compute modules. In order to implement these solvers in BkFr, I had to add the following functions added to each of them:

a) create initial covering set – selecting first few variables
b) splitting function – for splitting the current PATH and handing part of it to a different worker
c) path comparison
d) periodic calling of the framework – so the framework can check for messages from other nodes

3.14.3 Teaching Template Evaluation

Using template modules I have developed a method of introducing the message passing parallel programming paradigm as a topic to be included in the standard upper-division algorithms course. One particularly important module consists of a sequential SAT solver, together with a parallel version created using a master-worker parallel design strategy and implemented using MPI on a Beowulf cluster. The SAT programming assignment has been used in several classes so far, with good success. Students really appreciate the opportunity to be exposed to message passing parallel programming. Another positive aspect to the SAT solver problem is the availability of standard I/O formats, as well as extensive benchmarks for evaluating code performance. As an additional incentive, I ran a competition of the correct solvers on a set of benchmarks, where the fastest solver received extra bonus points. This emulates a scenario that exists in the SAT solver research community in which an annual competition is run to determine the currently fastest SAT solver based on a test-bed of problem instances. The student is thereby given an insight into the current state of the art in this important topic.

3.15 Performance Results

In order to determine the performance characteristics of the backtracking framework, I first implemented the sum of subsets problem and found that it scaled well over multiple clusters. I then implemented the SAT problem using two existing SAT solvers, and found similar scalability results.

Table 4. Single cluster computation performance

	Standalone	**8 node cluster**	**16 node cluster**
Sum of subsets bkfr03.sumset	5:28	0:56	0:42
SAT SBSAT vdw_178_2_5.cnf[44]	14:19	3:12	1:49
SAT zChaff longmult10.cnf	4:32	0:49	0:33

Table 5. Multi cluster computation performance

	Single node	16 node cluster	16+32 node clusters
Sum of subsets bkfr03x.sumset	1:13:27	9:04	3:36
SAT SBSAT slider_120_unsat.cnf	1:07:13	7:14	5:06
SAT zChaff slider_100_unsat.cnf	50:59	4:13	2:35

The performance data were obtained on a cluster with dual CPU AMD MP-1700 with 1GB RAM. The 2nd cluster is dual 450MHz Pentium III with 1GB RAM. The number of nodes is the number of processes participating in the computation. In general one process = one processor.

3.16 Summary

I have shown that the BkFr framework can be used as a general platform to solve problems amenable to parallel backtracking on both the single cluster and the multiple cluster environments. Performance data for the problems that I have implemented to date indicates near linear scaling (on average) within one cluster or two clusters. When implementing new problems in the BkFr framework, the compute module plays the main role in determining how fast the problem will be solved. The performance numbers illustrate good speedup for the algorithms employed, but leave open the question of finding more clever algorithms. One of my goals has been to design a platform with sufficient generality that new algorithms for a given problem could be readily adapted to the BkFr framework. Another goal was to design a flexible framework allowing resources (clusters) to be added to an ongoing computation, thereby providing a robust computational platform for lengthy computations. The fault tolerant and Path Repository features of BkFr allow for unexpected disconnects of the joined clusters without a major impact on the computation.

While my performance runs show that BkFr provides good speedup on average for the problems considered, as with parallel depth-first search in general, some instances might not yield any speedup due to the skewed nature of the instance. For example, I encountered SAT instances that took longer than sequential version, which was probably due to a screwed heuristic, and the fact that I utilized limited lemma sharing in the version currently implemented. I hope to enhance my implementation of SAT in future work. I also are considering looking into a threaded environment that would replace polling using two threads. The first thread would wait for messages and the second thread would handle the computations. Another direction for future work might be the decentralization of the supermaster function to eliminate this single point of failure. This decentralization could by done by replicating changes in the Path Repository to a backup location.

Chapter 4 - Dynamic Interoperable Point-to-Point Connection of MPI Implementations

4.1 Acknowledgment

Parts of this chapter were taken from the following paper(s):

M. Kouril and J. L. Paul, *Dynamic Interoperable Message Passing*, *12th European Parallel Virtual Machine and Message Passing Interface Conference PVMMPI*, Springer Berlin / Heidelberg, Sorrento, Italy, 2005, pp. 167-174.

B. Di Martino et al. (Eds.): EuroPVM/MPI 2005, LNCS 3666, pp. 167 – 174, 2005.

I was jointly responsible for writing the manuscript with Prof. Jerry Paul and I received substantial editing assistance from him. The remainder of the work is mine. The co-author(s) acknowledge that majority of the work was done by me and grant their consent to use the material in this dissertation as it is.

4.2 Introduction

When connecting multiple clusters there are two main paradigms and concomitant programming environments, *static* versus *dynamic*. In the static environment, all participating processes are captured at the outset. In the dynamic environment, processes can join and leave during the execution of the computation. Using the Inter-Cluster Interface (ICI) library that I have developed, I have applied this dynamic environment to computationally intensive, hard problems (such as NP complete problems) with good speedup results. In particular, I have parallelized some SAT solvers, including SBSAT [42, 45] and zChaff [43], as well as my own special solver. The latter solver was targeted to determine exact values or improved lower bounds for van der Waerden numbers W(K,L) (see Chapter 5). Amongst other results, I have almost doubled the previously known lower bound on $W(2,6)$ from 696 to 1132. ICI together with my backtracking framework BkFr is being used in an attempt to establish that this bound is sharp. The computation is estimated to take several months, but it is critical that I use all available computing resources, including using heterogeneous clusters dynamically.

There is a large problem domain for which it is extremely advantageous to have both interoperability and dynamic cluster participation available to the programmer. Grid computing and the heterogeneity of the current grid make yet another case for having an interoperable way to dynamically connect running MPI programs. Current MPI standards do not explicitly support *interoperability* (communication between two different MPI implementations). On the other hand, the IMPI Standard, while it supports interoperability, is limited to the static environment.

In the next section I summarize the existing implementations available for static and dynamic message passing programming environments. I then introduce my methods of solving the interoperability issue in the dynamic setting and discuss the performance data that I obtained.

4.3 Current Static and Dynamic Message Passing Environments

In my scenario I assume that the applications I wish to connect are started independently, as opposed to being spawned from the currently running task. In other words, there is no previous connection between the running tasks. I also assume that the applications are compiled by different MPI implementations, therefore making it impossible to directly use the existing functions introduced in MPI-2. In Table 6 I list the dynamic/interoperability status of the current standards. Although using compatible MPI implementations on both ends of intercluster communication is an option, usually every cluster has an optimized MPI implementation installed and compatible implementations might not be available. The user should not be expected to recompile MPI implementation on each cluster. My intercluster interface library is small and provides the bare minimum for the intercluster communication, so that applications can take advantage of the optimized intracluster communication. ICI does not include collective communication functions at this time, since many applications, including those discussed in this dissertation, do not require this functionality.

Table 6. Dynamicity and interoperability supported by the current MPI Standards

Standard	**Participating nodes**	**Interoperability status**
IMPI	Static	Interoperable
MPI-1	Static	Non-interoperable
MPI-2	Dynamic	Non-interoperable
Not currently available	Dynamic	Interoperable

Spreading static MPI-1 computations among multiple clusters without changing the source code has been addressed by PACX-MPI [46], PLUS [47], MPI-Glue [48] and MPICH-Madeleine [49]. The number of nodes for such computation is statically preallocated and no new dynamic connections are

possible. The project StaMPI [50] also allows the spawning of new processes, but does not allow connecting independent processes. Additional projects touching on the subjects are MagPIe [51] and MPICH-G2 [52]. None of these projects cover dynamic connections of independent MPI implementations.

Interoperable MPI (IMPI) [10] addressed the problem of interconnecting different MPI implementations, but it is done statically on a group of preallocated nodes without the possibility of connecting additional nodes or clusters using the IMPI protocol.

Relative to the dynamic environment, MPI_Connect [53] allowed dynamic connections between clusters by forming inter-communicators. This project had similar goals as my project, but it is no longer supported. MPI_Connect implementation was based on MetaComputing system SNIPE. MPI_Connect was succeeded by Harness/FT-MPI [54] which is a light-weight MPI implementation with a focus on fault tolerance, which retained the dynamic communication capabilities from MPI_Connect. Harness/FT-MPI has its own MPI implementation, and therefore is not interoperable.

4.4 The MPI Standards Needed for My Solutions

I now briefly describe the parts of the MPI Standards necessary to implement my solutions given in section 4.5 (see the MPI Standards [5, 6] for additional details). MPI-2 extends the existing MPI-1 Standard by adding a number of new functions. In particular, communication functions were added between two sets of MPI processes that do not share a communicator, but no standard was mandated for interoperable communication. However, the new functions for establishing communication, such as `MPI_Comm_connect` and `MPI_Comm_join`, have a few associated specifics that are not interoperable, such as port names, `MPI_Info` parameters, and communication protocols. Thus, implementers must decide how to communicate (no protocol specification is given), thereby effectively making any communication implementation specific, i.e., not interoperable.

The MPI-2 Standard defines MPI_Init_thread (to be used instead of MPI_Init) specifically targeted towards threaded applications. This function returns the highest level of thread safety the implementation can provide.

Three functions (MPI_Pack_external, MPI_Pack_external_size and MPI_Unpack_external) were added that allow conversion of the data stored in the internal format to a portable format "external32". This format ensures that all MPI implementations will interpret the data the same way. For example "external32" uses big-endian for storing integral values, big-endian IEEE format for floating point values, and so forth. The MPI-2 Standard mentions that this could be used to send typed data in a portable way from one MPI implementation to another. In both of my solutions, ICI allows the utilization of these functions when they are available.

Finally, MPI-2 added a group of functions called *generalized requests* allowing for the definition of additional non-blocking operations and their integration into MPI. These functions are MPI_Grequest_start and MPI_Grequest_complete. MPI_Grequest_start returns a handle that can be used in MPI functions such as MPI_Test and MPI_Wait. The completion of the operation is signaled to MPI using MPI_Grequest_complete. Typically the non-blocking operation runs in a separate thread, so that it is only usable in a sufficiently thread-safe environment.

While MPI-2 also added functions for spawning new processes, they are not considered in this dissertation, since in my problem domain parallel applications are typically started independently.

4.5 Two Solutions for Dynamic Interoperable Communication

I will describe two solutions to the problem of establishing interoperable dynamic connections, where the first solution assumes the relevant functions from the MPI-2 Standard are available, and the second is suitable for environments in which some of the relevant functions in this standard are missing. In fact, the second solution creates a communication layer that is independent of MPI and is usable with other communication libraries. In the both solutions, a user could choose to call ICI functions directly and

thereby explicitly recognize when the communication is with a process of another MPI implementation. On the other hand, a user could use the profiling interface where interoperable communication is hidden from the user.

4.5.1 A Solution That Assumes a Thread-Safe Implementation of MPI-2

Together with ICI, the following set of functions from the MPI-2 Standard can be used to create dynamic interoperable MPI communication:

1. Canonical `MPI_Pack_external` and `MPI_Unpack_external` for interoperability.
2. Generalized requests for integration with MPI.
3. Full thread-safeness.
4. Profiling interface for transparent integration with MPI (optional).

The ICI library has to be used on both ends of the communication. The library uses the existing MPI implementations and ICI low level functions (such as `recv_data`, `send_data` and so on). The ICI functions use the Berkeley TCP/IP API interface, which is available on the majority of platforms.

In Table 7 I list the basic ICI communication functions required to support dynamic interoperable communication. These functions are counterparts to MPI functions having the same arguments, with the possibility of using the profiling interface. In Table 7 I only show a sample set of MPI functions and their ICI counterparts. The full set would contain all MPI functions that allow communication using point-to-point communicators.

Table 7. A sample set of ICI functions

MPI	ICI counterpart	Description
MPI_Send	ICI_Send	Send data, blocking
MPI_Recv	ICI_Recv	Receive data, blocking
MPI_Isend	ICI_Isend	Send data, non-blocking
MPI_Irecv	ICI_Irecv	Receive data, non-blocking
MPI_Probe	ICI_Probe	Blocking wait for message
MPI_Comm_connect	ICI_Connect	Initiate connection, blocking
	ICI_Iconnect	Initiate connection, non-blocking

4.5.2 A Solution for Implementations Missing Some Functionality of The MPI-2 Standard

For interoperability, I encode the data using the MPI-2 "external32" format by `MPI_Pack_external` and `MPI_Unpack_external`. This encoding incurs overhead during sending and receiving data, but guaranties interoperability, not only among implementations, but also among platforms and high-level languages utilized.

Generalized requests are used to integrate inter-cluster communication into the existing MPI code. Non-blocking ICI_Isend and ICI_Irecv calls setup a request handle using MPI_Grequest_start, execute in a separate thread and call MPI_Grequest_complete once the operation is complete. It is therefore possible to use ICI just like MPI for point-to-point communication; that is, analogous to communication using an inter-communicator. Since the ICI communication happens in a separate thread,

MPI_Grequest_complete is called in a separate thread and therefore the MPI implementation has to be fully thread-safe (see Section 4.6).

I chose not to use the MPI profiling interface to create a uniform API interface for intra-cluster and inter-cluster communication, since its use would negatively impact performance. However, it could be added in future ICI implementations.

In addition to dynamic interoperability, the ICI library allows implementing properties such as encryption, authentication, tunneling, and so forth.

Although some of the MPI implementations comply with the MPI-2 Standard, at the time of writing this dissertation the implementation level and thread safety is low (see Section 4.6). I now discuss how the solution in Section 4.5.2 can be adjusted for the current state of the implementations when one or more of the following functionalities are missing. In the special case where MPI is not even available, the missing MPI functionality can be easily obtained by substituting appropriate ICI functions.

1. Implementations missing the canonical `MPI_Pack_external` and `MPI_Unpack_external` functions. These functions are not only platform dependent but also language dependent. However, it is feasible to implement them using `MPI_Type_get_envelope` and `MPI_Type_get_contents` which are generally available.
2. Implementations missing generalized requests. Instead of integrating ICI communication with MPI, I substituted blocking functions that potentially use both MPI and ICI communication, such as using `MPI_WaitAny` with a polling loop of non-blocking tests of ICI and MPI communication. In my applications the communication is not the critical component, and therefore the polling overhead is acceptable.
3. Implementations missing thread–safety. This can be overcome by implementing not only blocking ICI communication within ICI function calls, but also enabling progress checks on non-

blocking ICI communication within all ICI function calls. In my pseudocode this is called the `do_background_ops` function.

This solution was implemented as part of the BkFr project, and tested with MPI-1 implementations MPICH [7] and LAM[9] with good results.

4.6 Evaluation

The first solution places the most requirements on the implementation level of the MPI-2 Standard. Moreover, this solution requires full thread-safety, which renders it unusable for the widely available MPI implementations shown in Table 8. Even in cases where MPI_Pack_external is not available MPI_Type_get_contests and MPI_Type_get_envelope are available.

Table 8. MPI-2 implementation level in various MPI implementations

	Thread safety	**MPI_Grequests**	**MPI_Pack_external**
MPICH 1.2.7	FUNNELED	N	N
MPICH2 1.0.2	FUNNELED	Y	Y
MPICH G2 1.2.5.3	SINGLE	N	N
LAM 7.1.1	SERIALIZED	N	N
SUN HPC 5.0	SINGLE	Y	Y

My tests of the second solution show a comparison of MPICH 1.2.7, LAM 7.1.1 with my implementation of the ICI communication with and without encoding the messages into an interoperable format “external32.” The data were measured on AMD MP-1800+ cluster connected by 1000Base-T adapters using modified perftest 1.3a (a performance testing tool included with MPICH). MPICH and LAM data were measured using perftest as a communication performance between two nodes of the

cluster during intracluster communication. I also used perftest to measure communication performance of the ICI library as described in the Section 4.5.2 between the same two computers as used with LAM and MPICH. In the prototype the exchanged messages were MPI_INT typed arrays.

Figure 13 shows the performance for small and midsize messages of length up to 8K. LAM outperforms both ICI and MPICH, whereas ICI with and without "external32" outperforms MPICH for small messages (size < 1k). Figure 14 shows performance for long messages. The overhead of "external32" encoding is obvious and ICI with the "external32" encoding is outperformed by LAM and MPICH. Although the performance was not my primary goal for small and large messages (size<1000 and size>32K), the ICI implementation without "external32" encoding outperforms MPICH 1.2.7. For small messages the "external32" encoding poses only negligible overhead, which increases with the increasing message size up to 58%, after which it remains constant (see Figure 15).

The performance of the BkFr project utilizing ICI is shown in the Table 9 and further described in Chapter 3.

Table 9. Performance of BkFr without and with ICI communication

	One node	One cluster	2 clusters connected via ICI
BkFr Sum of subsets	1:13:27	9:04	3:36
BkFr SAT SBSAT	1:07:13	7:14	5:06
BkFr SAT zChaff	50:59	4:13	2:35

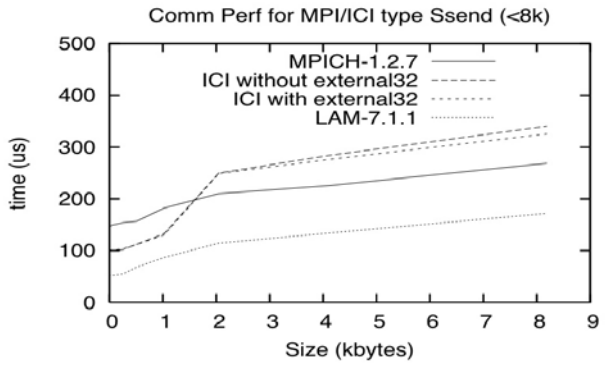

Figure 13. Performance data for small and midsize messages

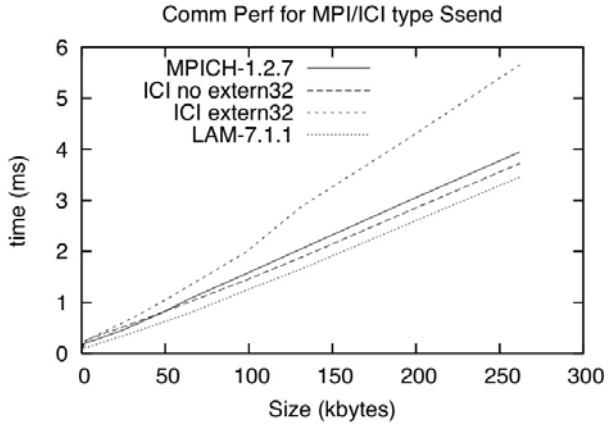

Figure 14. Performance data for long messages

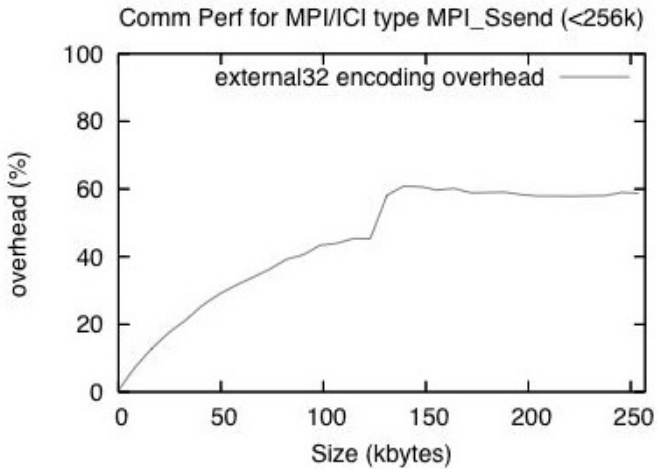

Figure 15. Perf. data for encoding overhead

4.7 Summary

There are many MPI applications where it would be very advantageous to dynamically add clusters, with possibly varying MPI implementations, to an ongoing computation. Using resources dynamically as they become available is currently possible, but the MPI-2 Standard lacks interoperability. I presented two solutions to the dynamic interoperable problem, both of which isolate inter-cluster communication within a separate library layer (ICI), thereby possibly introducing stronger fault tolerant capabilities for inter-cluster communication. The first solution utilizes MPI-2 general requests, threads and "external32" encoding. The second solution provides adjustments to the first solution assuming that some parts of the MPI-2 Standard are not implemented, including the absence of thread safeness. I have implemented a prototype version of the second solution, and successfully utilized it in a general backtracking framework BkFr over multiple clusters, as well as verifying its feasibility in various environments and implementations. The results show that ICI efficiently handles intercluster communication, and that BkFr running SAT instances that scale well within the cluster continue to scale well into multiple clusters. I hope that dynamic interoperable communication will eventually become part of the MPI Standard.

Chapter 5 - Van der Waerden Numbers SAT Solver

5.1 Acknowledgment

Parts of this chapter were taken from the following paper(s):

M. Kouril and J. Franco, *Resolution Tunnels for Improved SAT Solver Performance*, *Eighth International Conference on Theory and Applications of Satisfiability Testing*, Springer Berlin / Heidelberg, St. Andrews, Scotland, 2005, pp. 143-157.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 143 – 157, 2005.

I was jointly responsible for writing the manuscript with Prof. John Franco and I received substantial editing assistance from him. The remainder of the work is mine. The co-author(s) acknowledge that majority of the work was done by me and grant their consent to use the material in this dissertation as it is.

5.2 Introduction

In the 1920s B.L. Van der Waerden proved the following result about partitions of sufficiently large initial intervals of positive integers {1, 2, …, n}: given positive integers K and L, there exists a (smallest) integer n = W(K,L) such that whenever {1, 2, …, n} is partitioned into *K* disjoint classes, at least one of the classes contains an arithmetic progression of length L. A partition of {1, 2, …, n} into K classes is also called a *k-coloring* of {1, 2, …, n}, and an arithmetic progression contained in one of the classes is called *monochromatic*. There is no known closed form expression for W(K,L) and all but five of the first few numbers are unknown. Table 10 shows all the known Van der Waerden numbers. In 1979 W(4, 3) became the most recent addition to this table.

Table 10. Known Van der Waerden numbers

K \ L	**3**	**4**	**5**
2	9	35	178
3	27		
4	76		

Upper and lower bounds on some of the remaining numbers have been derived but they are so far apart that they are of little practical use. An unpublished general upper bound is $W(K,L) \le e^{e^{(1/K)^{e^{e^{(L+110)}}}}}$ [55], and a general lower bound, due to the Lovasz local lemma, is $W(K,L) > \left(\frac{K^L}{eLK}\right)(1+o(1))$.

Work on specific Van der Waerden numbers has sharpened some of these bounds as the results of Table 11 (taken from [56]) show.

Table 11. Lower bounds for some Van der Waerden numbers

K \ L	3	4	5	6	7	8
2	9	35	178	>1131	>3703	>11495
3	27	> 292	> 1209	>8886	>43855	>238400
4	76	>1048	>17705	>91331	>393469	
5	>125	>2254	>24045	>246956		

The number W(K,L) can be found by determining whether solutions exist for certain formulae of a class of CNF formulae described in Figure 16. I refer to a formula of this class, with parameters n,K,L, by $\psi(K,L)^n$. A solution exists for $\psi(K,L)^n$ if and only if n < W(k,l). So, several of these formulae may be solved for various values of n until that boundary is reached. In this manuscript $\psi(K,L)^n$ is treated as a set of clauses to make some algorithmic operations easier to express.

Variables	Subscript Range	Meaning
$v_{i,j}$	$1 \le i \le n, 1 \le j \le k$	$v_{i,j} \equiv 1$ iff $i \in C_j$
Clauses	**Subscript Range**	**Meaning**
$\{\bar{v}_{i,r}, \bar{v}_{i,s}\}$	$1 \le i \le n, 1 \le r < s \le k$	i is in at most one class
$\{v_{i,1}, \ldots, v_{i,k}\}$	$1 \le i \le n$	i is in at least one class
$\{\bar{v}_{r,j}, \bar{v}_{r+1,j}, \ldots, \bar{v}_{r+l-1,j}\}$ $\{\bar{v}_{r,j}, \bar{v}_{r+2,j} \ldots, \bar{v}_{r+2(l-1),j}\}$ ... $\{\bar{v}_{r,j}, \bar{v}_{r+t,j} \ldots, \bar{v}_{r+t(l-1),j}\}$	$1 \le r \le n-l+1$ $1 \le j \le k$ ... $t = \lfloor (n-r)/(l-1) \rfloor$	no arithmetic progression of length l in C_j

Figure 16. Formula $\psi(K,L)^n$ for finding Van der Waerden numbers. Equivalence classes are named C_1, C_2, ..., C_k for convenience.

The nature of $\psi(K,L)^n$ is such that the number of solutions increases with n up to a point, then decreases. The formulae, as expected, become more difficult in the latter range. This difficulty may prevent the boundary from being reached or even approached. In that case, there is no choice but to accept a lower bound which is the largest n for which a solution to $\psi(K,L)^n$ is found.

5.3 Analysis of the known Van der Waerden numbers

I analyzed the known Van der Waerden numbers to better understand the unknown Van der Waerden numbers. I will use 0/1 to identify colors for K=2 colorings. When K>2, the colors will be identified by 1,2,4,8.

5.3.1 W(2,3)=9

The extreme partition has 6 possible colorings:

11001100, 00110011, 10011001, 01100110, 01011010, 10100101

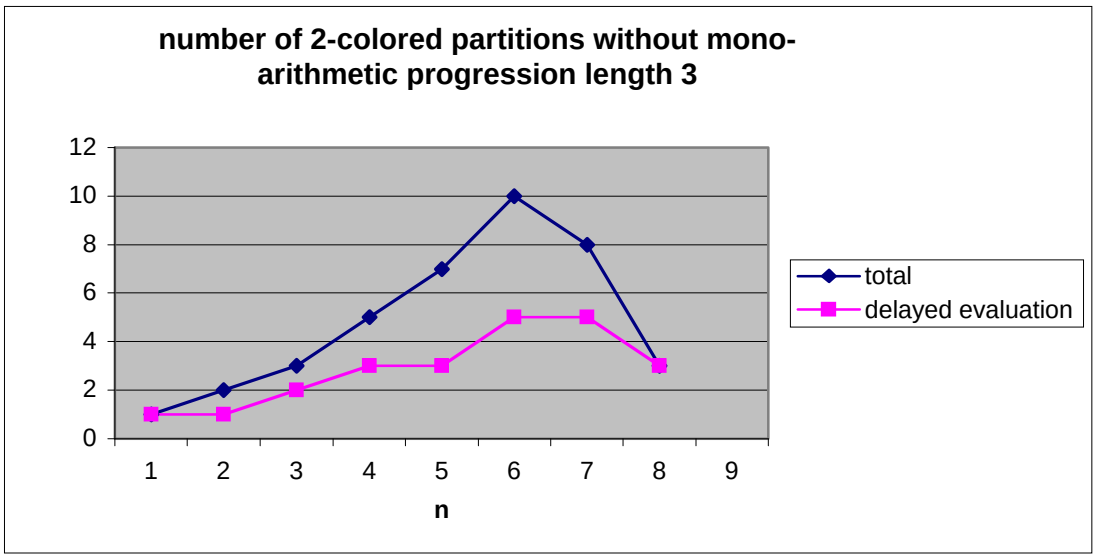

Figure 17. Number of 2-colored partitions without mono-arithmetic progression length 3

5.3.2 W(2,4)=35

The extreme partition has 28 possible colorings:

```
.0100011101.0100011101.0100011101.
.1011100010.1011100010.1011100010.
```

where four dots are substituted by one of the valid 14 colorings.

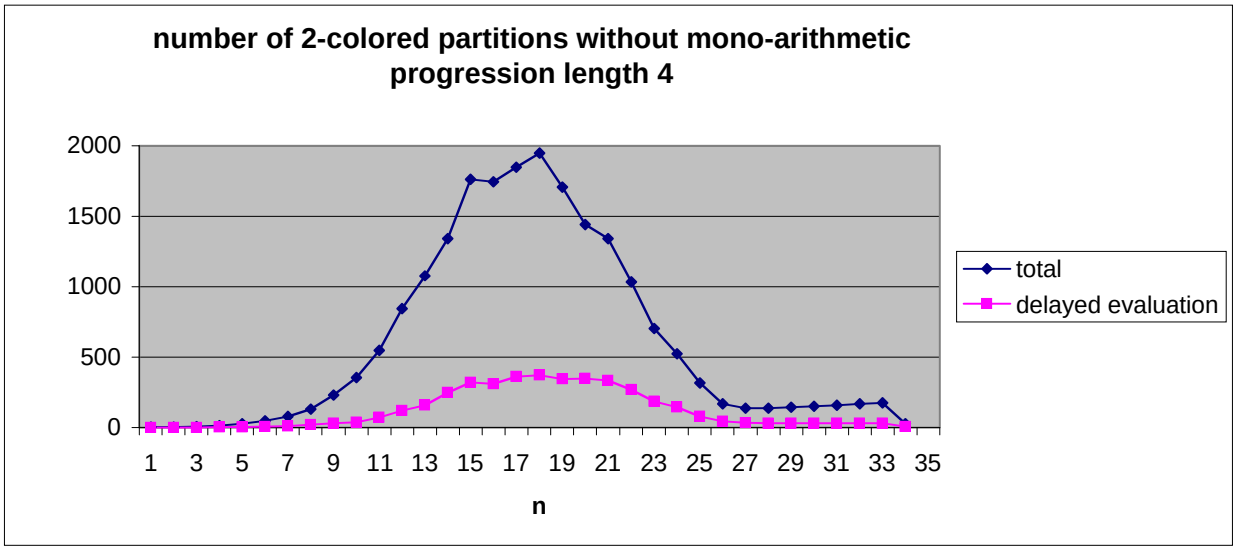

Figure 18. Number of 2-colored partitions without mono-arithmetic progression length 4

5.3.3 W(2,5)=178

The extreme partition has 193624 possible colorings:

```
.0010000100.0111010001.1101111011.1000101110.↵
 0010000100.0111010001.1101111011.1000101110.↵
 0010000100.0111010001.1101111011.1000101110.↵
 0010000100.0111010001.1101111011.1000101110.

(reversed)
.0111010001.1101111011.1000101110.0010000100. (4x)

(negated)
.1101111011.1000101110.0010000100.0111010001. (4x)

(reversed and negated)
.1000101110.0010000100.0111010001.1101111011. (4x)
```

where the 17 dots are substituted with one of 48406 valid colorings.

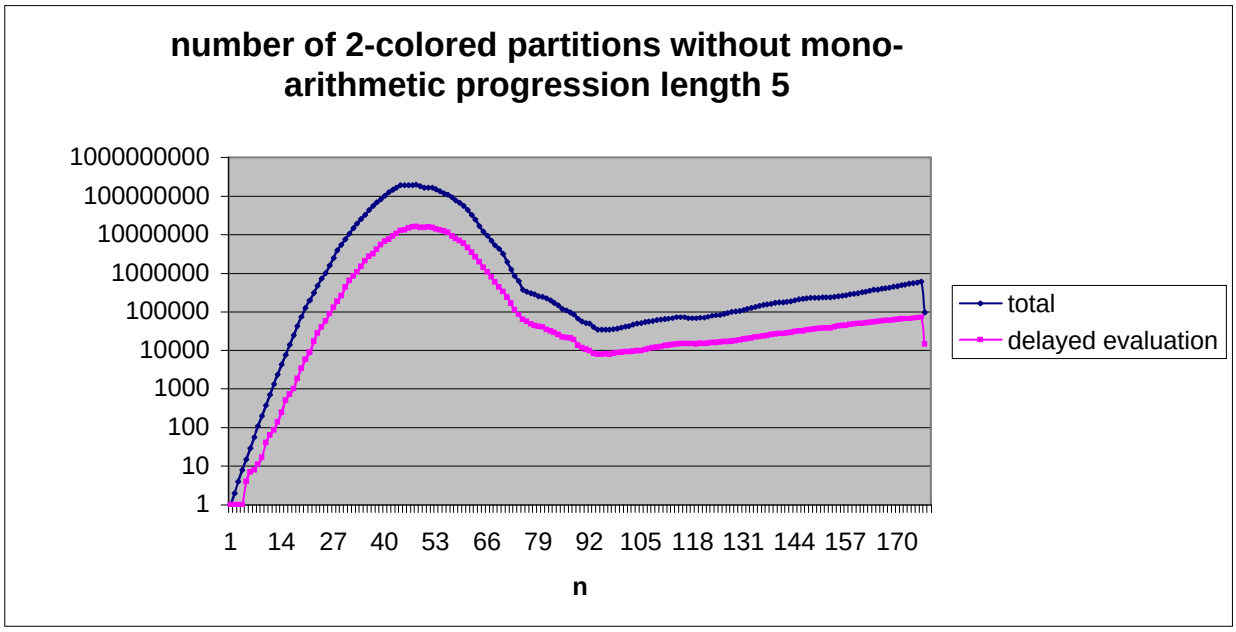

Figure 19. Number of 2-colored partitions without mono-arithmetic progression length 5

5.3.4 W(3,3)=27

The extreme partition has 48 possible colorings:

```
1) 112241141422442241411422 11
2) 112421141422442241411242 11
3) 112211242441411212241442 42
4) 121121244224421211244114 14
5) 121221142242411441142422 41
```

First two lines are palindromes so the recoloration only yields 3*2=6 patterns for each line. The remaining 3 lines are not palindromes therefore the reverse is not identical and should be also counted totaling in 12 possible patterns for each line.

5.3.5 W(4,3)=76

The extreme partition has 1440 possible colorings:

```
.12448818114182218282488241442422118 4.12448818114182218282488241442422118 4.
```

Both parts are the same and both of them are palindromes. There is 60 possible colorations of the dots (all combinations 4^3without all the same colors = 4*4*4 – 4). In addition there are 4*3*2=24 recolorations of the basic pattern totaling 24*60=1440 possible colorings.

5.3.6 W(2,6)=1132

Based on the results obtained so far (having searched about 80% of the W(2,6) initial patterns):

```
(basic pattern (len 56):
B = 01111001000001011110111001111101101110100011101001010011
. BB_rev,neg . B_negB_rev . BB_rev,neg . B_negB_rev . ↵
  BB_rev,neg . B_negB_rev . BB_rev,neg . B_negB_rev . BB_rev,neg . B_negB_rev .

.
 01111001000001011110111001111101101110100011101001010011↵
 00110101101000111010001001000001100010000101111101100001.
 10000110111111010000100011000001001000101110001011010110 0↵
 11001010010111000101110110111110011101111010000010011110.
 (5x)

(negated)
.
 10000110111111010000100011000001001000101110001011010110 0
 11001010010111000101110110111110011101111010000010011110.
 01111001000001011110111001111101101110100011101001010011
 00110101101000111010001001000001100010000101111101100001.
 (5x)
```

Where the 11 dots are substituted with one of 888 valid colorings. The 2 operations – reverse, negation and rotation+negation will quadruple this to the total of 3552.

5.3.7 Analytical expression of number of valid partitions for W(2,L)

In an attempt to find a closed form expression for the number of valid colorings for various values of n, the following recurrence relation arises for the number of valid colorings when the gap in the progressions is 1 (it seems that for higher gaps such recurrent relations do not exist).

N(0)=2

N(1)=2

N(2)=4

...

N(L-1)=2^(L-1)

N(n)=N(n-1)*2 - N(n-L)

Analysis of the progression gaps>1: For every L-1 additional colored number new additional gap kicks in. Eventually the number of arithmetic progressions with greater and greater gaps starts to influence the number of possible partitions from the upward (growing) trend to the downward trend. As shown in the W(2,4) and W(2,5) this downward trend hits a bottom of the valley and starts growing again.

5.4 Van der Waerden numbers and Satisfiability

SAT solvers have been applied to the above formulations with the results shown in Table 12 (taken from [57]), all lower bounds. Except in one case, all these bounds are greatly inferior to those obtained analytically.

Table 12. Bounds on Van der Waerden numbers obtained by SAT solvers

K \ L	**3**	**4**	**5**	**6**	**7**	**8**
2	9	35	178	> 341	> 614	> 1322
3	27	> 193	> 676	> 2236		
4	76	> 416				
5	> 125	> 880				
6	> 194					

Resolution is a general procedure that may be used to determine whether a given CNF expression has a model and to supply a certificate of unsatisfiability if it doesn't. The idea predates the often cited work reported in [58] and for decades resolution has been one of the primary engines for CNF SAT solvers. In the last 10 years tree resolution, in the form of variants of DPLL [41], has given way to DAG resolution through the introduction of clause learning and recording during search. This and other ideas have led to a spectacular improvement in the performance of resolution-based SAT solvers. Hardware and algorithmic improvements have together contributed to perhaps an order 10^9 speed-up in SAT solving over the past 15 years and, consequently, some problems considered very difficult then are now considered trivial. But there remain many problems, which are considered hard, for example in the important domains of protocol and hardware verification.

The last 15 years has also seen some brilliant theoretical work that has revealed exponential lower bounds for tree and DAG resolution, and has illuminated the reasons for it, when resolution is applied to ``sparse," unsatisfiable CNF formulae (e.g. [59-65]).

In such cases, very large resolvents must be created first, then resolved to get the smaller clause constraints that play a significant role in establishing the refutation. Generating the large resolvents is expensive, particularly since exponentially many have to be generated. In DPLL terms, this means a search space of great breadth may have to be explored. Metaphorically, one may think of search breadth as a mountain that must be climbed; on the other side of the mountain the search breadth may be significantly reduced due to the short resolvents that are finally generated. Clearly, one needs to find some way to ``tunnel" through this mountain and arrive quickly in the fertile valley of low-breadth search space, if it exists. It is one of the the aims of this dissertation to explore this possibility for Boolean expressions with a particular structure.

In the Satisfiability literature, the term ``tunnel'' has been applied to Stochastic Local Search algorithms, a class which includes members of the WalkSAT, GSAT, and other families [66-70]. In that context, one may think of a location as an assignment of values to variables and the height at a location as the number of constraints falsified by the assignment at that location. Then, for a given, hard Boolean expression, there are generally many mountain peaks and the objective is to locate and enter the deepest valley. Changing the value of a single variable merely moves the current location up and down the side of a mountain but changing values of several variables permits ``tunneling'' into another valley. This use of tunneling is to be distinguished from the way I use it: in my context there is one principal mountain to get over or around and the valley on the other side can be quite wide. To reach the top of my mountain one must wait for many large constraints to be learned. But, in many cases, one cannot afford to wait: to reach the valley in a reasonable time, *constraints must be efficiently added before they are learned*.

There are several ways to do this, some safe and some risky. A reasonably efficient method for finding a safe tunnel through the mountain has been identified in [71] for a class of non-CNF formulae. I would like to be able to add constraints aggressively without having to worry about retracting them and, as I show in this dissertation, this can sometimes be done to achieve a result that is an approximation to the actual result. In other words, the extra constraints may prevent an optimal solution from being returned, but are weak enough to admit suboptimal solutions that are not far from optimal. However, if the extra constraints are too weak, a solver may take too much time and not find a good suboptimal result. So, a good ``heuristic'' is needed for adding constraints in some optimal way.

At this point in time, it seems the spectacular tunneling success I seek, which will be a consequence of my choice of tunnel heuristic, is practical only by a careful analysis of the specific structure of a given formula. In this dissertation I underscore this point by developing aggressive tunnel heuristics for formulae associated with the problem of finding Van der Waerden numbers (described in the next section). Such formulae are currently extremely difficult for off-the-shelf SAT solvers, even though most of them are satisfiable. By adding aggressive tunnel constraints to the formulae, however, I

are able to find the best bound yet, by far, for W(2,6). My analysis serves as motivation for attempting aggressive tunnel heuristics for other hard problems.

When I employ aggressive tunneling techniques, I can increase the lower bound for W(2,6) to 1132 from the previously known best value of 696 [72, 73] shown in Table 11) (that itself far exceeds the best prior bound of 342 obtained by SAT solvers and shown in Table 12). The reason I have a bound instead of the actual number is that aggressive tunneling forces my SAT solver to be incomplete. I emphasize that although an incomplete solver precludes finding a refutation for the formula, it does provide an apparently tight bound for W(2,6). I believe the bound is tight because it is obtained using three widely different tunnels. In addition, using tunnels aggressively I have found a bound for W(3,4) which matches the best analytic bound shown in Table 11 and greatly exceeds the best bound obtained by SAT solvers as shown in Table 12. This is further evidence of the tightness of bound in the presence of these aggressive tunnels.

My interest in examining tunnels for Van der Waerden numbers is due partly to this being an interesting problem to many mathematicians, but mainly because the propositional formulae for solving Van der Waerden numbers have a structure that is similar to formulae found in Bounded Model Checking and other practical applications. Therefore, I view this work as a preliminary to investigations on problems in the area of Formal Verification, among others.

Formulae showing structural similarities to Van der Waerden formulae do so, in part, because their corresponding problems are typically circuit queries with a time dependent nature, such as verification problems. In such problems a circuit is fixed so there are numerous repetitions of subformulae, each corresponding to the circuit properties at a different time step. The subformulae differ in the variables they contain. This structure is characteristic of the Van der Waerden formulae shown in Figure 16. This structure sometimes is partly responsible for difficult formulae since the making of small inferences is delayed considerably in such cases.

5.4.1 CNF representation of Van der Waerden numbers

For K=2 assign each number to color a binary variable v_i. Each clause in the resulting CNF will prevent an arithmetic progression of length L. Add one clause for each arithmetic progression.

For example, to prove that W(2,3)=9 there are two possible approaches:

a) The SAT solver finds at least one solution for W(2,3)=8 and then finds no solution to W(2,3)=9
b) The SAT solver finds all solutions to W(2,3)=8 and then it is easy to show that none of the found solutions is extendable

The CNF representations of W(2,3)=8 and W(2,3)=9 are in the Figure 20. and Figure 21. respectively.

```
(1 2 3) (2 3 4) (3 4 5) (4 5 6) (5 6 7) (6 7 8)
(1 3 5) (2 4 6) (3 5 7) (4 6 8) (1 4 7) (2 5 8)
(-1 -2 -3) (-2 -3 -4) (-3 -4 -5) (-4 -5 -6) (-5 -6 -7) (-6 -7 -8)
(-1 -3 -5) (-2 -4 -6) (-3 -5 -7) (-4 -6 -8) (-1 -4 -7) (-2 -5 -8)
```

Figure 20. CNF representation of W(2,3)=8

```
(1 2 3) (2 3 4) (3 4 5) (4 5 6) (5 6 7) (6 7 8) (7 8 9)
(1 3 5) (2 4 6) (3 5 7) (4 6 8) (5 7 9) (1 4 7) (2 5 8) (3 6 9) (1 5 9)
(-1 -2 -3) (-2 -3 -4) (-3 -4 -5) (-4 -5 -6) (-5 -6 -7) (-6 -7 -8) (-7 -8 -9)
(-1 -3 -5) (-2 -4 -6) (-3 -5 -7) (-4 -6 -8) (-5 -7 -9) (-1 -4 -7) (-2 -5 -8)
(-3 -6 -9) (-1 -5 -9)
```

Figure 21. CNF representation of W(2,3)=9 (bold are addition clauses vs. W(2,3)=8)

5.5 Computing the lower bound

In order to show a lower bound for W(K,L) it is enough to provide a coloring of the length n such that there is no monochromatic progression of length L, since then W(K,L) > n. There are various methods to reach such lower bound [57, 72].

I explored tunneling – aggressive adding uninferred constraints to the problem.

5.6 The Lower bound W(2,6)>=1132

When considering the case K=2, I reinterpret variables to remove some of the constraints shown in Figure 16. The formulae I consider are described in Figure 22. They use single index variables. For purposes of discussion, variable indices have been translated so that v_0 and v_1 are the middle variables. In doing so, the number of variables is always even. It is straightforward to consider odd variable formulae as well and I leave this for the reader. In what follows, n is even when considering the formulae $\psi(2,6)^n$.

Variables	Subscript Range	Meaning
v_i	$-n/2 < i \leq n/2$	$v_i \equiv 1$ if $i + n/2 \in C_1$ $v_i \equiv 0$ if $i + n/2 \in C_2$
Clauses	**Subscript Range**	**Meaning**
$\{\bar{v}_i, \bar{v}_{i+1}, \ldots, \bar{v}_{i+5}\}$ $\{\bar{v}_i, \bar{v}_{i+2} \ldots, \bar{v}_{i+10}\}$ ... $\{\bar{v}_i, \bar{v}_{i+t} \ldots, \bar{v}_{i+5t}\}$	$-n/2 < i \leq n/2 - 5$ ... $t = \lfloor (n/2 - i)/5 \rfloor$	no arithmetic progression of length 6 in C_1
$\{v_i, v_{i+1} \ldots, v_{i+5}\}$ $\{v_i, v_{i+2} \ldots, v_{i+10}\}$ ... $\{v_i, v_{i+t} \ldots, v_{i+5t}\}$	$-n/2 < i \leq n/2 - 5$ ... $t = \lfloor (n/2 - i)/5 \rfloor$	no arithmetic progression of length 6 in C_2

Figure 22. Formula $\psi(2,6)^n$, n even, for finding W(2,6). Classes are named C1, C2, ..., Cn for convenience.

5.6.1 Motivating the Use of a Tunnel – Analyzing $\psi(2,L)^n$

The approach to finding W(2,6) that is described below is the result of an analysis of the performance of an off-the-shelf SAT solver on formulae $\psi(2,L)^n$ (Figure 16), and patterns of variable assignments satisfying those formulae.

Figure 23 shows SAT solver performance on $\psi(2,5)^n$, for various values of n. The vertical axis measures the number of nodes of the search space at the depth indicated by the horizontal axis. In all

cases, the entire search space was explored, even if the input formula was satisfiable, but the results are similar if the solver stops immediately upon discovering a solution. Although the displayed results have been obtained using stock settings, similar results, which are not shown, apply for various settings of SAT solver parameters. According to Figure 23 there is a performance mountain that has roughly the same shape, regardless of n. The mountain tails off to a point of little significance after rising to a formidable peak. In addition to the mountain is a smaller peak, which behaves more like a wave since it always appears near n.

From performance curves and known Van der Waerden numbers and bounds I have observed the following:

Observation 1: Consider a performance plot of search breadth vs. depth for any common SAT solver applied to $\psi(2,L)^n$. Such a plot has two maxima, the greatest of which occurs at approximately the same depth, say W(2,L)/(2(L-1)), for n > W(2,L)/(L-1). The value of the greatest maximum is approximately the same for n>W(2,L)/(L-1) and several orders of magnitude greater than the search breadth at depth W(2,L)/(L-1).

Observation 2: $W(2,L) \approx L*W(2,L-1)$, at least for small L.

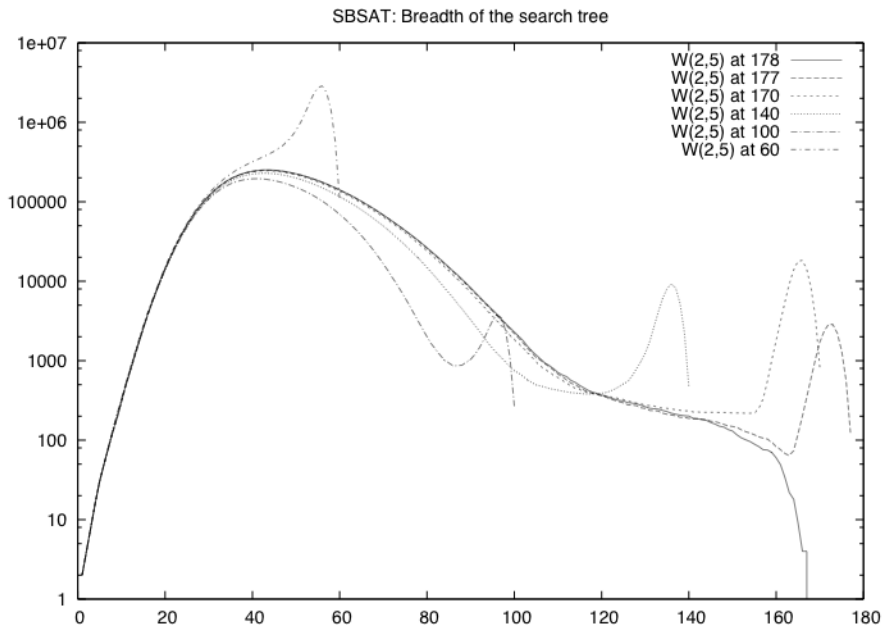

Figure 23. Typical SAT solver performance on $\psi(2,5)^n$ for various values of n.

Each curve in Figure 23 shows the breadth of the search space at the search depth indicated on the horizontal. Curves shown are due to SBSAT, which is a SAT solver currently under development at the University of Cincinnati. Other SAT solvers have been shown to perform similarly.

From the above observations I propose a way to solve the very difficult W(2,6) formulae. Start by searching for solutions to $\psi(2,6)^{n_0}$ where n_o is large enough so that the mountain *can* be crossed but smaller than the suspected value of W(2,6). By Observations 1 and 2, n_o should be greater than about 210. To get through the mountain, append a *tunnel* to the formula. I have developed three applicable tunnels which are specially designed to take advantage of certain structural or analytical characteristics of the formulae and are described separately in Section 5.6.3. The mountain is crossed when the search breadth is considered ``low." From then on, by Observation 1, the search is accomplished efficiently no matter what. Retract the tunnel. Add clauses so that the solver is effectively seeing $\psi(2,6)^{n_0+x}$ for x=1,2,3,... until search breadth falls to 0. The depth at which this occurs is a bound on W(2,6).

Details are given in the following sections.

5.6.2 Procedure for Finding a Bound on W(2,6)

I used a SAT solver designed specifically to solve formulae $\psi(2,L)^n$, and which incorporates special optimizations as outlined in Section 5.6.8. The outline below briefly describes the search process using the special solver.

1. The solver is started on the input $\psi(2,6)^{n_0}$, for some even n_o, plus a collection of clauses representing the tunnel. The parameter n_o is determined according to the description given in Section 5.6.1. There are three types of tunnels that have been applied. Detailed descriptions are given in Section 5.6.3. All three yield the same lower bound on W(2,6) as stated in Section 5.6.3.
2. Instead of a depth-first, or even priority-driven evaluation of the search space, as is commonly practiced, the solver conducts a strictly breadth-first evaluation. I chose breadth-first search to find all solutions (not just the first one) so I can at the point of n_o remove the tunnel and continue the search by extending the existing set of solutions. The order in which variables are considered for evaluation is fixed for all branches of the search tree, regardless of values assigned previously. The order represents a reflection around the center variable and is given as follows, from left to right: v_0, v_1, v_{-1}, v_2, v_{-2}, ... This is undoubtedly an inferior choice with respect to building the entire search space. Reasons for this choice are given in Section 5.6.7.
3. The search reaches depth n_o because there are so many solutions to $\psi(2,6)^{n_0}$ and the tunnel constraints do not filter some of them. At this depth all variables of $\psi(2,6)^{n_0}$ have been assigned values on all leaves of the search tree. Clauses from the set $\psi(2,6)^{n_0+1}$ - $\psi(2,6)^{n_0}$ are then added to the clause database of the solver and the search commences as before. The tunnel is retracted.

Remark: The tunnel has been designed so that the mountain has been crossed, for the most part, by this time. From this point on, the breadth of the search space is moderately small because the values of many variables are inferred on all branches, so the search continues quickly.

4. The following is repeated for m=2,3,... until the search breadth becomes 0: when the search depth reaches n_o+m-1, clauses from $\psi(2,6)^{n_0+m}$ - $\psi(2,6)^{n_0+m-1}$ are added to the solver's clause database and the search continues. When the search breadth becomes 0, a lower bound of n_o+m is found for W(2,6).

Remark: Upon completion of every iteration of this step, a non-zero search breadth means at least one solution for $\psi(2,6)^{n_0+m-1}$ exists, hence n_o+m is a lower bound for W(2,6).

Remark: There is no clause recording. Experiments show that zChaff, for example, does not benefit from clause recording on this family of formulae. I believe this is because the structure of the formulae is such that inferences can only be determined at high search depth. This is why I needed the tunnels in the first place.

With the above modifications to the SAT solver, and the improved formulation shown in Figure 22, a greatly improved bound for W(2,6) was obtained. However, this was not the case if no tunnels had been added at the outset.

Below I present performance figures for zChaff (v.2004.11.15) and Berkmin (v.561) as well as the special SAT solver described above. It is important to realize that zChaff and Berkmin can succeed with tunnels only if some mechanism is provided for determining when the other side of the mountain is reached. The mechanism I used was to apply my special solver up to Step 3. Then I removed the

tunnel, and sequentially applied zChaff or Berkmin to the given formula for *every* partial assignment not falsified at depth n_o until either a solution is found or it is determined that no solution is possible.

5.6.3 The Tunnels

Three different aggressive tunneling techniques to improve SAT solver performance are described in the subsections below. All three yield the same new lower bound of 1132 for W(2,6). This is significant for two reasons: 1) this is a big improvement over the previous best bound of 696[72, 73]; 2) since three different techniques stopped at the same point, I conjecture W(2,6)=1132.

5.6.4 First Tunnel

The first tunnel arises from an analysis of solutions to $\psi(2,L)^{W(2,L)-1}$ (recall W(2,L)-1 is the greatest n for which a solution to $\psi(2,L)^n$ exists). Figure 24 and Figure 25 help visualize patterns associated with a typical solution to $\psi(2,4)^{34}$ and $\psi(2,5)^{177}$, respectively. For both figures, the solution is shown as a sequence of 0's and 1's representing an assignment of values to variables in increasing order of index, from left to right. Each curve shown is called a *solution curve* and is derived from the solution in the figure. A solution curve raises one unit for every 1 encountered and drops one unit for every 0 encountered when traversing the solution from left to right. Observe that the number of peaks in each figure is l-1 and, more importantly, there appears to be a limited length pattern of reverse symmetry, which I call a *reflected pattern*, in the vicinity of at least one of the peaks.

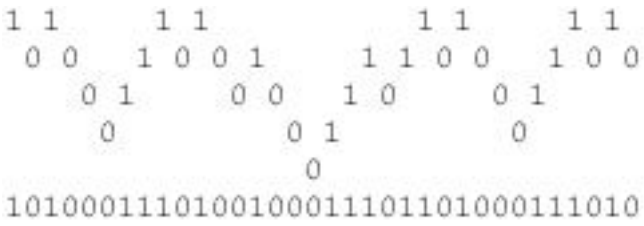

Figure 24. Typical solution curve for $\psi(2,4)^{34}$. This is the largest satisfiable formula for W(2,4).

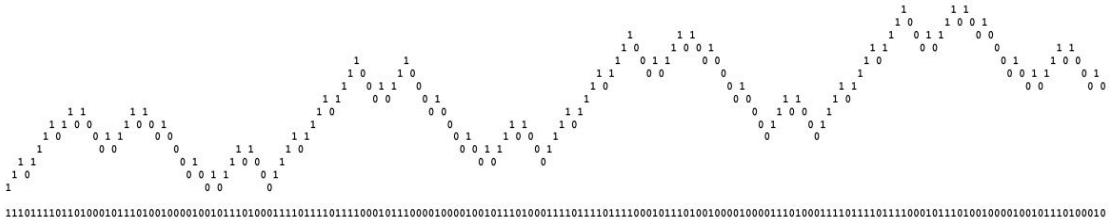

Figure 25. Typical solution curve for $\psi(2,5)^{177}$. This is the largest satisfiable formula for W(2,5).

I conjecture the following based on observations such as those depicted in Figure 24 and Figure 25:

Conjecture 1: For every $\psi(2,L)^{W(2,L)-1}$ there exists a solution that contains at least one reflected pattern of length W(2,L)/((L-1)*2) with the middle positioned somewhere between W(2,L)/(L-1) and W(2,L)*(L-2)/(L-1).

The tunnel is designed as a filter for consecutive variable assignment patterns that are not reverse symmetric. The tunnel consists of clauses involving s consecutive variables where, s is even and by Conjecture 1 and Observation 2, $s < \lfloor 1068/10 \rfloor = 106$. I tried several values for s including 60, 80, 100, even 150 and all worked, but speed increased significantly with increasing s up to 150.

The first family of tunneling constraints is shown in Figure 26. By using negative indices in Figure 22 these constraints remain fixed as n grows. Otherwise, the tunnel would have to move with n.

Tunnel Clauses	Subscript Range	Meaning
$\{v_{-i}, v_{i+1}\}, \{\bar{v}_{-i}, \bar{v}_{i+1}\}$	$0 \leq i < s/2$	force $v_{-i} \equiv \bar{v}_{i+1}$.

Figure 26. First tunnel constraints added to $\psi(2,6)^{n_0}$ initially, then retracted after "tunneling."

5.6.5 Second Tunnel

From the results obtained by using the first tunnel alone, it was observed that some small assignment patterns *did not* occur in solutions. The second tunnel filters those patterns. This action is opposite to that of *forcing* patterns to occur, which is the objective of the first tunnel. Consequently, the first tunnel spans a small number of variables because longer forced reverse symmetric patterns do not exist in any solution to $\psi(2,6)^{W(2,6)-1}$, but the second tunnel spans all variables because non-solution patterns can appear anywhere in the clauses of $\psi(2,6)^{W(2,6)-1}$.

The second family of tunneling constraints is shown in Figure 27. The maximum value of 20 for t is a compromise: the tunnel needs to be big enough to have an impact but small enough to keep some solutions around to the end. The number 20 was determined by experiment on $\psi(2,6)^n$ formulae.

Tunnel Clauses	Subscript Range	Filters
$\{v_i, \bar{v}_{i+t}, v_{i+2t}, \bar{v}_{i+3t}, v_{i+4t}, \bar{v}_{i+5t}\}$	$-n/2 < i \leq n/2 - 5t$	010101
$\{\bar{v}_i, v_{i+t}, \bar{v}_{i+2t}, v_{i+3t}, \bar{v}_{i+4t}, v_{i+5t}\}$	$1 \leq t \leq 20$	101010
$\{v_i, v_{i+t}, \bar{v}_{i+2t}, \bar{v}_{i+3t}, v_{i+4t}, \bar{v}_{i+5t}, \bar{v}_{i+6t}, v_{i+7t}\}$		00110110
$\{\bar{v}_i, \bar{v}_{i+t}, v_{i+2t}, v_{i+3t}, \bar{v}_{i+4t}, v_{i+5t}, v_{i+6t}, \bar{v}_{i+7t}\}$		11001001
$\{v_i, \bar{v}_{i+t}, \bar{v}_{i+2t}, v_{i+3t}, \bar{v}_{i+4t}, \bar{v}_{i+5t}, v_{i+6t}, v_{i+7t}\}$		01101100
$\{\bar{v}_i, v_{i+t}, v_{i+2t}, \bar{v}_{i+3t}, v_{i+4t}, v_{i+5t}, \bar{v}_{i+6t}, \bar{v}_{i+7t}\}$	$-n/2 < i \leq n/2 - 7t$	10010011
$\{v_i, v_{i+t}, \bar{v}_{i+2t}, \bar{v}_{i+3t}, \bar{v}_{i+4t}, v_{i+5t}, v_{i+6t}, \bar{v}_{i+7t}\}$	$1 \leq t \leq 20$	00111001
$\{\bar{v}_i, \bar{v}_{i+t}, v_{i+2t}, v_{i+3t}, v_{i+4t}, \bar{v}_{i+5t}, \bar{v}_{i+6t}, v_{i+7t}\}$		11000110
$\{\bar{v}_i, v_{i+t}, v_{i+2t}, \bar{v}_{i+3t}, \bar{v}_{i+4t}, \bar{v}_{i+5t}, v_{i+6t}, v_{i+7t}\}$		10011100
$\{v_i, \bar{v}_{i+t}, \bar{v}_{i+2t}, v_{i+3t}, v_{i+4t}, v_{i+5t}, \bar{v}_{i+6t}, \bar{v}_{i+7t}\}$		01100011

Figure 27. Second tunnel constraints added to $\psi(2,6)^{n_0}$ initially, then retracted after "tunneling."

5.6.6 Third Tunnel

In [73] solutions to $\psi(2,6)^n$ are found for various values of n including 565 and 695. The latter number implies the best known analytic lower bound on W(2,6). Regarding the solution to $\psi(2,6)^{565}$, re-index the assigned variables

$v_{-282}, \ldots, v_0, v_1, \ldots v_{282}$

to

$v_{-564}, v_{-562}, \ldots, v_0, v_2, \ldots, v_{562}, v_{564}$

and add unassigned variables

$v_{-565}, v_{-563}, \ldots, v_1, \ldots, v_{563}, v_{565}$.

Observe that this operation will not introduce any arithmetic progression among the even indexed variables. The assignment to the even indexed variables is the third tunnel. Search for a solution to $\psi(2,6)^{1131}$ among the variables above, keeping the values of the even indexed variables fixed. The search completes in a reasonable time with a solution.

Remarkably, using any of the three tunnels or even combining them leads to the same bound of 1132 for W(2,6).

5.6.7 Choosing the Search Heuristic

Variables are always considered in the following order, regardless of assignment:

$v_0, v_1, v_{-1}, v_2, v_{-2}, v_3, v_{-3}, \ldots$

The reason is that by assigning values to middle variables first, there is a good chance of inferences developing, symmetrically, for higher and lower indexed variables. If, say, the lower indexed variables were assigned first, perhaps half of the potential future inferences would not be realized early. This is particularly important for the first tunnel where the inferences are needed to appear outside of the tunnel variables.

5.6.8 Optimizations and Special Procedures

I was able to get the lower bound of 1132 for W(2,6) with either tunnel using the special solver described in Section 5.6.2 with the optimizations mentioned below. However, for stock SAT solvers I

could not get this result in reasonable time using either the first or second tunnels alone, but could get the result using both simultaneously.

The following optimizations to the special solver were used.

1. Data structures specifically designed for very fast checking of arithmetic progressions and inferences were used. Estimated speed up due to these structures is about a factor of 25. Another factor of approximately 2 speedup is due to item 3 below.
2. Without optimization, all nodes of the search space would have two children representing True and False assignments to the variable of that node. However, I generate two children only when inferences force that to be necessary. In other words, I implement a primitive form of conflict-analysis, but do not record the result as a clause as is done in modern SAT solvers. The estimated speed up due to this optimization is about a factor of 2.
3. The static search heuristic described in Section 5.6.7 accounts for an estimated speed up factor of 2.

5.6.9 Performance Results

On W(2,5) formulae, my solver with special data structures took 1 second with the first tunnel of 6 variable width on a 2 GHz Pentium 4 computer, and made 143184 variable assignments. Without a tunnel my solver with special data structures took 8 seconds on the same machine and made 719640 variable assignments. On the same machine zChaff (2004.11.15) took 702 seconds and made 570981 assignments and Berkmin (561) took greater than 1500 seconds without a tunnel (see the end of Section 5.6.2 for information about how zChaff and Berkmin were run). Upon adding the first tunnel of 6 variable width to the Van der Waerden formulae, zChaff (2004.11.15) took 25 seconds and made 93031 assignments. This factor of 30 improvement demonstrates the strength of the idea on the same SAT solver. On W(2,6) formulas zChaff ran out of memory (2GB) in a few hours and BerkMin took greater

than 24 hours and were not able to finish, hence this comparison could not be made, only estimated, from the W(2,5) results.

5.7 Lower bounds for K>2

Using a similar tunnel as described in the previous section I have obtained previously known bound W(3,4)>292. I have also applied the tunnels to W(5,3) but the tunnel was too restrictive and no valid coloring was extendable past 123 (compare with the current best bound of W(5,3)>125).

5.8 Preprocessing for the complete search

In the due course of my research I identified several ways to cut the search space. During the preprocessing some of the branches will not be searched because they are found to be equivalent with other branches. I will also show how some patterns are always present in a pattern particular length for a given K and L – I call this property *unavoidable* patterns. There is another property called *forbidden* patterns that has been mentioned in the previous section and which might be also useful. I am not taking advantage of this property during my search for W(2,6).

5.8.1 Preprocessing patterns

In the course of the search for the value of W(2,6) I would like to eliminate as much of a redundant search as possible. As a simple example, it is obvious that the search tree can stem from a variable v_i being assigned 0 while all other variables are unassigned. I do not have to explore the case where this variable i is assigned 1 because the search would explore the identical search space with 0 and 1 negated.

For 2-colorings as defined earlier I will consider color 0 to be identified as ‘0’ and color 1 to be identified as ‘1’. To explore the cutting the redundant branches further I define a few functions:

negation (*p*) returns the same pattern reversed colors. Example: *negation*('00001') = '11110'

reverse (*p*) returns the same pattern spelled backwards. Example *reverse*('00001') = '10000'

min (p_1,p_2) returns the minimal binary comparison of two equal length patterns. Example:

'0' = '0', '0' < '1', '0001' < '0101'

minimal(*p*) returns the minimal binary representation of a pattern p.

minimal(p) = min(*min(p*, reverse(*p*)), *min*(*negation*(*p*), *reverse*(*negation*(*p*)))

extend(*p, direction*) = returns two patterns that are the extension of the pattern *p* in the desired *direction*. For example extend(‘000’, left) = ‘0000’ and ‘1000’

When exploring the search space for n numbers, starting the variable assignment in the middle allows redundancies to be found soon at the price of only shrinking the value of n.

I will use *x* to identify unassigned variables:

Start with: $x^{50}\ 0\ x^{49}$ and extend on the right:

$$x^{50}\ 00\ x^{48}$$

$$x^{50}\ 01\ x^{48}$$

Extend on the right again:

$$x^{50}\ 000\ x^{47}$$

$$x^{50}\ 001\ x^{47}$$

$$x^{50}\ 010\ x^{47}$$

$$x^{50}\ 011\ x^{47}$$

Taking the second and fourth pattern and combine them together, I end up with the following two patterns:

$$x^{50}\ 001\ x^{47}$$

$$x^{47}\ 110\ x^{50}$$

Here you can see that the second pattern is a negation of the first one only in the different place in the original pattern length n. The result is a smaller pattern length 47*2+3 = 97:

$$x^{50}\ 000\ x^{47}$$

$$x^{47}\ 001\ x^{47}$$

$$x^{50}\ 010\ x^{47}$$

If I only consider patterns in their minimal form and eliminate redundancies the *n* will shrink by 2*(the number of extensions).

Now assume that there is a sufficient number of *x* on each side and focus only on the expanded patterns. For example, I know that any K>1 and L>1 will have a pattern '01' for any $n>n_0$, where $n_0 = L-1$. I have a choice to either expand this pattern on the left (a) or on the right (b).

(a) 01x or (b) x01

(a) 010, 011 and after recoloring 010, 001

(b) 001, 101 and after recoloring 001, 101

Notice that after recoloring the patterns are identical

Going one step further I have four options

x010	x010	010x	010x
x001	001x	x001	001x

Resulting (after recoloration):

0010	0010	0010	0010
0101	0101	0101	0101
0001	0010	0001	0010
0110	0011	0110	0011

In the next step, notice that if I extend 0010x and 0101x I get the following result (after recoloration):

00100

00101 the following two patterns are the same and one of them can be safely eliminated

00101

01010

In general, for every existing pattern I have a choice of extending it either on the left or right. The number of patterns to consider can be greatly reduced by a careful choice of the direction in which they are extended, since some of them extend to identical patterns.

Finding the optimal way to extend each pattern is a hard problem and looking for the best solution by enumerating all possibilities yielded results only for a few steps. The problem of finding the

minimum number of jobs is probably a hard problem, and I came up with a palindrome-based approximation discussed in the next section.

Figure 28 and Figure 29 show pattern expansion trees. Highlighted patterns in Figure 29 indicate which patterns have a choice of expansion on the left or on the right. Non-highlighted patterns are palindromes and yield the same patterns if expanded on the left or on the right.

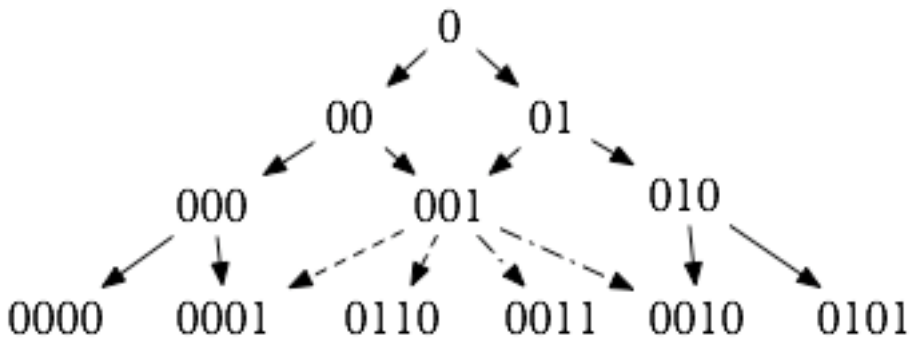

Figure 28. Tree of minimal patterns for 4 numbers

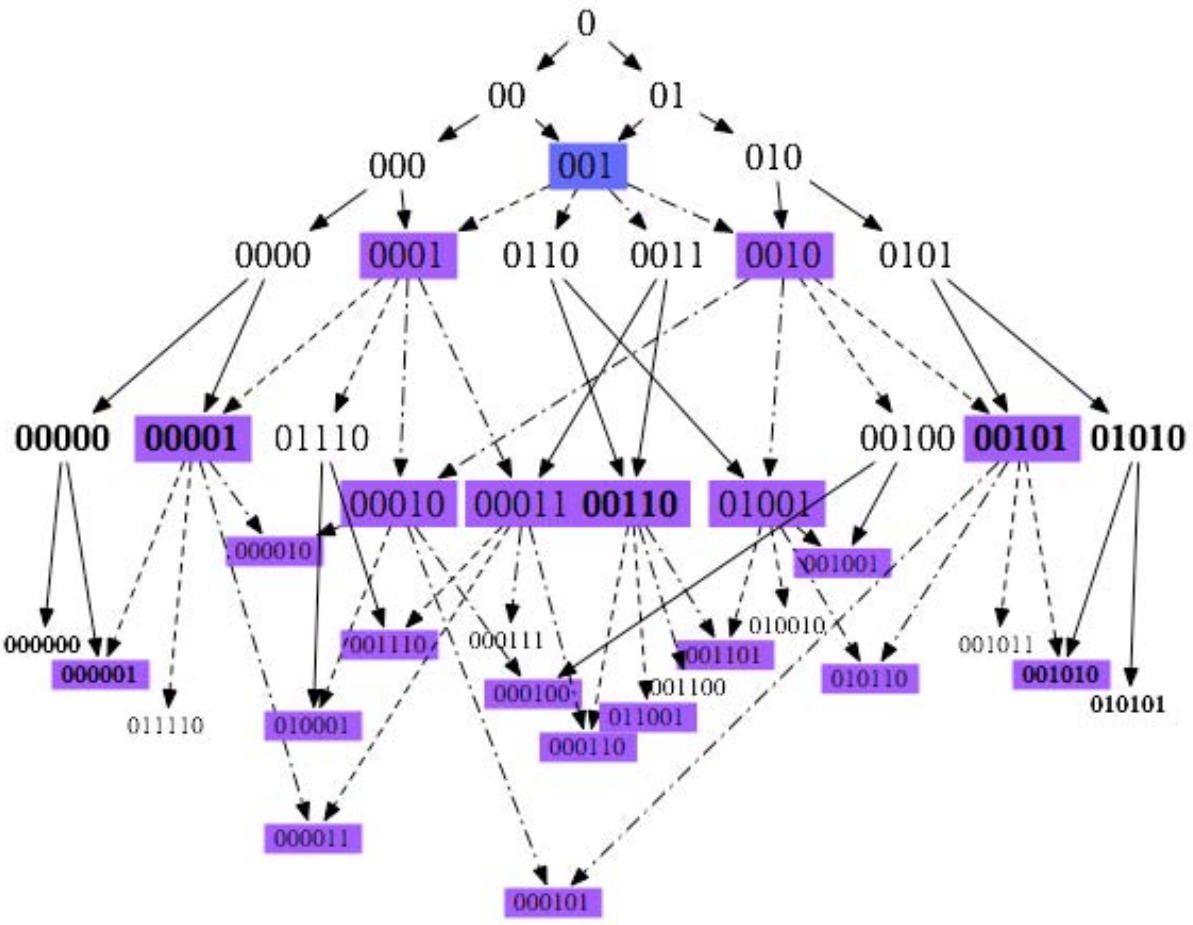

Figure 29. Tree of minimal patterns for 6 numbers

5.8.2 Palindrome symmetries

The best approximation to pattern extension I have seen so far is based on the obvious property of symmetrical (palindrome) patterns where an extension on each end of the pattern yields 3 new patterns instead of 4 patterns. Another advantage is that two of the newly formed patterns are again palindromes.

I recognize tree types of pattern palindromes:

a) positive palindrome (either odd or even n)

   Examples: 00100, 10101

b) negative palindrome (only even n)

Examples: 1100, 101010

Example of the expansion process:

```
X 010 x
0 010 0
1 010 1
0 010 1
1 010 0
```

Where last two are identical (only rotated) and one of them can be eliminated.

Another example:

```
x 0011 x
0 0011 1
1 0011 0
0 0011 0
1 0011 1 (=> 1 1100 1 -> 0 0011 0)
```

Again last two are identical – only reversed and negated.

The non-palindromes are extended so they reach palindrome in the shortest number of steps. For example 0001 is not a palindrome but if extended on the left it yields one (10001) and a non-palindrome (00001).

To recover the original search space I only have to shift the final set of patterns in the n numbers. Since I will use the resulting set and apply the SAT solver first to get over the hump (as defined in the section 5.4) the shifting is not computationally intensive compared to crossing the hump.

5.8.3 Unavoidable patterns

For some patterns (given K and L) it is easy to prove that they have to be present in any partition longer than some n_0. For example '01' has to be present for $n_0 > L-1$ (for K>1 and L>2). Also a pattern '001' has to be present for $n_0 > 2*(L-1)$ (for K>1 and L>2).

Please notice that from the pattern equivalence 001 is equivalent with 110, 100 and 011.

Finding n_0 for longer patterns is becoming increasingly difficult.

For some patterns *p* exist n such that every pattern longer than n that does not have a monochromatic arithmetic progression length L will contain minimal representation of pattern *p*.

Using the computer search I established n_0 for selected patterns. The longest pattern for W(2,6) took several weeks on a single computer. I used a modified SAT solver where the solver not only backtracked when found a monochromatic arithmetic progression but also when found one of the unavoidable patterns. The longest partition without an unavoidable pattern plus 1 indicates the length n_0 which has to contain the unavoidable pattern for a given K and L. This of course assumes that $n_0 <$ W(K,L). If the solver, given the above description, returned the length of the longest partition equal to W(K,L) it would mean that the pattern is not unavoidable and the extreme partition does not contain this pattern.

Table 13. n_0 for various unavoidable patterns.

K	L	Pattern	n_0
2	L	01/10	(L-1)
2	L	00	2(L-1)+1
2	4	00/11	7
2	4	000/111	21
2	5	00/11	9
2	5	000/111	61
2	5	0000/1111	85
2	5	00/11	11
2	5	000/111	190
2	5	0000/1111	240

5.8.4 Forbidden Patterns

Every pattern has n_{f0} but only those that have $n_{f0} < n$ are interesting.

For larger Ls patterns that are not part of the extreme partition typically have much smaller n_{f0}.

See the second tunnel for example of forbidden patterns.

5.8.5 Putting all preprocessing together for W(2,6)

The longest unavoidable pattern I proved for W(2,6) is 0000. This pattern can be even extended to ‘00001’ without the loss of generality. Growing this pattern when the lower bound is 1131 gives us plenty of space for a number of iterations of growing the patterns. I grow the initial pattern while the

number of patterns is manageable stopping after 12th iteration when most of the patterns are 28 lengths long and total count 2,537,546 patterns (see Table 14).

1. ‘00001’
2. ‘100001’, ‘000001’
3. ‘01000010’, ‘11000011’,’01000011’, ‘1000001’
4. …

Table 14. Counts of patterns during the palindrome based pattern growing

Iteration	Palindromes	Non Palindromes	Lengths	Total Count
0	0	1	5	1
1	2	0	7	2
2	4	3	9	7
3	8	18	11	26
4	14	78	13	92
5	28	319	15	347
6	46	1252	17	1298
7	97	4581	19	4678
8	176	17034	21	17210
9	363	62092	23	62455
10	722	224113	25	224835
11	1218	763984	27	765202
12	2429	2535117	29	2537546

There are total of 77948442 patterns length 29 without a monochromatic arithmetic progression of length 6 when one coloring is forced to 0 (so double the number to include even all the negated patterns).

11403360 of the original 77948442 patterns do NOT have the initial patterns of 0000/1111.

The result of the palindrome based preprocessing with the initial pattern of 00001 is 1248 patterns of the length 28 and 2536298 patterns of the length 29 to the total of 2537546 patterns (the difference in lengths is due to even and odd palindromes).

These considerations show that only 3.26% of the original search space needs to be explored in order to still cover the entire original search space.

5.9 SAT Solver for Van der Waerden Numbers

5.9.1 Pseudocode for the Preprocessor

Internal representation of the patterns in the preprocessor has the following format. Since the negations are treated the same way as original patterns I can simplify the format to count the number of equally colored positions between transitions between 0/1 and 1/0. For example, a string 323 would identify 00011000 as well as 11100111 without loss of generality.

The expansion/growing of the pattern on the left is *a* simple addition of 1 on the left or increasing the left-most number by one. The same holds to the expansion/growing on the right.

This format was chose for its efficient storage of the patterns although checking for progressions is somewhat cumbersome compared to regular pattern array.

Finding the minimal binary representation is also simple as it amounts to the smallest of the strings in the string-wise comparison.

The preprocessor performs the expansion and then sorts the patterns to weed out the duplicities. This approach scales over 30 steps when it ends up with millions of patterns and almost exhausted 2GB of RAM.

Internally two arrays are allocated – one for palindromes and another for non-palindromes since both groups are expanded in different way. Palindromes are expanded on both sides – two expansions in a single step and one of the resulting non-palindromes is eliminated. Non-palindromes are expanded on the side closest to creating a palindrome. This creates an approximation for the smallest set after *i* steps, and I believe that it is not optimal but it is the best I can do.

5.9.2 Pseudocode for the Solver

```
push('0')
while(!stack_empty())
begin
   path=pop()
   if last position is NOT part of progression then
       push(path + '0')
       push(path + '1')
   endif
end while
```

Figure 30. Solver pseudocode version 1

```
path is stored as a string with the set position as set_pos and the last heuristically chosen
variable as cp_pos


set path.set_pos=0
set path.cp_pos=0
set path to '0' at path.set_pos
push(path)
while(!stack_empty())
begin
   path=pop() // path.set_pos, path.cp_pos
   if path.set_pos does not create a contradition then
       path.cp_pos++
       while path.cp_pos position is set do
            path.cp_pos++
       end while
       path.set_pos=path.cp_pos
       set path to '0' at path.set_pos
       push(path)
       set path to '1' at path.set_pos
       push(path)
   endif
end while
```

Figure 31. Solver pseudocode version 2

```
path is stored as a string with the set position as set_pos and the last heuristically chosen
variable as cp_pos
for contradiction and inference checking it is beneficial to save min and max position that has
been set


set path.set_pos=middle of path
set path.cp_pos=0
set path to '0' at path.set_pos
push(path)
while(!stack_empty())
begin
   path=pop() // path.set_pos, path.cp_pos
   if path.set_pos does not create a contradition then
       path.cp_pos++
       while heuristic_mapping(path.cp_pos) position is set do
            path.cp_pos++
       end while
       path.set_pos=heuristic_mapping(path.cp_pos)
       set path to '0' at path.set_pos
       push(path)
       set path to '1' at path.set_pos
       push(path)
   endif
end while
```

Figure 32. Solver pseudocode version 3

```
path is stored as a string with the set position as set_pos and the last heuristically chosen
variable as cp_pos
for contradiction and inference checking it is beneficial to save min and max position that has
been set


set path.set_pos=0
set path.cp_pos=0
set path to '0' at path.set_pos
push(path)
while(!stack_empty())
begin
   path=pop() // path.set_pos, path.cp_pos

   if position path.set_pos has potential for progression (due to dots) then
        pos=last dot
        set save_path to path
        set path to '0' at pos
        if check for contradictions with inferences in path at pos then push(path)
        set path to save_path
        set path to '1' at pos
        if check for contradictions with inferences in path at pos then push(path)
   else
        path.cp_pos++
        path.set_pos=heuristic_mapping(path.cp_pos)
        if path.cp_pos position is NOT set then  set path to '.' at path.set_pos
        push(path)
   endif

end while
```

Figure 33. Solver pseudocode version 4

5.9.3 Performance results

The solver can prove W(2,5)=178 in 2.613s, provide a partition of length 177 for W(2,5) in 0.091s and provide all satisfiable assignments for W(2,7) for n=177 in 2.683s on Intel Pentium 4/3GHz.

For W(2,6) the solver given an initial pattern and has to determine whether this pattern has a solution in W(2,6) for n=240.

I have available the clusters in the following configurations:

- 66x AMD Opteron cores running 1800MHz
- 34x AMD Athlon running 1533MHz
- 64x Intel PIII running 450MHz
- around 50 other Intel and Sun Sparc based processors no exceeding 500MHz each

It is my estimate that it would take more than a year to complete the search of all ~2.5mil preprocessed patterns.

In fact, after 6 months I completed about 45% of the patterns. The remaining 55% is being completed with the help of FPGAs described in the next chapter, and it is estimated that this portion will take less than 2 months, with the help of FPGAs.

5.10 Van der Waerden Sub Numbers

So far I have only considered Van der Waerden numbers where the length of the progression was limited equally for all colors. I call Van der Waerden sub number $W(K;L_1,L_2,..,L_k)=n$, the smallest integer with the property such that whenever integers 1,..,n are K-colored, there always exists monochromatic arithmetic progression in at least one of the color of length L_k. There is a number of exact Van der Waerden sub numbers known as illustrated by Table 15, Table 16, Table 17 and Table 18. Number in bold are those that I contributed compared to [74].

Table 15. Sub numbers for W(2; x,x)

W(2; x,x)

	2	3	4	5	6
2	3				
3	6	9			
4	7	18	35		
5	10	22	55	178	
6	11	32	73	**206**	*1132*
7	14	46	109		
8	15	58	**146**		
9	18	77			
10	19	97			
11	22	114			
12	23	135			
13	26	160			
14	27	186			
15	30	218			
16	31	**238**			
17	34				
18	35				
19	38				
20	39				

Table 16. Sub numbers for W(3; 2,x,x) and W(3; 3,x,x)

W(3; 2,x,x)

	2	3	4
2	4		
3	7	14	
4	11	21	40
5	15	32	71
6	16	40	83
7	21	55	**119**
8	22	72	

W(3; 3,x,x)

	3	4
3	27	
4	51	89
5	80	
6		
7		
8		

Table 17. Sub numbers for W(4; 2,2,x,x) and W(4; 2,3,x,x)

W(4; 2,2,x,x)

	2	3	4
2	5		
3	8	17	
4	12	25	53
5	20	43	75
6	21	48	
7	28	65	
8	29		

W(4; 2,3,x,x)

	3	4
3	40	
4	60	
5		
6		
7		
8		

Table 18. Sub numbers for W(4; 3,3,x,x)

W(4; 3,3,x,x)

	3
3	76

5.11 Summary

I have shown how an analysis of performance curves and solution patterns of a class of CNF formulae can present insight for designing effective tunnels through a search depth of high breadth. I have chosen to experiment with formulae for finding Van der Waerden numbers since they are a difficult class for standard SAT solvers, apparently because clause recording (learning) is ineffective. By tunneling, reasonable yet uninferred constraints are added just long enough to get to a search depth that has relatively low breadth. Then the tunnel is retracted with the hope that not all solutions have been destroyed by the tunnel. Doing so, I found a solution to $\psi(2,6)^{1131}$ and therefore a new, significantly improved lower bound on the number W(2,6) of 1132.

The symmetric nature of the formulae and solution played an important part in designing effective tunnels and three tunnels were tried with the same result. This type of symmetry is common in many formula classes that arise from practical applications including problems of formal verification. I believe other difficult problems will succumb to tunneling.

I believe performance curves of search breadth vs. depth, such as shown in Figure 23, provide a "fingerprint" for tunnel effectiveness of problem classes. I speculate the fingerprint will not change much qualitatively from one solver to the next, but cannot rule out that possibility. Assuming this, I make some claims about the performance curve with respect to hardness as follows. If a performance curve should reach a peak at very high depth, tunneling is unlikely to be effective. The family of *queue* formulae from bounded model checking seem to fall into this category. I speculate that early, large peaks mean delayed inferences and force exhaustive exploration at search depths corresponding to many unassigned variables.

Although the development of general purpose propositional solvers that work well on all inputs is strongly desirable, at this point it is hard to imagine how this is going to be accomplished. However, a general purpose solver can be assisted greatly by giving special consideration to an input problem class, mining its structure for some property that may be used to improve search. This is what I have done with tunneling. Although I do not foresee generally applicable principals for developing tunnels, I do believe that normal human intuition is enough to uncover exploitable structure in many cases. The Van der Waerden numbers illustrate both points.

Chapter 6 - A FPGA-Based Van der Waerden Solver

SAT solvers are finding increasing applicability in various fields such as combinatorics, model checking, hardware verification, and so forth. Increasing the speed of SAT solvers has a direct effect on advancements in those fields. I describe how I have used FPGAs (Field Programmable Gate Arrays) to implement a special SAT solver dedicated to the problem of determining new lower bounds or exact values for certain Van der Waerden numbers.

Previously I have used a special SAT solver to show that $W(2,6) \geq 1132$. This result roughly doubled the previously known lower bound for this number. I am currently attempting to show that $W(2,6) = 1132$ by using FPGAs to speed up my SAT solver to complete an exhaustive search of the space of 2-colorings of {1, 2, … 1132}. Using various equivalences (complementation and reversal) to bound the search, the solver takes an assignment (2-coloring) for m initial variables (integers) and tries to

determine whether there is an assignment to the remaining variables such that the formula is satisfied – in my case such that there is no monochromatic arithmetic progression of length 6. The solver will backtrack whenever it encounters an assignment having arithmetic progression of length 6.

The process of creating the FPGA downloadable design involves the following major steps. The first step is to create the design using a tool such as VHDL [75, 76] or verilog languages. The next step involves synthesizing the source files (comparable to compilation of source code to object files). The resulting intermediate form is then placed and routed, thereby producing a bit file that can be uploaded to the FPGA.

FPGAs retain the design while powered up, but when reset the design is lost and has to be uploaded again. Some boards have PROM memory attached to FPGA, which “boots” the FPGA with the initial design.

The primary motivation for using FPGAs for my Van der Waerden number search came from the following observation. During my profiling of the PC-based SAT solver it was clear that 90% of time the solver spent checking whether a newly set color did not form an arithmetic progression, or whether the newly set color did not infer a coloring on another number. These operations can be translated as checking a member of a set for some property (in my case an arithmetic progression). Such a property can be implemented on FPGA to take one or a limited number of cycles. One gate with six inputs and one output can signal whether the inputs form a progression of length six. The PC based program would take 6 comparisons to find this information.

Obviously this approach has a number of concerns. One concern is the space (number of LUTs on the FPGA) it takes to accommodate the entire problem of checking for progression on a set of 240 numbers. Another concern might be the clock speed of my design and achieved speed up compared to the PC-based solver. The price effectiveness of the reconfigurable hardware versus an off-the-shelf PC must also be taken into account.

6.1 Architecture

As mentioned earlier, my solver is searching the search space of 2-colorings of {1,2, …, n} for a coloring not containing a monochromatic arithmetic progression of a certain length. Each number has a color represented using two registers. At the start of the computation both registers are assigned 0 for all numbers except those that are part of the initial assignment. An assignment of either registers with the value 1 constitutes a coloring of that number with either color. Assigning both registers the value 1 is not a possible state.

All registers of one color for all numbers form a vector (C_0 and C_1).

My design consists of the following major blocks:

IB – Inference Block

CD – Contradiction Detection Block

CP – Choice Point Selection Block

LB – Logic block

RAM – Backtrack Memory

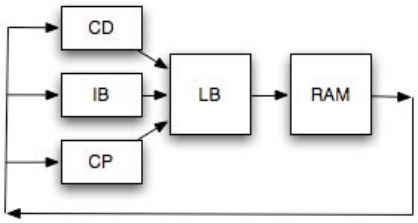

Figure 34. The FPGA solver block diagram

The CD (Contradiction Detection Block) has an input of a color vector and the output is True or False depending on whether or not the input vector contains an arithmetic progression.

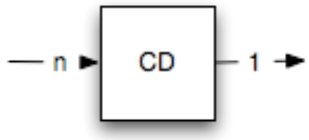

Figure 35. Contradiction Detection Block

For example if the input is “01111110001” the output will be 1 (True) since positions 2 to 7 contain an arithmetic progression of length 6.

The IB (Inference Block) has as input a color vector and as output a list of inferences – list of colorations forced to happen in the other color.

Figure 36. Inference Block

For example if the input vector is “111101” it is obvious that I cannot color 5^{th} position using the same color and therefore the opposite color is inferred. The output of the IB is “000010”.

The CP (ChoicePoint Selection Block) has as input both color vectors and an output of length n. The output will only have one bit set which corresponds to the next unassigned number. My solver uses a simple selection of the next choicepoint by selecting the numbers to color starting from the middle working its way out. The CP selects the next unassigned number that is closest to the middle (n/2).

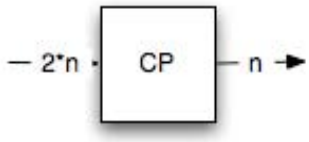

Figure 37. The Choice Point Selection Block

The LB (Logic Block) takes the output of all three previously described blocks and does the following. If the assignment contains a contradiction (as would be indicated by CD block) backtracking will be triggered and the previously saved assignment will be retrieved from RAM to be expanded. If there is no assignment in memory the computation is done and no assignment without an arithmetic progression is possible. If the assignment does not contain contradiction but a new coloring was inferred by the IB block, then the inference is made and set as the new input to all three previously described blocks to check for contradictions and possible additional inferences. Finally, if the assignment contains neither a contradiction nor new inferences, a new choice point is selected. If all numbers are assigned a color, then the solution is found. Otherwise the selected number, colored using the first color, is the current new assignment and the selected number, colored using the second color, is saved into RAM for future backtracks.

This design simplifies my fastest SAT solver version by removing the delayed literal evaluation logic while still maintaining excellent speedup over the sequential version and taking the advantage of the fine grain parallelism in the FPGA.

6.2 Equivalence Check

The solver consists of about 1000 lines of VHDL code. It is fully pipelined and parameterized for n, L and the number of stages in the CD, CP and IB blocks. In addition to running the solver and comparing the output with my PC-based solver, I verified the equivalence of some parts of the VHDL

with the cryptol code. In other words the cryptol code presented later was found to be equivalent to my VHDL code even though it is pipelined.

6.2.1 Equivalence check process example

The following is a description with examples of how the equivalence check was performed. For the illustration a simple VHDL and cryptol code is used.

1. Create a VHDL specification

For example I will assume a function that takes 3 inputs and one out where output is ‘1’ when all inputs are ‘1’ and ‘0’ otherwise.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

Entity example is
port (clk: in std_logic;
      i : in  std_logic_vector(3-1 downto 0);
      o : out std_logic);
end example;

architecture behavioral of example is
begin
    o <= i(0) and i(1) and i(2);
end behavioral;
```

Figure 38. Source of example.vhdl

Note: For the purpose of the equivalence check the first port has to be the input clock

2. Create a Cryptol specification

The same function as in the first step, but now written in the cryptol language.

```
f:([3][1]) -> [1];


f(x) = x@0 & x@1 & x@2;
```

Figure 39. Source code of f.cry

3. Create a Makefile and compile the VHDL specification

This step assumes that Xilinx ISE tools are available.

```
example:
        xflow \
        -p XC4VLX60-10FF668 \
        -synth xst_vhdl.opt \
        -implement fast_runtime.opt \
        example
```

Figure 40. Makefile for the VHDL compilation

Type “make example”

4. Create verilog netlist (.v) from the ncd file

The Xilinx tools include the netgen utility, which allows the generation of a verilog netlist from a compiled design. The interaction with the utility is shown in Figure 41.

5. Create a formal model (.fm) from the verilog netlist

The cryptol suite contains a program for the generation of the cryptol formal model files from the verilog netlist. Since the VHDL example file is not pipelined I can omit the output latency specification and leave it 1 as default. For pipelined designs I would use “-l <num_stages>” as another option for the *symbolic_netlist* command (see Figure 42).

6. Perform an equivalence check for the cryptol function and the verilog netlist formal model (see Figure 43).

```
[mkouril@beowulf bin]$ netgen -ecn formality -w example.ncd
Release 8.2.02i - netgen I.33
Copyright (c) 1995-2006 Xilinx, Inc. All rights reserved.

Command Line: netgen -ecn formality -w cd.ncd

WARNING:NetListWriters:629 - The -ngm switch was not used in the netgen command
   line. This will prevent netgen from being able to create an assertion file
   for the equivalency checking tool.
Read and Annotate design 'example.ncd' ...
Flattening design ...
Processing design ...
  Preping design's networks ...
  Preping design's macros ...
Writing Verilog netlist file 'example_ecn.v' ...
Number of warnings: 1
Total memory usage is 191784 kilobytes

Created netgen log file 'example_ecn.nlf'.
```

Figure 41. Interation with netgen utility to create verilog net list

```
 [mkouril@beowulf bin]$ ./symbolic_netlist -s example_ecn.v
```

Figure 42. Interaction with symbolic_netlist to generate formal model file

```
 [mkouril@beowulf bin]$ ./cryptol_fpga f.cry
Cryptol FPGA version 0.8, Copyright (c) 2001-2006 Galois Connections Inc.
A beta release.                                          www.cryptol.net

Type :? for help
Loading "f.cry".. Checking types.. Processing.. Done!
cd> :equals f "example_ecn.v.fm"
EQUIVALENT
cd>
```

Figure 43. Interaction with cryptol program during the equivalence check

6.2.2 Results of Equivalence Checks

I wrote cryptol files for three major blocks of my design, namely, the CD, CP and IB. They are shown in Figure 44, Figure 45 and Figure 46, respectively.

For the CD block the function cd(x) returns 0 if the output of clause(x,start,step) is True for any start and step between 1 and n-1. The function clause(x, start, step) returns True if the clause starting with *start* and having step *step* forms an arithmetic progression length L

```
n=240;
L=6;

clause: ([n*2+1][1],[width n],[width n]) -> Bit;
clause(x,start,step) =
      if ((n-start-1)/step>=(L-1))
      then [| x@(start+(p-1)*step) || p<-[1..L] |]==[| 1 || p<-[1..L] |]
      else False;

cd: ([n][1]) -> [1];
cd(x) =
  if
    [|
        if clause(x#[| 0 || p<-[1..n+1] |], start-1, step) then 1 else 0
     || start<-[1..n], step<-[1..n]
      |]  == [| 0 || p<-[1..((n-1)*(n-1))] |]
  then 0 else 1;
```

Figure 44. Source for cd.cry

The function cp(c0,c1) returns the result of the function done(c0,c1), which is 1 if there is no uncolored number, 0 otherwise. The function cp also returns a vector c in which one bit is set indicating the next choice point. The numbers are chosen from the center and then work their way out. The function map(x) remaps the input vector to place the highest priority selection to the lowest bit of the resulting vector and vice versa. The function lowest(x) finds the lowest set bit and leaves it as the only set bit. If no bit is set in the input vector the output of lowest is all zeros. The function unmap is the reverse function of map(), and it remaps the bits such that the lowest bit is placed in the middle, second lowest is placed next to the middle bit, and so on up to the highest bit which is placed at position 0 or n-1 in the output vector.

```
n=240;


done: ( [n][1], [n][1] ) -> [1];
done(c0, c1) = if ~(c0 | c1) == [| 0 || p<-[1..n] |] then 1 else 0;

map: ( [n][1] ) -> [n][1];
map(c) = c@@ [| if (index%2)==1 then n-1-(index/2) else (index/2) || index<-reverse [0 .. n-1]
|];

unmap: ( [n][1] ) -> [n][1];
unmap(c) = c @@ ( [| index || index <- reverse [ 1 3 .. n-1 ] |] # [| index || index <- [ 0 2 ..
n-1 ] |]);

is_zero: ( [n][1], [ width n ] ) -> Bit;
is_zero(c, len) = [| if (p<len) then c@p else 0 || p<-[0..n-1] |] == [| 0 || x<-[ 0 .. n-1] |];

lowest: ( [n][1] ) -> [n][1];
lowest(c) =
[ (if (c@0 == 1) then 1 else 0 ) ]
#
[| if (is_zero(c, p) & (c @ p == 1)) then 1 else 0 || p <- [1 .. n-1] |]
;

give_me_lowest: ( [n][1], [n][1] ) -> [n][1];
give_me_lowest(c0, c1) = unmap(lowest(map(~(c0 | c1))));

cp: ( [n][1], [n][1] ) -> [n+1][1];
cp (c0, c1) =  [ (done(c0,c1)) ] # give_me_lowest(c0,c1) ;
```

Figure 45. Source for cp.cry

```
n=20;
L=6;

index_arr: [ width n] -> [L-1][ width n];
index_arr(id) = [| if (i-1)<id then i-1 else i || i<-[2..L] |];

/* i is 0 based, id is 1 based */
infer_lit: ([n*2+1][1],[width n],[width n],[width n]) -> Bit;
infer_lit(x,i,step,id) =
  if   (i/step >= (id-1))   /* enough number before i/step >= (id-1) */
     & ((n-1-i)/step>=(L-id+1-1))
  then
  if (id==1) then
      ( [| x@(i+p*step) || p<-[1..L-1] |]==[| 1 || p<-[1..L-1] |] )
  else if (id==L) then
      ([| x@(i-p*step) || p<-[1..L-1] |]==[| 1 || p<-[1..L-1] |] )
  else ([| x@(i-(id-1)*step+(p-1)*step) || p<-index_arr(id) |]==[| 1 || p<-[1..L-1] |] )
else
  False;

infer_one: ([n][1], [ width n ]) -> [1];
infer_one(x,i)=if [| (if (infer_lit(x # [| 0 || p<-[1..n+1]|] ,i,step,index_L))  then 1 else 0)
|| step<-[1..n],index_L<-[1..L] |] == [| 0 || p<-[1..n*L] |] then 0 else 1;

infer: ([n][1]) -> [n][1];
infer(x) = [| infer_one(x,i) || i<-[0..n-1] |];

any_infer: ([n][1],[n][1]) -> [1];
any_infer(c1, infer_v) = if (infer_v& (~c1))==[| 0 || p<-[1..n] |] then 0 else 1;

ib: ([n][1],[n][1]) -> [n+1][1];
ib(c0,c1) = [  ( any_infer(c1,infer_v) ) ]  #  (infer_v|c1)  where { infer_v:[n][1]; infer_v =
infer(c0); };
```

Figure 46. Source for ib.cry

Finally, the function ib(c0, c1) returns a tuple consisting of a bit and an output vector, where the first bit indicates whether there is a new inference, i.e., whether the output vector is any different than c1 on the input vector. The output vector consists of an inferred coloring from c0 OR c1. The function infer(c0) returns a vector of all bits inferred by c0. The function infer_one(x,i) returns 1 if bit *i* in x should be inferred, 0 otherwise. The function infer_one calls infer_lit(x,i,step, index_L) for all possible combinations of step and index_L, which is the bit placement in the possible progression (between 1 and L). The function infer_lit first decides whether such clause is possible, and then returns 1 if the remaining bits in the progression are set to '1', i.e., if the current position is inferred (should be forced to be set to a different color to prevent formation of an arithmetic progression).

I was able to equivalence check the following modules.

Table 19. Equivalence check results

	Result
CD	Checked in full (at n=240, L=6 with 8 stage pipeline)
CP	Checked in full (at n=240 with 8 stage pipeline)
IB	Checked in full (at n=240, L=6 with 8 stage pipeline)

The remaining part of the code is relatively trivial, and consists of LB, RAM and the framework VDW_SOLVER.

6.3 Evaluation

To address the limitation concerns discussed in the introduction, I synthesized the initial design. After a number of optimizations it was clear that the design would not fit into available Xilinx Spartan devices and Virtex-4[77, 78] would be needed. The product offering Virtex-4 starts with just a few thousand LUTs and goes up to hundreds of thousands of LUTs. The ideal fit in terms of chip size and the availability of evaluation boards led to a Virtex-4 LX60 with almost 60K LUTs.

Also I developed the simulation equivalent C-code to evaluate the speed of the final design with respect to the PC-based solver. The simulation yielded the speedup of 1.1x 1MHz, which indicated great potential for the FPGAs. Virtex-4 maximum (unrealistic) speed is cited at 450MHz. The realistic speed for available evaluation boards is 100MHz, yielding 110x speedup for one FPGA, which matches the speed of all Linc Lab's currently working clusters.

The design was evaluated on four AVNET ADS-XLX-V4LX-EVL60[79] evaluation boards with Xilinx Virtex-4 FPGA. Specifically the boards have XC4VLX60-10FF668 with almost 60 thousands LUTs. The board was selected for the chip's large number of available logic to fit my design, and also for its excellent price/performance ratio.

I used Xilinx tools ISE 8.1i and 8.2i to synthesize and place and route the design as well as Xilinx EDK 8.1 to integrate the design with the embedded MicroBlaze soft core processor described later. The Xilinx tools were run on Linux based server with two 3GHz Pentium 4 processors and 2GB RAM.

The simulation and debugging was mainly done using Cadence tools. The simulation is relatively slow and only relatively small designs were simulated. The full design could not be simulated for any reasonable initial paths for the time it would take to return a result.

Initial one-stage (no pipeline) designs were clocked at 25-50MHz, and even then confirmed the simulation prediction of about 1.1x1MHz speedup compared to the sequential SAT solver, achieving 55x speed up over 1800MHz Opteron.

The depth of the logic limits the speed of the one-stage design. The pipelining of my design proved to be relatively simple. Each of the major blocks (IB, CD and CP) was pipelined into a number of stages, while the LB and RAM were split into one stage each. The LB block had to be altered to not only end the computation when one stage found a contradiction while the RAM is empty, but also when there was no other working stage.

The pipelined architecture allows us to achieve speeds of up to 240MHz, or 260x of the sequential SAT solver running on a sequential processor. The test of the synthesis and place and route on the latest Xilinx chip Virtex-5 showed even greater potential due to the increased number of inputs to 6 on LUTs.

On the Virtex-4 evaluation board the fastest frequency I can achieve is 210MHz due to the limitation of the DCM (Digital Clock Manager) component. This embedded component is responsible for generating the desired clock frequency from the input clock, which, runs at 100MHz.

6.4 Software Interface

In order to interface the evaluation FPGA with a host computer, make the initial assignment, and retrieve additional information about the computation (such as the value of the internal cycle, choicepoint and solution counters), I decided to use the embedded serial port. This choice was primarily driven by the simplicity of the communication over the serial port. My solver does not have great bandwidth requirements and the serial port, albeit slow, does not create a bottleneck in the computation.

In addition to using the serial port I explored the Xilinx EDK (Embedded Development Kit) and found the supplied MicroBlaze processor core to fit my needs. MicroBlaze is a 32-bit soft core processor with user-friendly tools supplied in the EDK such as gcc compiler suite and debug tools for onchip debugging. I was able to quickly design OPB (Onchip Peripherial Bus) interface for my solver and connect it with the instantiated MicroBlaze. A relatively small piece of code for the MicroBlaze allowed us to control the solver from a host computer.

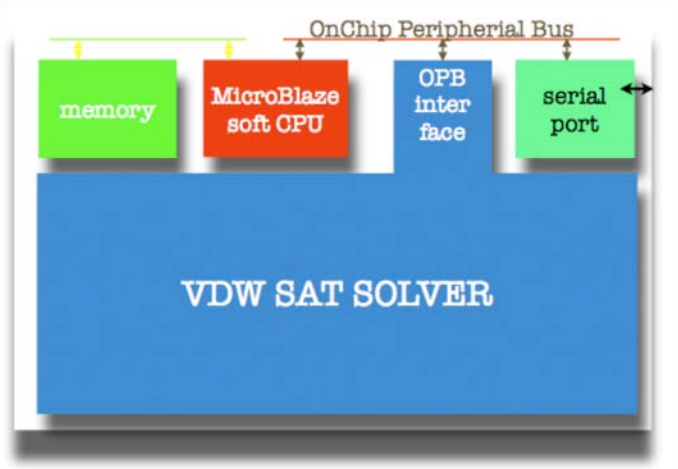

Figure 47. Overall architecture of the special FPGA SAT solver

The MicroBlaze core, however, also limits the speed of the design. Although the solver is capable running at 200MHz, the MicroBlaze core at its minimal configuration is not able to exceed 133MHz on this particular FPGA chip. Therefore, in the end my design is only clocked at 100MHz achieving the speed up of about 110x over the single Opteron core.

6.5 Integration with ICI Based Computation

In order to integrate FPGA based computation with the ongoing cluster based computation I connected all four available evaluation boards to the Cyclades terminal server card in a cluster node. A simple program in a shell script retrieves the next initial configuration to work on from the ICI work server and passes it on to the FPGA board over the serial port. When the FPGA board is done the shell script indicates back to the ICI work server that the FPGA completed the task and hands over the results.

6.6 Summary

I designed the FPGA-based SAT solver that significantly increases the speed of the searching compared to the sequential-based SAT solver. The FPGA-based SAT solver has a number of limitations.

One of these limitations is the space (number of LUTs on FPGA) it takes to accommodate the entire problem of checking for progression on a set of 240 numbers. Another limitation might be the clock speed of my design and achieved speed up compared to the PC-based solver. Another concern is the price effectiveness of the reconfigurable hardware vs. an off-the-shelf PC.

The design utilizes up to 72% of the LUTs and 84% of the slices. The MicroBlaze core occupies about 5-10% of this space. Although this might not seem like a fully utilized design, it is necessary to keep in mind that 100% utilized design is increasingly hard to route and might not be feasible.

I also evaluated architecture with a PicoBlaze 8-bit core, and even one without an embedded soft core. In both cases the speed increased significantly reaching about 133MHz for PicoBlaze core and 210MHz for core-less architecture with just a simple interface to the serial port.

When clocking the design at the higher speeds the heat dissipation becomes a serious problem. The evaluation board that comes with the FPGA chips has no heatsinks whatsoever. Although I have not encounter any issues when the design was clocked at 100MHz, the chips ran rather hot. A simple added heatsink together with a fan placed above the evaluation board solved the problem. The Virtex-4 has a built-in temperature sensor so in theory it should shut down when there is a danger of overheating. Unfortunately the sensor readings became unavailable starting Xilinx ISE 6.3 and no other way was found to monitor the temperature of the FPGA chip.

I showed that FPGAs can have a great impact in the field of SAT solvers. I achieved a good speedup with a relatively cheap ($700) board. The downsides I encountered are primarily connected with the development cost and development time as well as the constraint in the fabric space. The combined time for the synthesis and place and route ranged from 2-3 sometimes reaching the extremes of 6 hours on 3GHz Pentium 4.

Chapter 7 - Conclusions

Scientific computing today is headed toward expanded utilization of multi-processor clusters. The price of hardware is steadily falling while at the same time processor speed is steadily increasing. These trends place greater burdens on efficiency of software management and make more demands on the efficient utilization of existing hardware resources. Some problems require more processing power than clusters of reasonable size can provide, and newer technologies must be employed. Fortunately, such technologies have become available and at competitive price-performance ratios. One such technology, namely, reconfigurable computing as supported by FPGAs, has emerged that can be used in combination with clusters or in a standalone fashion. In this way even more computational power can be delivered with lower energy and environmental costs.

In this dissertation I presented a number of solutions for employing a single cluster, multiple clusters, or even other devices into long-term computations. I addressed an important problem of connecting clusters running diverse software and hardware through the InterCluster Interface (ICI)

library. Although this library does not address issues such as aggregate operations and group communication, its point-to-point interoperable aspect makes it appealing for projects where the computation is not limited to one homogenous cluster, but can be performed by multiple computers or clusters.

My Backtracking Framework (BkFr) not only enables communication between heterogeneous clusters, it also supports load balancing and uniform distribution of the problem over all participating resources. Although the framework is not limited to the backtracking paradigm, much of the development of the framework was connected with this paradigm, and most of the modules make use of backtracking. For intracluster communication (communication within the cluster) the standard MPI library is used. As noted earlier BkFr also takes advantage of ICI and can span the execution over multiple clusters with the uniform load balance and failure recovery using the Path Repository component. A cluster can come in and out of computation, as it becomes available taking advantage of cluster that might be available only for a limited time.

The research involving BkFr not only supports efficient utilization of computing resources, but also facilitates the introduction of MPI to the CS student using a template library. The template library helps the student get started in the somewhat confusing environment of cluster computing, and supplies the basic functionality of MPI. Students can then focus on the algorithm and its implementation, while filling in blanks based on learned properties of a very basic set of MPI functions. In the end student can quickly get the MPI code running on a cluster.

One of the major contributions of this thesis is the work on the problem of Van der Waerden numbers. In concert with SAT solvers I developed a methodology called tunneling that allowed us to almost double the existing lower bound for W(2,6). My technique involved adding uninferred constraints to the problem. This move effectively cut the search space while keeping sufficient number of partial solutions around so that even after retracting the constraints I was led to solutions. Enormous speedup

was thereby obtained, and I generated the longest partition known so far that can be 2-colored without a monochromatic arithmetic progression of length 6. More precisely, I showed that $W(2,6) \geq 1132$.

In addition to establishing the best lower bound so far I have found a way to compute the Van der Waerden number W(2,6) exactly. Employing assertive elimination of the redundant paths throughout the search space during the preprocessing, and using the notion of unavoidable patterns in a partition of certain length, I was able to cut the original search space by 96.74%. I was also able to speed up my special Van der Waerden SAT solver taking advantage of the various profiling tools and optimization tools yielding a feasible running of about one year on available department clusters consisting of about 200 CPU. The speed was verified after 6 months of computation where I explored about 50% of the search space.

I also ported the Van der Waerden solver into the FPGA environment. Using FPGAs was motivated by the observation that the majority of time in the PC-based solver is spent checking for progressions. This single task can be done in the reconfigurable environment in a single clock step, suggesting a potentially much faster solver. After simulation and final implementation the FPGA based solver proved to be significantly than the PC-based solution. One FPGA evaluation board running 100MHz design yielded the same speed as all the department clusters. Obviously with the available 4 boards the remaining work will be completed in less than two months compared to previously anticipated 6 months.

7.1 Future Work

What I have done can be built upon and expanded to various other problems.

The InterCluster Interface library could highly benefit from group communication and aggregate operations although this is a challenging problem in itself. I hope that the MPI standard will eventually include interoperable communication even on a simple point-to-point communication level. I have been informed that this extension to the standard might happen as early as next year.

The Backtracking Framework can be expanded as well. Not only can new problems can be implemented using BkFr, but also the framework itself can be optimized in various areas. For example, more paradigm specific functionality can be added in addition to those for the backtracking part. In the teaching aspect of the framework, more templates and greater accessibility, such as using a web interface, can be further developed

Results in the Van der Waerden numbers area might bring even more significant outcomes once the methodology developed in this thesis is further applied. The FPGA design shows good potential, and with 210 MHz designs or newly released Xilinx Virtex-5 chips it is not unreasonable to assume that an additional lower bound or two or even the exact number might be determined. Specifically W(2,7) with the tunneling might bring a new lower bound, and W(5,3) could be computed exactly. I believe that both of these numbers have significantly greater lower bounds than currently known.

The tunneling technique described in this dissertation has greater applicability. If applied to other problems such as BMC (Bounded Model Checking) I might yet again see other problems previously considered unsolvable to yield solutions much earlier in the search.

In the FPGA world more work could be done to make the designs more flexible and easier to manage. For example implementing ICI directly onto an FPGA is an option that would reduce the problem of interfacing the FPGA with the rest of the computation to merely plugging it into the regular Ethernet network. Also, the partial self-reconfiguration could achieve independence from the configuration interface and allow the board to function truly independently. This independence would also allow quick changing of the designs, not only in response to input parameters, but possibly also in response to the position in the search tree. In this way even bigger problems might be solvable in the FPGA world.

Bibliography

[1] "Ethernet, http://en.wikipedia.org/wiki/Ethernet."

[2] "Myrinet, http://en.wikipedia.org/wiki/Myrinet."

[3] "Infiniband, http://en.wikipedia.org/wiki/Infiniband."

[4] "Beowulf Clusters, http://www.beowulf.org."

[5] "MPI Standard 1.1, http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html."

[6] "MPI Standard 2.0, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html."

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-performance, portable implementation of the MPI Message Passing Interface Standard," in *Parallel Computing*. vol. 22, 1996, pp. 789-828.

[8] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: A High-Performance, Heterogeneous MPI," in *Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, 2006.

[9] J. M. Squyres and A. Lumsdaine, "A Component Architecture for LAM/MPI," in *10th European PVM/MPI Users' Group Meeting*, Venice, Italy, 2003.

[10] IMPI Steering Committee, "IMPI - Interoperable Message-Passing Interface, http://impi.nist.gov/IMPI/." 1998.

[11] M. Kouril and J. L. Paul, "Brief announcement: dynamic interoperable point-to-point connection of MPI implementations," in *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, Las Vegas, NV, USA, 2005, pp. 352-352.

[12] M. Kouril and J. L. Paul, "Dynamic Interoperable Message Passing," in *12th European Parallel Virtual Machine and Message Passing Interface Conference PVMMPI*, Sorrento, Italy, 2005, pp. 167-174.

[13] M. Kouril and J. L. Paul, "A parallel backtracking framework (BkFr) for single and multiple clusters," in *Proceedings of the first conference on computing frontiers on Computing frontiers*, Ischia, Italy, 2004, pp. 302--312.

[14] J. L. Paul, M. Kouril, and K. A. Berman, "A template library to facilitate teaching message passing parallel computing," in *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, Houston, Texas, USA, 2006, pp. 464-468.

[15] "Boolean Satisfiability Problem, http://en.wikipedia.org/wiki/Boolean_satisfiability_problem."

[16] S. A. Cook, "The Complexity of Theorem Proving Procedures," in *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, Shaker Heights, Ohio, United States, 1971, pp. 151-158.

[17] B. L. Van der Waerden, "Beweis einer Baudetschen Veermutung," *Nieuw Archief voor Wiskunde* vol. 15, pp. 212–216, 1927 1927.

[18] "Arithmetic progression, http://en.wikipedia.org/wiki/Arithmetic_progression."

[19] M. Kouril and J. Franco, "Resolution Tunnels for Improved SAT Solver Performance," in *Eighth International Conference on Theory and Applications of Satisfiability Testing*, St. Andrews, Scotland, 2005, pp. 143-157.

[20] "FPGA, http://en.wikipedia.org/wiki/FPGA."

[21] "Moore's Law, http://en.wikipedia.org/wiki/Moore's_law."

[22] X. C. Robert, "Be absolute for death: life after Moore's law," *Commun. ACM*, vol. 44, p. 94, 2001.

[23] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience.," *Concurrency - Practice and Experience*, vol. 17, pp. 323-356, 2004.

[24] "Condor description, http://www.cs.wisc.edu/condor/description.html."

[25] "The Cactus Code, http://www.cactuscode.org."

[26] "Using the Globus Toolkit to Extend Cactus Capabilities to Address Network Performance Problems (Draft), http://www.ncsa.uiuc.edu/News/datalink/0107/Globus/cactus2.html."

[27] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing : Design and Analysis of Parallel Algorithms*: Addison-Wesley Pub. Co, 1994.

[28] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *International Journal of Supercomputer Applications,* vol. 8 (3/4), 1994 .

[29] Message Passing Interface Forum, "MPI-2: A Message-Passing Interface Standard," *The International Journal of Supercomputer Applications and High Performance Computing,* vol. 12(1-2), 1998.

[30] "The Globus Project, http://www.globus.org."

[31] A. Marzetta, "A Library of Parallel Search Algorithms and Its Use in Enumeration and Combinatorial Optimization." vol. PhD Dissertation: ETH Zurich, 1998.

[32] A. Bruengger, A. Marzetta, K. Fukuda, and J. Nievergelt, "The Parallel Search Bench ZRAM and its Applications " *Baltzer Journal* 1997.

[33] P. Sanders, "Parallelizing NP-Complete Problems Using Tree Shaped Computations " in *Journées de l'Informatique Messine (JIM)* Metz, 1999.

[34] R. Karp and Y. Zhang, "Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation," *ACM Journal,* vol. 40, 1993.

[35] K. A. Berman and J. L. Paul, *Algorithms: Sequential, Parallel and Distributed*: Thomson Course Technology, 2005.

[36] M. J. Quinn, *Parallel programming in C with MPI and openMP*. Dubuque, Iowa: McGraw-Hill, 2004.

[37] P. S. Pacheco, *Parallel programming with MPI*. San Francisco, Calif.: Morgan Kaufmann Publishers, 1997.

[38] N. Truong, P. Roe, and P. Bancroft, "Automated Feedback for "Fill in the Gap" Programming Exercises," in *Australasian Computing Education Conference* 2005.

[39] L. R. Neal, "A system for example-based programming," in *SIGCHI Conference on Human Factors in Computing Systems: Wings For the Mind* New York, NY: ACM Press, 1989.

[40] D. L. Silverstein, "Template Based Programming in Chemical Engineering Courses," *American Society for Engineering Education,* 2001.

[41] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communication of the ACM,* vol. 5, pp. 394–397, 1962.

[42] J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. M. Vanfleet, "SBSAT: a state-based, BDD-based satisfiability solver," in *6 thInternational Conference on Theory and Applications of Satisfiability Testing (SAT'03)* S. Margherita, Italy, 2003, pp. 151-158.

[43] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *39th Design Automation Conference* Las Vegas, 2001.

[44] M. Dransfield, V. Marek, and M. Truszczynski, "Satisfiability and Computing van der Waerden numbers," in *Proceedings of SAT-2003,* 2003.

[45] J. Franco, M. Kouril, J. S. Schlipf, S. Weaver, M. Dransfield, and W. M. Vanfleet, "Function-complete lookahead in support of efficient SAT search heuristics," *Journal of Universal Computer Science,* 2004.

[46] E. Gabriel, M. Resch, T. Beisel, and R. Keller, "Distributed Computing in a Heterogeneous Computing Environment," in *Proceedings of the 5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*: Springer-Verlag, 1998, pp. 180--187.

[47] M. Brune, J. Gehring, and A. Reinefeld, "A Lightweight Communication Interface for Parallel Programming Environments," in *HPCN'97*: Springer-Verlag, 1997.

[48] R. Rabenseifner, "MPI-GLUE: Interoperable High-Performance MPI Combining Different Vendor's MPI Worlds," in *European Conference on Parallel Processing,* 1998, pp. 563-569.

[49] O. Aumage, G. Mercier, and R. Namyst, "MPICH/Madeleine: a True Multi-Protocol MPI for High Performance Networks," in *International Parallel and Distributed Processing Symposium,* 2001, p. 51.

[50] T. Imamura, Y. Tsujita, H. Koide, and H. Takemiya, "An Architecture of Stampi: MPI Library on a Cluster of Parallel Computers," Springer-Verlag, 2000, pp. 200--207.

[51] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPI's collective communication operations for clustered wide area systems," in *ACM SIG-PLAN Notices*. vol. 34, 1999, pp. 131-140.

[52] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface," *Journal of Parallel and Distributed Computing*, 2003.

[53] G. E. Fagg, K. S. London, and J. Dongarra, "MPI_Connect Managing Heterogeneous MPI Applications Interoperation and Process Control," in *Proc. of the 5th European PVM/MPI Users' Group*: Springer-Verlag, 1998.

[54] J. Dongarra, G. E. Fagg, G. A. Geist, J. A. Kohl, P. M. Papadopoulos, S. L. Scott, V. Sunderam, and M. Magliardi, "HARNESS: Heterogeneous Adaptable Reconfigurable NEtworked SystemS," *HPDC*, pp. 358-359, 1998.

[55] E. W. Weisstein and e. al., "van der Waerden Number. From MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/vanderWaerdenNumber.html."

[56] P. Herwig, M. Heule, M. van Lambalgen, and H. van Maarejn, "A new method to construct lower bounds for Van der Waerden numbers," TUDelft.

[57] M. Dransfield, L. Liu, V. Marek, and M. Truszczynski, "Using Answer-Set Programming to study van der Waerden numbers," in *Logic and Artificial Intelligence Laboratory, Computer Science Department, College of Enginnering, University of Kentucky*. vol. Available from http://cs.engr.uky.edu/ai/vdw/, 2004.

[58] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the Association for Computing Machinery*, vol. 12, pp. 23-41, 1965.

[59] P. Beame and T. Pitassi, "Simplified and improved resolution lower bounds," in *37th Annual Symposium on Foundations of Computer Science* 1996, pp. 274-282.

[60] P. Beame, R. M. Karp, T. Pitassi, and M. Saks, "On the complexity of unsatisability proofs for random k-CNF formulas," in *30th Annual Symposium on the Theory of Computing*, 1998, pp. 561-571.

[61] E. Ben-Sasson and A. Wigderson, "Short proofs are narrow - resolution made simple," *Journal of the Association for Computing Machinery*, vol. 48, pp. 149-169, 2001.

[62] V. Chvátal and E. Szemeredi, "Many hard examples for resolution," *Journal of the Association for Computing Machinery*, vol. 35, pp. 759-768, 1988.

[63] Z. Galil, "On resolution with clauses of bounded size," *SIAM Journal on Computing*, vol. 6, pp. 444-459, 1977.

[64] A. Haken, "The intractability of resolution," *Theoretical Computer Science*, vol. 39, pp. 297-308, 1985.

[65] A. Urquhart, "Hard examples for resolution," *Journal of the Association for Computing Machinery*, vol. 34, pp. 209-219, 1987.

[66] I. P. Gent and T. Walsh, "Towards an understanding of hill-climbing procedures for SAT," in *Proc. 11th National Conference on Artificial Intelligence*, 1993, pp. 28-33.

[67] J. Gu, "Eficient local search for very large-scale satisfiability problems," *SIGART Bulletin*, vol. 3(1), pp. 8-12, 1992.

[68] J. Gu, P. W. Purdom, J. Franco, and J. Wah, "Algorithms for the Satisfiability problem: a survey," *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, vol. 35, pp. 19-151, 1997.

[69] D. McAllester, B. Selman, and H. A. Kautz, "Evidence for invariants in local search," in *International Joint Conference on Artificial Intelligence* 1997, pp. 321-326.

[70] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *12th National Conference on Artificial Intelligence*, 1994, pp. 337-343.

[71] S. Weaver, J. Franco, and J. Schlipf, "Extending Existential Quantification in Conjunctions of BDDs," in *University of Cincinnati Technical Report*.

[72] J. R. Rabung, "Some Progression-Free Partitions Constructed Using Folkman's Method," *Canadian Mathematical Bulletin,* vol. 22(1), pp. 87-91, 1979.

[73] H. Y. Song, S. W. Golomb, and H. Taylor, "Progressions in Every Two-coloration of Zn," *Journal of Combinatorial Theory,* vol. Series A 61(2), pp. 211-221, 1992.

[74] B. Landman, A. Robertson, and C. Culver, "Some new exact van der Waerden numbers," *Integers: Electronic J. Comb. Number Theory,* vol. 5, 2005.

[75] A. M. Dewey, *Analysis and design of digital systems with VHDL*. Boston: PWS Pub. Co., 1997.

[76] Z. Navabi, *VHDL : analysis and modeling of digital systems*, 2nd ed. New York: McGraw-Hill, 1998.

[77] "Xilinx Virtex-4 User Guide."

[78] F. Rivoallon, "Achieving Breakthrough Performance in Virtex-4 FPGAs," in *Xilinx white paper*. vol. 218, 2006.

[79] AVNET, "Xilinx Virtex-4 Evaluation Kit User Guide."