Parallel Search in Gecode

Morten Nielsen

May 17, 2006

Abstract

Constraint programming is an area of computer science which has developed rapidly over the last years. Constraint programming offers the tools needed to solve larger combinatorial problems with sufficient speed. The key to the success of constraint programmings success in combinatorial problem-solving over conventional search are clever algorithms and resource management. Today, computers with multiple processors working on shared memory are increasingly common. It is therefor natural to provide tools to use multiple processors for solving combinatorial problems with constraint programming. GECODE is a constraint programming library which introduces new and faster algorithms for finding solutions and using resources. GECODE however lacks support for using multiple processors. This thesis describes the work done too produce an adaptible system for using GECODE to do parallel search in a shared memory environment.

This thesis handles the design and implementation of the tools needed for GECODE to transparently take advantage of multiple processors in a shared memory environment.

Contents

1 Introduction **3**
- 1.1 Acknowledgments . . . 3
- 1.2 Related work . . . 3
- 1.3 Plan of the Paper . . . 3
- 1.4 Short Introduction to Constraint Programming . . . 4

2 Sequential search **6**
- 2.1 Short overview of branching . . . 6
- 2.2 Backtracking . . . 6
- 2.3 Search-engine . . . 8
- 2.4 Best solution search . . . 8
- 2.5 Controlling search . . . 9

3 Search in GECODE **10**
- 3.1 Space . . . 10
- 3.2 Stack . . . 11
- 3.3 Search-engine . . . 12

4 Parallel Search **14**
- 4.1 Benefits of parallel search . . . 14
  - 4.1.1 Super-linear speedup . . . 15
- 4.2 Issues with parallel search . . . 15
  - 4.2.1 Load balancing . . . 16
  - 4.2.2 Backtracking . . . 17
  - 4.2.3 Nondeterminism . . . 17
  - 4.2.4 Shared stack or individual stack . . . 18
  - 4.2.5 Termination detection . . . 18
- 4.3 Possible frameworks . . . 18
  - 4.3.1 Search loop in parallel . . . 18
  - 4.3.2 Manager-Worker architecture . . . 19

5 Architecture **20**
- 5.1 Overview . . . 20
- 5.2 Worker . . . 21
- 5.3 Manager . . . 21

6 Implementation **23**
- 6.1 Implementation decisions . . . 23

6.2 Framework 24
6.2.1 Controller 25
6.2.2 Worker 26
6.2.3 Search Script 27
6.2.4 Sharing strategy 28
6.3 Implemented Search Scripts 29
6.3.1 Depth First Search 29
6.3.2 Branch and Bound Search 29
6.4 Possible expansions 29

7 Analysis **31**
7.1 Correctness 31
7.2 Portability 32
7.3 Test cases and test environment 32
7.4 Overhead 33
7.5 Runtime 34
7.6 Improvements 37

8 Conclusion **38**

Chapter 1

Introduction

1.1 Acknowledgments

I would like to thank Christian Schulte for offering me the oppertunity for writing this master thesis, as well as guiding me through my work with it. I would also like to thank Guido Tack and Didier Le Botlan for their paper: Compositional Abstractions for Search Factories which inspired me in how to implement the parallel search-engine. I would also like to thank Fredrik Liljeblad for helping me set up a test-environment.

1.2 Related work

This system is a part of the Gecode constraint programming system. A system introducing some novel ideas in the area of constraint propagation and propagator construction. Other constraint programming systems exist with the capability of doing parallel search such as: Mozart, ILOG[3] and Eclipse.

Though not constraint programming systems the ALPS[12] and SYMPHONY[2] systems are systems for handling integer programming, capable of doing parallel search in trees.

1.3 Plan of the Paper

This paper will start by giving a short explanation of the basic concepts in constraint programming, after which the concepts of search are examined. The focus will then turn to parallel search, for which the rest of the paper will be about. First the concept of parallel search is presented, followed by an examination of different ways to implement it. Lastly the implementation is presented along with motives as to why its designed as it is.

1.4 Short Introduction to Constraint Programming

Constraint programming provides a generic way to solve a wide variety of problems. The problems are modeled as discrete variables which are assigned a range of values, in which a solution should lie, as well as constraints for the values of the variables. These three components, the variables, their values and the constraints, are kept in an object called a store.

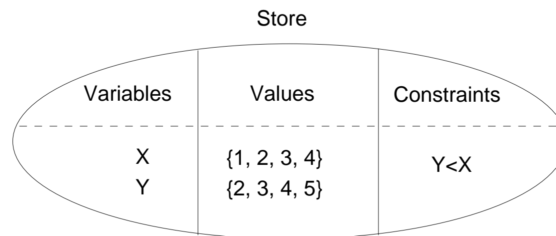

Figure 1.1: An abstract representation of a store with its variables, values and constraints.

Through a process called propagation the values which does not hold for the constraints, are removed from the possible values of the variables. The constraints in the store are actually functions called propagators which implement the constraints on the store, propagators can only remove values for which they are absolutely sure that they cannot be a part of a solution. The goal of propagation is to reduce the ranges to either achieve a solved or a failed store. However in most cases propagation is not enough to solve the problem. For example propagation on the store represented in Figure 1.1 will result in a store where $X \models \{3, 4\}$ and $Y \models \{2, 3\}$ which is still not a solution for the problem.

A store can have three possible states. Failed, one or more of the variables has no values left in its range; solved, each variable has exactly one value to choose from; and distributable which are all store which are neither solved nor failed. The propagated store mentioned earlier is thus a distributable store.

To further reduce a distributable store, branching is used. Branching is a way to reduce the store without losing accuracy by removing solutions. Branching takes a copy of the store and adds the constraint β to it, it also takes another copy of the store and adds the constraint $\neg\beta$. An example of the constraint β could be $X = 3$ which would then reduce the store to a solved store. The branching forms a binary tree where each store is a node in the tree. Every leaf node in the tree is either a failed or solved store, while the body of the tree consists of distributable stores.

This tree is explored through the process of exploration or search. Exploration is done through a search-engine, which can explore the tree in various ways, for example: depth first or breadth first. As propagation can take a long time the tree is not constructed before the search is started. Instead the tree is lazily constructed while the search-engine is exploring the tree. Meaning each node is constructed on demand rather than in advance. Sequential search-engines will

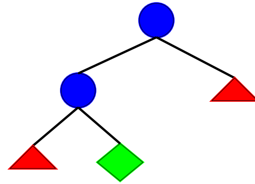

Figure 1.2: An example of a search tree. The round nodes are distributable, the triangular are failed and the diamond are solved.

be covered in greater detail in Section 2.

To conclude this section lets consider how a search-engine would explore a search-tree using the example store in Figure 1.1. First the initial store is propagated to yield a distributable store. The search-engine then resorts to branching too further reduce the store. The store is copied and the constraint $X = 3$ is added too one of the stores. Propagation is done again on the specialized store which would show that the store is now a solution and the search is stopped. This example has many solutions and it might be the case that all solutions need too be found, in which case the search-engine continues its search.

Chapter 2

Sequential search

This chapter will go deeper into the details of searching. First the concept of branching will be briefly described. The paper will then shortly explain how a search-engine works and then continue with different aspects of searching.

2.1 Short overview of branching

How the constraints of the branching are defined affects what the search tree will look like. There are some common techniques for branching, such as: first-fail and naive. The goal when selecting which technique too use is to try to move the solutions closer to a part of the search tree which will be explored early in the execution. Since it is tradition that most depth first searches move towards the bottom left part of the tree, the selected technique should place solutions close to the bottom left of the tree. For this reason, selecting which technique to use is often done when modeling the problem to solve.

Naive branching selects a variable and binds a value to it as a constraint β. The second constraint $\neg\beta$ is of course the constraint where the variable does not have this value. First-fail selects the variable with the smallest set of possible values. In order to quickly reduce the size of the variables value ranges.

2.2 Backtracking

When a depth first search based search-engine explores the tree it will in most cases come too a point where it has reached a failed node. If so, then the search-engine will have to go back to a previous node and try a different path. This is called backtracking, and it is essential for the search-engine to explore the entire tree.

Backtracking is often done using a stack as illustrated in Figure 2.1. For each node the search-engine passes it pushes a copy of it onto the stack. The search-engine continues down a specific path of the tree until the current node fails. If a

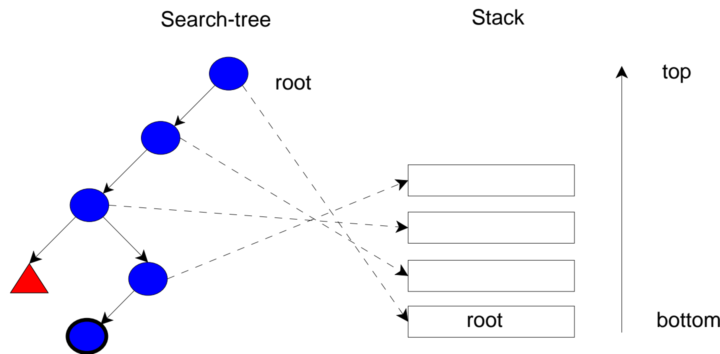

Figure 2.1: An illustration of how a stack is used to keep track of passed nodes in a search-tree for backtracking purposes.

node fails the search-engine pops the previous node from the stack and explores the unexplored path from it. If the node the search-engine just popped has no unexplored path it is treated as a failed node and is discarded. Again a new node is popped from the stack. When the stack is empty, the search-engine has examined the entire search tree.

As problems grow in complexity, the size of their search trees grows as well. Thus keeping information about each node in the search-tree can lead to memory consumption problems. Keeping all the copies of all the nodes in the stack is referred to as copying. There are other techniques to handle the backtracking which reduce the memory consumption.

Trailing is a technique where the system keeps record of all the changes to the store, in order to be able to undo the changes in case backtracking is needed. By using this technique only one store needs to be kept in memory at all times, however information about every change on that store also needs to be kept. In the case of major changes to the store the information about the changes can at times consume more memory than if copying had been used instead [7].

Another technique is called re-computation. By using re-computation the search-engine keeps the initial node at all times. All other nodes can then be recomputed as long as the system keeps a record of the track taken in the search-tree to come to a specific node. It is said that the root node contains a working copy of the spaces, the same as the current node. A working copy is just like a full copy. The other nodes except for the one being examined are reduced to save memory.

If all the nodes in a tree need to be recomputed from the root of the tree, then all re-computations will take longer and longer as the tree gets deeper and deeper. To speed-up the re-computation a working copy can be kept deeper in the tree. Having several working copies in the search-tree is illustrated in Figure 2.2. Usually if this is the case then the distance between a working copy and the working copy above it is called the copy distance. As at specific distances from the top a copy of the non-reduced space is maintained.

Both trailing and re-computation are efficient techniques to reduce the memory

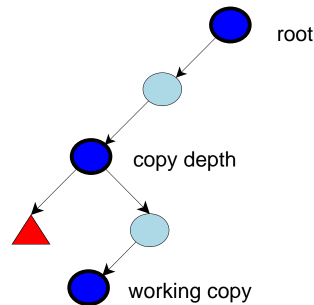

Figure 2.2: An illustration of a tree using fixed re-computation. The darker nodes contain a working copy while the lighter shaded nodes are reduced and needs to be re-computed. The triangle node is failed and thus naturally not kept at all.

consumption of a search-tree. However both techniques have drawbacks when it comes parallel search. These issues will be covered in Section 4.

2.3 Search-engine

The exploration of the search-tree is done by the search-engine. The search-engine is a central part of a constraint programming service. Defining how a search-engine should explore the tree often requires an understanding of the entire system and the search-engines are often very complex and monolithic[10].

In GECODE a sequential search-engine mostly consists of two parts. The first part is a stack on which it keeps track of nodes it has to go back to in case it has to backtrack. The second part is a search-loop which describes how the search-engine should handle each node of the tree as it works down a path in the search-tree. In general the search-engine starts by doing propagation on a node, and then if the node is distributable it commits to one of its alternatives. If the node is failed the search-engine discards it and uses the stack to go back to the previous node and down the second alternative. If the node is a solution that solution is returned to the calling process.

2.4 Best solution search

In some problems it is the optimal solution which is desired. A common technique for achieving this with sufficient speed is through the branch and bound paradigm. A branch and bound search-engine first explores the tree to find a solution. It then restarts the search but with an additional constraint that the next solution it finds will be a better solution than the previous one, and searches the search-tree again. This is repeated until no solution is found, and thus the previously found solution is the best solution for the problem.

2.5 Controlling search

When a search-engine explores a tree it has to choose between one of two paths. There is no way of knowing what each path will lead to other than the limited control of selecting a branch heuristic. Selecting the wrong path can lead to the search-engine going down a branch which does not lead to any solutions and in effect is a waste of effort. Therefor it would be prudent to try to control the search so that the search-engine tries to avoid such situations. Several methods are available to reduce the risk of getting stuck in a fruitless subtree, for example: to allow the search-engine to make jumps in the search tree[3] or to limit the depth to which the search-engine searches and then increment the depth if no solutions are found [3, 1].

Chapter 3

Search in GECODE

Section 1.4 and Section 2 describes the workings of sequential search. This section aims to build on Section 2 with how search is done in GECODE. First the section will describe basic data structures in GECODE, and then it will continue by describing how search-engines are implemented.

The GECODE system has several kinds of search-engines implemented. To cover all these however is somewhat out of the scope of this thesis. Instead I will describe only two of these search-engines, the two of which I have constructed parallel versions of.

3.1 Space

The store described in 1.4 is a fundamental construct for doing constraint programming. In GECODE the store is implemented in a construct called a Space. A space is the working space for propagators, in GECODE a problem is modeled as a Space with added propagators and variable ranges. The propagators of the space are defined as the space is first constructed; in practice this mean that propagators are defined in the C++ constructor of the space, creating a model of the problem. The space then has the additional functionality to be handled efficiently in a search-engine.

A space has the ability to *clone* itself. Cloning is used in backtracking, a clone of the space is put on the stack so the search-engine can come back to it later. How the stack works will be explained in Section 3.2. A space also has functionality to test its *status*, to see if it is failed, solved or distributable. The *propagate* function forces the space to do propagation. The last function a space has is the *commit* function. The *commit* function adds the branching propagators for one of the subbranches for the store which the space represents. It is thus used for selecting which branch to explore next. Committing to alternative 1 is the same as committing the the left branch of a search-tree and committing to alternative 2 is the same as committing to the right branch.

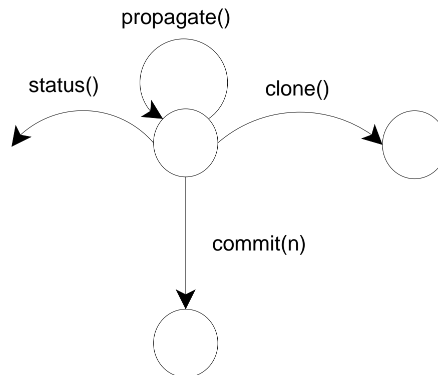

Figure 3.1: Illustration of the functions in a space object. The circles represent a Space and the functions the arrows. The clone and commit functions return a new space, illustrated by the arrow leading to a new space. While the propagate function updates the space by doing propagation.

3.2 Stack

In the GECODE system the functionality for backtracking is handled by the stack. The stack keeps track of which nodes the search-engine has previously been to as described in Section 2.2. In GECODE the responsibility of doing re-computation is also managed by the stack. By keeping the backtracking and re-computation in the stack, the complexity of the search-engine is reduced.

The stacks in the GECODE system are derived from a basic stack with memory handling functions. The two different stacks extends this basic class of stack. One of the two more specialized stacks in GECODE is for backtracking using re-computation, while the other stack is used in copying based backtracking. The different stacks will be explained in greater detail in later sections, one for each type. Each type of stack differs somewhat which functions interface to it, so search-engines need to be constructed to use either re-computation or copying.

Nodes

In order to do re-computation and keeping track of which subtrees have already been explored in the search-tree, a special object is needed. In GECODE the stack makes use of a node object, which keeps the information needed for doing re-computation and backtracking. The node objects can contain spaces, but they can also forget the spaces; keeping only the information needed to do re-computation of the space when it is later requested by the search-engine. The node object is quite generic, it keeps track of the the amount of subbranches there is at the specific space and also which of them what already been explored by the search engine. This information is kept as integer values.

It is important to note that the search-engines in GECODE only worry about Spaces. Therefor the node objects are only used by the stack.

Re-computation

Re-computation in GECODE is handled by the stack. The search-engine calls the stack to receive the next space which it needs. The stack then does re-computation if needed to provide the search-engine with a new workspace. The stack keeps track of the spaces which the search-engine passes. When doing re-computation the information stored in the node object on the stack is which branch was followed by the search-engine when it passed this space. This information is then used when doing re-computation of a particular space, as the stack can follow all the commits done by the search-engine to get to that space.

Copying

The version of the stack which handles copying based backtracking is quite simple compared to its re-computation based cousin. The stack maintains nodes which the search-engine needs to return to at a later point in time. This means that the stack holds the right branches of the nodes passed in the search-tree. When a search-engine asks for a new space this version of the stack simply returns the top of the stack. The stack contains clones of the passed spaces so all which is needed to be done is to commit to the alternative which has not yet been explored.

3.3 Search-engine

The GECODE system offers many different search-engines. These search-engines are available in versions supporting both re-computation and copying.

All search-engines in GECODE work through a stack. The workings of this data structure and how it is implemented in GECODE is described in Section 3.2. Each search-engine is implemented in two parts. The first part is a templated version which handles the specialized Space in which the problem is specified, the second part is a more general part handling search for any Space object. The templated part call functions in the more general version of the search-engine. This generic part of the search-engine is dependent on the arguments which were specified as the search-engine is instanced. These arguments decide, for example; if the general part of the search-engine should use re-computation or copying.

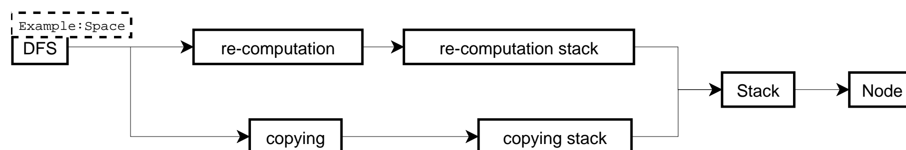

Figure 3.2: An illustration of how the search-engine is connected with the other components for doing search in GECODE.

Depth first search

The depth first search engine in GECODE is perhaps the simplest of the search-engines in the system. It consists as mentioned earlier of a templated version which then starts a more generic search-engine which uses either re-computation or copying. Figure 3.2, which describes how search-engines are connected with the components for search, is illustrated using the depth first search search-engine as a model.

Branch and bound

The implementation of the branch and bound search-engine in GECODE is somewhat more complex than the implementation of the depth first search. This is because of the way problems are modeled using the GECODE constraint programming library. As mentioned earlier models of problems are specified in the constructor of a space. While this is true a problem is also a specialized case of a space meaning that a problem is implemented as an extended object from the space class. Thus it is only in the this specialized form that the a search-engine has access to problem specific functions, such as the *constrain* function. The *constrain* function is used to constrain a problem to be better than the solution which is specified through the argument. Therefor when the branch and bound search-engine needs to constrain a space it needs to be done in the templated part of the search-engine which has access to the special function.

When a new solution is found, the search-engine continues to try to prove that it is either the best solution or that there is a better one. Therefor the entries which are on the stack when a solution is found need to be constrained. To keep track of which entries on the stack that have already been constrained since the last solution was found the search-engine maintains a mark. The mark is set to the number of entries on the stack when a new solution is found. Each entire below that mark then needs to be constrained before any further work is done on that space. Naturally as nodes are popped from the stack the mark is decreased so that spaces wont be constrained in vain.

Chapter 4

Parallel Search

This section begins by describing some of the benefits of parallel search. It continues by addressing some of the issues with parallel search and last it covers different suggested frameworks for constructing parallel search engines. The focus of this paper is parallel search in a shared memory environment, not distributed search over several computers. While there are similarities in the issues and benefits, distributed search needs to cover issues which are out of scope for this thesis.

4.1 Benefits of parallel search

Parallel search has several benefits from sequential search. In the case of multiple processors the parallel search can take advantage of all the processors in the system, where sequential search only uses one. This gives parallel search the advantage of additional computational power. This extra power makes exploration fast, however the speedup is not necessarily linear to how many CPUs used in the search[11]. Nor does this added power allow parallel search to solve problems sequential search cannot solve[4]. In a time where hardware with multiple CPUs is increasingly available it seems wasteful not to have a search-engine which can take advantage of the additional resources.

Parallel search has other advantages over sequential search, for example: A sequential search engine using depth first search can, as mentioned in Section 2.5, wander down a branch in which no solutions are available, thus resulting in an entire subtree of the tree is fully explored before the search engine returns to a point where it can explore the other parts of the tree. A parallel search engine can explore several branches at the same time, reducing the risk of the search getting stuck in a branch where no solutions are found.

4.1.1 Super-linear speedup

Due to the non-deterministic nature of a parallel search-engine the search time for a solution can at time be significantly better than that of a sequential search-engine. This depends on the order of the exploration; a sequential search-engine always follow a specific route while a parallel search-engine can differ from this static path, since it can explore spaces in parallel. Should a solution be found faster in one of the paths taken by a parallel thread which is not in the sequential path a speedup is obtained. This speedup is called super-liner speedup. This is when N threads explore a search-tree a speedup greater than N is obtained[6]. Super-linear speedup is however not predictable and at most times the speedup obtained by a well implemented parallel search-engine is slightly less than N.

4.2 Issues with parallel search

This section will discuss issues with parallel search-engines. Therefor the focus of this section will not handle problems with general parallel programming, but rather try to describe the pitfalls of parallel search.

While parallel search offers speedup and better resource utilization it also suffers from some issues. Doing search in parallel gives an overhead in runtime for handling threads and synchronization. However, the efficient use of several processors can often compensate for some of the overhead compared to sequential search engines.

There are several sources of inefficiency in a parallel search engine, which are not present in sequential search engines. It is important to manage these sources to increase the performance of the parallel search engine. The most common sources of inefficiency are:

- Communication Overhead, referring to the overhead resulting from communication between threads.
- Idle Time, it is inefficient to maintain a thread which does no work. A thread which has no work will also in most implementations of parallel search-engines result in an increased communication overhead. This is because the system will try to contact the other threads to collect work for the idle thread.
- Redundant work. Parallel search can produce more work than a sequential search engine would have before finding a solution. If this work is not directed towards finding the solution, then that work is redundant and thus inefficient.

In most systems there is a period at the start and sometimes at the end of an exploration where most threads are idle[12]. A sort of warm-up period before the system is stable, followed by a cool-down period at the end of the exploration. At the start this is because most systems start an exploration by giving a space to one of the threads. This thread then shares work with the other workers. As the workers complete the exploration of the search-tree the amount of work available is significantly less than at the beginning, thus not all workers will

be able to receive work. The cool-down period is only significant if a complete exploration of the search-tree is done. The effect is caused by the lack of work as the search-tree is being explored.

Coordination

In certain types of parallel search engines the parallel searches need to coordinate their progress with each other. For example the branch and bound search needs to keep information about the previously found solution. When searching in parallel this information needs to be shared between all the threads. To do this a communication layer is required. Different paradigms for sending information between threads are available.

- Message passing, where each thread has a mailbox and to communicate threads send mail to each other. Message passing provides an abstract way of handling communication and several search engines use message passing for communication[9, 10].
- Data flow, where threads communicate through mutexes acting as traffic lights, as well as shared variables.

Both of these approaches requires extra checks in the system. Either to verify that no other thread is currently in a critical section of the code, or to check for new messages at certain time periods. As mentioned earlier these checks lead to runtime overheads, compared to a sequential search-engine.

4.2.1 Load balancing

An issue with doing parallel search is to always make sure that the different threads always have work to do. When a thread has nothing to work on it has to find work in order to continue. To look for work takes time and often locks other threads from working. Therefore it is not a good idea to look for work too often. To further complicate things each thread also needs to have a fruitful part of the search tree to work on[12].

The decision to share work with another worker should consider giving a fertile part of the tree to the worker without sacrificing its own work to do so. Giving a fertile part such as the lowest left-most part might lead to starvation if there is too little work in the shared part. While giving a huge chunk of the rightmost part of the search tree will keep the thread with work, the thread will most likely not find many solutions.

Effective load balancing can reduce the amount of redundant work done by the parallel search engine[12]. As an example: consider the case where there are two threads, one with work and one looking for work. The branching algorithm has been chosen so the solutions are in the leftmost part of the tree. If the one with work available decides to give a huge chunk of its rightmost unexplored tree then both workers will most likely have work until the solution is found. However giving a part of the leftmost part of the tree might make the other worker run out of work quicker. However it will also have a greater chance of finding a solution and thus ending the search faster.

There are other aspects of load balancing as well such as: should load balancing be initiated by a thread which has no work, or should it be initiated by another thread that notices that a thread has no work? If the idle thread looks for work which working thread should it contact? Should it contact a supervisor which then finds work for the thread? There are a number of solutions too how load balancing can be handled [11], therefor I will briefly describe two techniques. One where a request for work is issued by the idle thread and one where the sharer initiates the work transfer.

- Global round robin, every-time a thread is idle it looks for work at other threads in a round-robin fashion. The system needs to keep a global variable to keep track of which thread was contacted last.
- A supervising thread keeps a record of which threads currently have work, when it notice that a thread is idle it requests work from one of the working threads and assigns it to the idle thread.

4.2.2 Backtracking

As covered in Section 2.2, backtracking is an important part of search. The information needed for doing backtracking consumes memory, which leads to memory consumption problems. Problem which are solved either by re-computation or trailing. When it comes to a parallel setting however, these techniques have drawbacks.

Re-computation sufferers issues when it comes to parallel settings. While re-computation is more expressive in the sense that any node can be made available at any time. The node needs to be recomputed, which will lock both the sharing thread and the receiving thread from doing any work during that time.

Trailing also sufferers from lack of expressiveness. There is only a single node available to the search-engine during search. This property of trailing makes it harder to share nodes with other workers, as they require nodes which the current thread is not working on [7].

A copying approach consumes more memory than both of the previous alternatives, but has the advantage that all the nodes are available for sharing without the need for re-computation.

4.2.3 Nondeterminism

Exploring the tree in parallel may lead to a solution being found in a path which would have been explored after a different solution had been found, had it been a sequential search engine exploring the tree. Issues of scheduling may also affect which solution is found next. This gives parallel search an issue of non-determinism. It is therefor not possible to be sure which solution is found next. This is sometimes called Or-Parallelism[5]. However this nondeterminism also has benefits, and is rather a consequence of the parallel search.

4.2.4 Shared stack or individual stack

As mentioned in Section 2.2 the search engine keeps a stack of the nodes it has passed during search. When doing a search in parallel this stack can be unique for each thread or it can be shared between them. Sharing the stack with all other threads may lead to a lot of synchronization on the stack. Since the stack is a critical section of data only one thread can have access to it at the time. This forces the other threads to wait for their push and pop operations on the stack, operations which occur often in each thread. This makes the shared stack a source of contention [1, 12, 2].

An individual stack on the other hand does not force the thread to wait for other threads when doing its push and pop operations. However, each thread has to find work when its stack is empty, and termination detection is somewhat harder when each thread has its own stack. Several implementations of parallel search-engines use individual stacks for each thread [3, 6, 8].

4.2.5 Termination detection

To detect that a search is done in a sequential search is easy. The search engine knows that the search is done when the stack is empty. However, in a parallel search-engine there can be several stacks to keep track of, as well as the possibility of a node being shared to keep the system balanced. Implementations of parallel search-engines which use a shared stack to keep track of the nodes in the tree are basically the same as a sequential search-engine in this regard.

4.3 Possible frameworks

In this section concepts which are more implementation specific is covered and benefits and drawbacks of different approaches are described.

4.3.1 Search loop in parallel

A relatively easy way to implement parallel search is to take a search engine and make its search loop run in parallel in different threads. All the loops would use the same stack to pick unexplored nodes from. The implementation has to add control to the stack as its access functions are the critical sections of the parallel code. These extra controls however force the different search-loops to wait for each other when accessing the stack, which could increase the execution time to the point where it is equivalent to running a sequential search-engine. This kind of framework would also lack basic functions for controlling search, and exploration of the search tree would be non-deterministic.

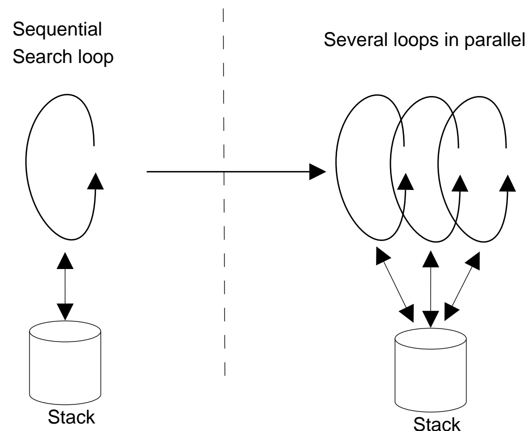

Figure 4.1: Illustration of running several threads with the same search-loop.

4.3.2 Manager-Worker architecture

Another architecture which can also used for doing distributed search over networks, could be used for doing parallel search as well[9]. This architecture can be used in both a distributed environment as well as in a parallel shared memory environment. The architecture suggests using a Manager object along with several Worker objects as autonomous agents communicating by message passing. The benefits of this architecture is that it can be implemented to be used over a network since the different agents of the system use message passing as communication. The solution is not restricted to one agent on each machine so the architecture can support parallel search on a single multi-processor machine.

Figure 4.2: A rough outline of the Manager-Worker Architecture

The Manager handles initialization, load balancing and termination detection. It does so by keeping track of which workers are working and which are not. When a worker is idle the Manager tries too find work for the Worker by asking the ones who have work to share. Termination is detected by the Manager when it receives a solution from one of the Workers; or the Manager notice that there is no work left.

The Workers each have their own stack to work on to which work is assigned by the Manager. As long as the Manager does not tell the Workers to either share work or stop, the workers keep exploring their part of the search-tree.

Chapter 5

Architecture

In this section the architecture which I chose to implement parallel search with is described. It starts by giving an overview of the architecture and then it describes the two main components in the architecture, namely the worker and manager components. The architecture was first presented in Section 4.3.2.

5.1 Overview

A brief overview of the architecture was giving in Section 4.3.2. This overview is somewhat more detailed, but also presents some of the reasons to why I selected this architecture for my implementation.

The architecture was first presented to do distributed search, using several networked computers. But this architecture can just as easily be used in a parallel shared memory environment [9]. I have however made some changes to the architecture as certain overheads such as communication overheads are much smaller in a parallel shared memory environment than in a distributed environment.

The manager is the main part of this architecture. It is the manager who is first initialized and through which the workers then are initialized. The hierarchical structure of the manager and worker is presented in Figure 4.2 in the previous section of this thesis.

The manager and workers in this architecture communicate through messages. The manager has four messages which it can send to the worker; *stop*, *explore*, *status* and *share*. The purpose of these messages are primarily to enable the manager to control search. The worker on the other hand only have two messages to send to the manager process; *find* and *collect*.

5.2 Worker

Each worker corresponds to a search-engine exploring a subtree of the complete search-tree. The worker keeps exploring as long as it has worker or until it receives a stop message from the manager.

Initially the workers are created by the manager. When they are created they are paused, waiting for the manager to send the *explore* message, so that they can start working.

The workers are supposed to take certain actions as their receive messages from the manager and as their state changes. A worker which is idle sends a *find* message to the manager to receive work. A worker which is busy will continue to work on the work it has until one of three things happen.

- The worker runs out of work, and thus becomes idle again or it finds a solution which it then sends to the manager using a *collect* message.
- The worker receives a *stop* message from the manager, in which case it stops all work it is doing and wait for an *explore* message from the manager.
- The worker receives a *share* message from the manager, in which case the worker pauses its exploration of the search-tree to check if it has work which it can share.

A reason why I chose this architecture is that it is not forced to run the same kind of search-engine on each worker[10]. As long as the worker maintains the same set of messages and each response for each message hold correct format, the workers can explore using any kind of exploration technique it wishes. In GECODE search-engine work using the Space object which was described in Section 3.1. As workers share work and return solutions between them they are actually exchange Spaces between each other. Thus allowing the meshing of several exploration techniques in a parallel environment. While this is not the goal of my thesis, I found this property to be intriguing.

5.3 Manager

The manager is the over all coordinator of the search process. It keeps track of several workers which it delegates work to. The manager is thus responsible for handling load balancing, termination detection and maintaining control so the exploration can be stopped.

When the search is first initialized the manager is the first object to be started. The manager in turn starts the specified amount of workers and give the initial root space to the first worker. The manager then starts the exploration of the search-tree with the *explore* message.

As the search progresses the manager has to take care of workers which are idle, finding and providing them with work. The worker is thus sending *find* messages to the manager which in turn tries to find work for the worker. Finding work is

done by using the *share* message, which encourages the receiver of the message to share of its workload; to provide others with work as well.

There are two situations which the manager has to take care of. Termination detection and solution collection. The manager has to figure out when the workers have no more work and thus the search is done with no solution found. The manager also has to take care of solutions which the managers find, and in the case of single-solution search the manager should stop the workers when a solution is found.

A manager receives a solution from a worker as it is found via the *collect* message. As the message is received the manager stops all workers and returns the solution to the one who requested the search. Stopping the search is done through the *stop* message. If the search is stopped due to a solution being found, the solution is sendt with the *stop* message.

Termination detection is more difficult then solution collection. There are no messages sent to the manager as the workers run out of work, instead the manager has to check if all the workers are out of work. If so then the manager should tell all workers to stop and inform the one who requested the search that no solution was found.

Chapter 6

Implementation

In this section I will describe the implementation of the parallel search-engine for the GECODE system. The section starts by giving an overview of major design decisions taken and the developed framework. Then it continues with a more detailed view of each component.

6.1 Implementation decisions

One of the key features of the GECODE system is it's portability. The GECODE constraint programming system can run on several architectures including 64-bit architectures. When implementing my parallel search architecture I tried to keep this in mind. Therefor when deciding which technique to use for parallelization I settled for PThreads. The reason for this is that PThreads is available on many different architectures. Often there is no need to install extra packages for using it. PThreads is also ideal for parallelization on shared memory machines.

The downside of using PThreads is that it does not support message passing. Pthread programs work in a data-flow manner, using synchronization primitives to protect critical sections of code if several threads need to access it at the same time. This presented the challenge of reworking the architecture to use data-flow mechanisms instead of message passing.

The solution was to have each message represented as a function, each function protected by synchronization primitives depending on how sensitive the function was for changing data. For example asking for a workers status does not affect search in anyway, so checking status is does not need extra protection. Compared with for example the *share* function which directly affects the stack. Thus the *share* function has to wait for the search to be in a state which it does not risk to destroy the sensitive memory structure of the stack or otherwise affect the search negatively. The conclusion is that running a function on a worker is the same as sending a message to that worker, and the same goes for the manager.

Using data-flow instead of message passing the manager class no longer have

to run its own thread, since all the functions it would be performing might as well be performed in the functions called by the workers as they want to communicate. Therefor the manager is no longer managing the workers, but acts more as a control layer. Therefor in the implementation I renamed the manager class to controller.

As mentioned in Section 5.2, the possibility of running different search techniques intrigued me. As a result I wanted my implementation to be able to support this. The result was to template the search-technique on the workers. To allow the system which initialized the search to select which search technique to use. Abstracting away all synchronization from the actual search also made it easy to implement different kinds of parallel search-engines, as the synchronization with other workers was taken care of by the worker class. How the worker does search is thus controlled through a templated object called a search-script. This make the system easy to expand and enables code reuse.

Additionally I made an effort to allow the developers who would use the system to specify how sharing should be done between the workers in the system. This is implemented as an object, called search strategy, which the developer can pass to the search-engine to do changes in the basic rules of when and what to share.

6.2 Framework

As mentioned the different parts of the parallel search-engine is thus communicating with each other through functions. Though still acting like a message passing version this change on the architecture is quite significant. The other change is that workers are prepared so they are not required to do search in the same way. The platform still requires some extra code to make this work, but it is prepared for supporting this feature.

A coarse grained overview of the framework is given in Figure 6.1. Which illustrates most of the major changes to the architecture. The function-calls between the objects can still be abstractly viewed as messages, though they differ slightly.

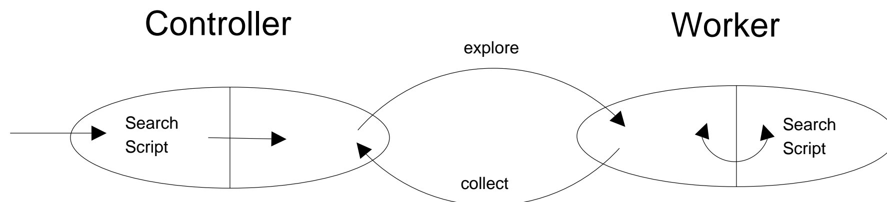

Figure 6.1: A coarse grained overview of how the controller, worker and search script objects cooperating; along with the important collect and explore function calls.

In the architecture described in Section 5 the manager checks for termination detection by checking to see if the workers are idle or not. With the implementation using data-flow the functionality of the manager can just as easily

be done in the find and collect functions. Reducing, as mentioned earlier, the manager to a controlling communication layer, thus named controller instead of manager. This change allows a parallel shared memory machine to use all the processors of the machine to conduct search. Compared to having an additional thread which checks for termination detection. This solution should reduce the overhead of thread handling and communication somewhat, as well as reducing the complexity of the controller object.

6.2.1 Controller

The controller object acts as the communication layer for the workers. The functions in the controller class handles sharing, termination detection and result collection similar to the Manager class described in Section 5.3. The controller however does no computation in its own thread while the system is searching.

The responsibilities of the manager still needs to be taken care of. In the controller these responsibilities are distributed between specialized functions which take care of the specific problem. The functions which the controller interacts with the workers in an abstraction of message-passing using the functions on both the controller and worker objects.

The responsibilities of the controller along with the functions which take care of them are the following:

- Provide a framework for workers to contact each other is search of work, this is provided to the system through the *find* function in the controller.
- Detect termination, which is detected as the *find* function notices that all workers are looking for work.
- Provide methods for starting and stopping threads when a solution is found or termination is detected. The *collect* function is used for stopping threads when a solution is found, while the *find* function can stop threads as termination is detected.
- In the case of search-engines where each worker needs to know the current solution, such as branch and bound. The controller is responsible for making sure that the workers are informed of the current solution. The *collect* function makes sure that each worker receives a copy of the new solution, but sending it as an argument to the workers *stop* function.

When the controller is constructed it initiates all the workers. These workers then eagerly wait for the signal to start exploring the search-tree.

The controller has three major function *next*, *find* and *collect*. The *next* function is a simple function. All it does is to block the calling thread and wait for one of the workers to report a solution using the *collect* function, or termination is detected. The *find* and *collect* function will be explained in greater detail below.

Due to the data flow approach of the implementation the controller is protected by two mutex locks. One is used to make sure that only one worker is currently checking for work or reporting found solutions. The second is used in the *next*

function to pause the starting call of the search until either a solution is found or a termination is detected. That way the application using the system can benefit from the full power of the parallel search but only needs to call a single function, just as with the sequential search-engines it is the *next* function. This make the system easy to use, compared to using a sequential search-engine in GECODE only one extra variable is required to run a parallel search-engine. The extra variable is to tell the parallel search-engine which search-script to use.

When a worker is looking for work it calls the *find* function in the controller. What this function does is to check each worker for work in a round-robin fashion. The system thus uses a variation of the global round-robin scheme described earlier, for load balancing. While checking for work another check is done as well. The second check is to see if all the workers are out of work. In that case then the system is done and termination is detected. The function then signals the mutex which is waiting for the solution, which return null to the calling application.

The *collect* function receives the solutions of the workers. It is protected through mutexes making sure that there is only one worker calling the function. This mutex check is however non-blocking, so if the mutex fails to lock because of another calling thread then the function returns false. This informs the worker that another thread just reported a solution and thus it makes sure the solution is preserved in case subsequent searches will be made. When a solution is collected it is stored in a variable in the controller and the mutex waiting for the solution is signaled. The solution is also passed to all the workers, in case they need to keep a copy of it for example in the case of branch-and-bound search.

6.2.2 Worker

The worker class is the communicating wrapper class for the actual search-engine, also called the search-script. Each worker has its own thread, which is created upon instantiation of the worker object. The worker has functions for stopping, exploring and checking to see if it has work to share. The worker starts exploring when a call to the worker's *explore* function is made, through the controller. Initially the worker will be paused until an explore call has been made. When the search-tree is explored the worker thread spins in a loop, which communicates with the search script and the other workers. The loop does the following:

- if we have work, do a search step and check for pause.
  - if a solution is found, call the collect function in the controller.
- if we don't have work, check for work and then check for pause.
  - if work is found give it to the search-script for exploration.

The pause check is used with the *stop* function, if the worker's search loop should stop it will do so when it checks for pause. A paused worker will resume exploration at the same place it stopped, if the search is started again later.

This makes sure that the work already done in a search will be preserved in the case of subsequent searches.

The workers always maintain their threads when instanced. To completely stop a thread the worker object needs to be destroyed. This makes subsequent calls for search faster as the workers don't have to start new threads, but rather run the ones already available. So maintaining a pool of worker is similar as to maintain a pool of threads.

6.2.3 Search Script

The search script handles the actual exploration. It describes how to explore the search tree, how to share and receive work, etc. The goal is to be able to construct new search scripts which take advantage of the parallel engine without having to deal with thread synchronization and creation. The search script has a few functions, most important are the functions *newRoot*, *searchStep* and *share*. The *newRoot* function is to set the root of an empty search script so the search script receives work. The *searchStep* function does one step of the exploration of the subtree this search script has. Note that the worker class iteratively calls the *searchStep* function to ensure that the search moves forward. The *share* function is responsible for finding work to share with other search scripts.

The search-scripts are in turn implemented in a similar way as the sequential search-engines in the GECODE system, Section 3.3 describes how search-engines are constructed in GECODE. So in a similar fashion the different search-scripts implemented consists of several parts. The first part is a wrapper which starts the rest of the script depending on if re-computation should be used or copying, as well as setting other initialization variables.

The search-scripts can be seen as a sequential search-engine, but instead of doing the search in a loop it does only one step of what a sequential search-engine would do. This and the functions for handling sharing of spaces are the two main differences between a search-script and a sequential search engine in GECODE.

The search-script even reuse parts of the code which was originally designed for the sequential search-engines. Apart from the code for spaces the search-script also use the same stacks as the sequential search-engines. Using these stacks benefit re-computation based backtracking far more than it benefits copying-based backtracking.

As described in Section 3.2 the re-computation based stack maintains more information in the nodes of the stack, compared to what the copying-based stack does. The copying based stack just maintains stacks which it should explore next, which the re-computation based stack maintains all nodes the search has passed so far.

The nodes in the re-computation stack describe the path which has been passed by the search. This is so that spaces easily can be recomputed. To maintain this information each nodes holds two numerical values, the number of the previous path which the search used when it passed this node and the paths available from this node to lower parts of the tree. These values are ordered so that the

leftmost branch of the node has the value 1 and the rightmost value is equal to the total amount of branches available.

This information proved to be quite useful when it comes to sharing work between workers. To cut off a subtree of the search-tree all that is needed is to decrement the maximum number of paths on the node you which to share from. This prevents the re-computation based search-engine to explore down that path, but the path can still be explored by another worker.

This is how sharing is done in the re-computation based search-scripts. The value of the right-most path is sent to the worker requesting work (sharee) along with a copy of the space. The sharee then commits to the right-most alternative on its side. On the sharers side however the maximum number of paths available from the node which the space was shared from is decreased by one, which prevents dual exploration of the same subtree.

6.2.4 Sharing strategy

The Sharing strategy is an optional object which contain rules for the search-engine. The rules describe how and when to share work with other workers. If this object is not passed to the Controller the search-engine is designed to use some default values defined in the search scripts. The sharing strategy provides the option of defining how large a stack should be before the worker will share, where the worker will start looking for work in the stack and in which direction it will look. For example, a worker can look for a node to share at the bottom of the stack and then move up. or it can look from the top and then move down. Both have their respective benefits and drawbacks. If a sharing strategy is not set during worker initialization then the default values for sharing in the search scripts will be used. Currently the default for the search-scripts doing re-computation is to share from the top of the stack, and for the copying search-scripts sharing can only done from the top.

In fact the copying based search-script does not check the sharing strategy at all. This is because the re-computation search scripts needs to keep all passed nodes on the stack. It is thus easy to reserve paths from nodes which have not yet been examined, making re-computation easier for defining different methods of where and how to search for work. How this is done was described in the end of the last section. The search-scripts for copying however only keeps the nodes yet to examine on the stack. The copying based stack is not prepared for removing elements from the middle of the stack. Thus sharing a node which is not on the top of the stack will require restructuring the stack. While this can be implemented, it is costly. Due to time constraints during the end of the development the search scripts for doing copying do not reconstruct the stack. Instead the copying search-scripts only pops the top of the stack and shares the top node with the worker looking for work.

6.3 Implemented Search Scripts

The search scripts mentioned in this section are the search scripts which have been implemented for the search-engine. However, it should be easy to implement other search scripts and I hope that these implemented search scripts will assist anyone who wishes to do so.

6.3.1 Depth First Search

The depth first search script is the most basic search method and it was the first to be implemented in the system. It is implemented both too use re-computation and copying. Where the re-computation based search-script is the more expressive search-script.

6.3.2 Branch and Bound Search

There is a branch-and-bound search script implemented for the system for doing best solution search. The branch-and-bound search-script is similar to the depth first search script in many ways, except for the actual search. The branch-and-bound search script is also a bit more complex than the depth first engine. This is because of the need to constrain store when sharing solutions, and when new solutions have been found. To handle this the branch-and-bound search script records how many entries are on the stack each time a solution is received. When a space is taken from the stack which results in a lower number of entries than the mark then the mark is decremented by one and the space which has just been popped from the stack is to be constrained. When work is received from a search then the injected space is automatically constrained to the current best solution. This requires the newRoot functions of the search-script to make extra checks on the received space, and may result in the space turning out to be failed to begin with.

In the GECODE system the spaces do not contain any general function for constraining. Instead the constrain function is implemented in a more specialized form of space. The space where the problem is modeled. Since the search scripts use standard spaces, they cannot constrain the spaces in the search script. This is instead done in search script which wraps the actual search scripts, just as in the normal sequential search-engine in GECODE. Which is described in Section 3.3. This search script use the specialized version of the spaces and can thus access the constrain functions.

6.4 Possible expansions

As I have mentioned at several points in this section, the implemented architecture is highly adaptive and offers the ability to implement parallel search-engines without forcing the developer to pay too much attention to issues of synchronization and communication of the workers. A natural expansion of this

architecture would be to make it distributable. It would be relatively easy to make a new controller class which handles message passing and cross-network communication between other search-engines. This would make a search-engine which can take advantage of all implemented search script for shared-memory environment, and such a search-engine would also take full advantage of the resources should there be shared memory multiprocessor computers in the network.

Using this platform for constructing parallel search-engine one could create search-engine in which different workers use different search-scripts. Where the search-scripts explore the system in different ways but they can still communicate and take advantage of the work available in the different workers. It would be interesting examine the benefits and drawbacks of this in greater detail.

Chapter 7

Analysis

To prove the functionality of the implemented system a analysis is needed. This section analysis the performance and the functionality of the platform in regard to four different aspects. These are: Correctness, Portability, Overhead and Runtime.

The correctness part aims to show that the platform produces the correct results, the portability subsection is a short discussion of how portable the platform is and which design decisions affect this. The overhead subsection describes the runtime overhead imposed by running search in parallel on the platform, the section tries to show how much overhead can be expected as well as where the sources of overhead are in the system. The last subsection of this section aims to show the speedup obtained from running parallel search using the implemented platform.

7.1 Correctness

One of the most important properties of the implemented system is its correctness. For this reason it must be examined. To see if my implemented platform for parallel search produces the correct results, runs of the search-scripts have been made and examined. These runs examined different things depending on which search-script was currently being examined.

For the Branch and bound search-script the results were compared to the sequential engine. This to see if the search-script produced the same result, that is found the same optimal solution.

For the depth first search search-script runs were made to find all the solutions for the problems. The amount was then compared if the same test was done on the normal sequential execution. Also on some test the solutions were examined to see if they were the same. The problem is that the parallel depth first search engine produces the same results, but in a different order. But examining the test runs to see if the parallel search produced the same result and found all the

solutions showed that the platform and search-script indeed gives the correct result.

Note that in this correctness analysis the tests were compared to the sequential search-engines of GECODE. So if these engines turn out to give the wrong result then so does my parallel platform. Should the normal sequential search-engines prove to give the wrong result, then most likely is it because of a problem with one or more propagators. Propagators which are also used in my parallel platform and thus my platform produces the same result.

7.2 Portability

One of the main reasons for using PThreads was to make the system more portable. The PThread library is available in any POSIX standard OS, where other available methods need extra libraries for implementing parallelism. While the PThread library can differ in its implementation on different operativsystem the functions used for implementing the platform are simple synchronization primitives which even though the implementation differs still produce the same result.

However the platform makes use of a barrier primitive which is implemented in some implementations, while not in others. This is simple to workaround, but it would still require some porting of the code to make the system work on some systems. These systems are, on the other side, not supported by GECODE. Therefor the selection of the PThread library should make the implemented system available on any POSIX standard OS which has implemented barriers and is otherwise supported by the GECODE system.

7.3 Test cases and test environment

In order to examine the performance of the parallel search-engines different test cases are needed. The test cases consists of classical problems for which the solutions are know. In the following two sections; Overhead and Runtime, the problems stated in this section are used.

- magic-square
- golomb
- hamming-codes
- packing
- partition

Each problem is tested using both the re-computation based search-script and the re-computation based search-script. Which gives a total of 8 tests.

The tests are run on a 4 processor machine where each processor has a clock-speed of 900 MHz. The machine has 16 GB of memory available. However, the machine is not dedicated for running the tests, so it is a loaded system.

7.4 Overhead

It would be interesting to know how much overhead the parallel search-engines have compared to a sequential search-engine. Naturally a parallel search-engine needs to do synchronization checks, so knowing it is interesting to know how much overhead is added for these extra checks, compared to running a sequential search-engine.

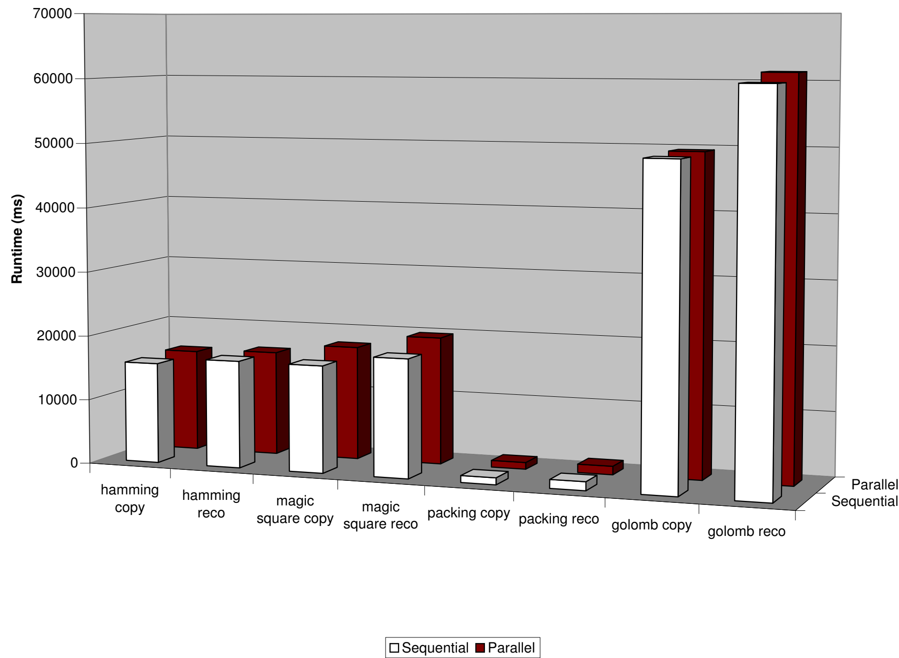

Figure 7.1: A graph illustrating the overhead caused by running the parallel system with one thread compared to running search sequentially

To compare the overhead added by the system the parallel search-engines are run with one thread. To achieve a good estimate of how the system will perform the tests are run several times to gather an average of the runtime. These tests are then compared to similar runs, using the sequential search-engines in GECODE. By comparing the results of the tests we can get an estimate of how much overhead the system gets just from doing the synchronizations and extra controls.

As can be seen in Figure 7.1, the overhead caused by the system due to extra synchronization checks is quite small. In the case where the search-scripts and search-engines use a copying based approach for backtracking the overhead is on average less than half a second of the execution time. When comparing how much of the overhead is of the execution time then the overhead is around 1% of the execution time for all the testcases.

Since the tests are run on a loaded system there are some cases where the test-cases does not show any overhead. However since the tests were run several times and an average was computed these fluctuations in execution time can be ignored.

The small overhead seems reasonable since the system does not have to wait for other threads. By using a data-flow approach instead of a message passing approach the synchronization is quickly done if no other threads are there to interrupt. As more threads are used in the system the overhead for synchronization is bound to go up. But the 1% of can be seen as a minimum overhead caused by the design of the platform.

7.5 Runtime

The main reason for running search in parallel is to achieve lower runtime for problem search. This section will examine how much the runtime of the test problems are decreased as a search is run with more and more degrees of concurrency. To gather the necessary run-times each test-case was run 100 times with 1, 2, 4, 8 and respectively 16 threads. From the runtime of the test runs the average runtime is calculated, as well as the standard deviation of the runtime.

The results of these calculations can be seen in Table 7.1 and Table 7.2. Each table is the calculated data for the runs where Table 7.1 contains the results for when the tests are run with re-computation based backtracking, and Table 7.2 contains the results from using copying based backtracking. The average runtime for the tests can also be seen in graph from in Figure 7.2.

As can be seen in both the tables and in Figure 7.2, running the parallel system results in longer average runtime for all the depth first search problems. There are two possible explanations for this.

- The extra parallel threads are assigned parts of the tree which contain no solutions. Thus the platform only enforces extra synchronization and contention, but the first started thread find the first solution. The solution which happens to be the same solution as the one a sequential search-engine would find. The Magic-squares problem however do sometimes find another solution and finds it faster than a sequential search-engine would do. This can be seen in the tables since the magic-squares problem have a higher standard deviation than the other problems.
- The platform suffers from racing conditions, where the threads are competing for exclusive access to the one source from where they initially can get work form. Unfortunately this source is the first threads stack, the first thread wont share any information until it has produced enough nodes to actually share. This racing condition produces a small overhead as the number of threads are low, but as the number of threads increase this overhead increases with it. The overhead increases faster as the number of threads surpass the number of processors. When this happens not only is the first thread competing for access to its own stack, but also for a processor to work on.

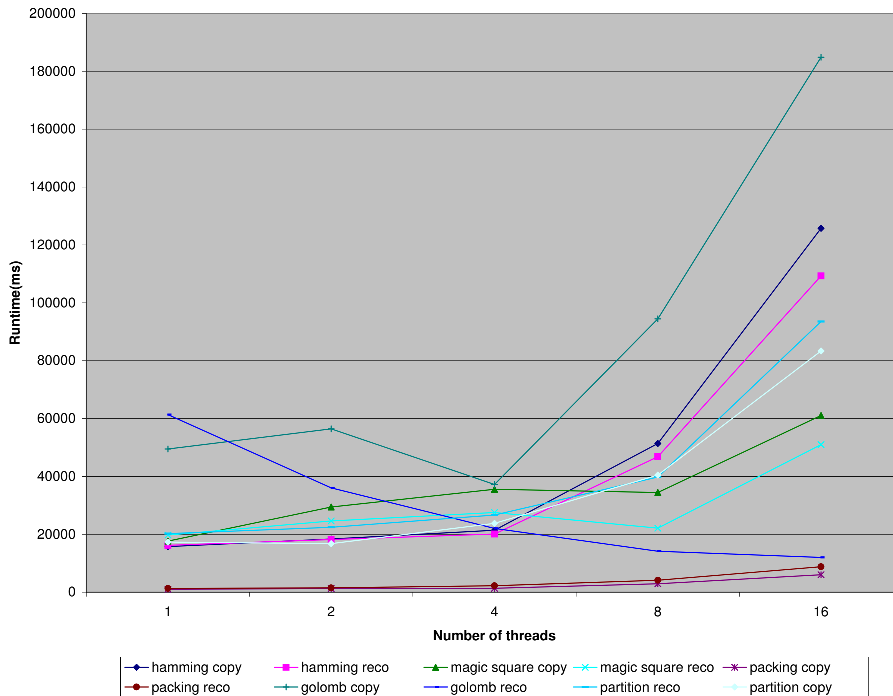

Figure 7.2: A graph illustrating the average speedup when running several threads. The graph is based on the data in Table 7.1 and 7.2.

It is my belief that the poor results are from a combination of these two explanations. There is a racing condition in the system, and the search-trees for the problems which are tested are very narrow.

Threads	1		2		4		8		16	
	mean	std	mean	std	mean	std	mean	std	mean	std
Golomb	61352.8	99.	36039.0	2503.0	22049.9	2944.9	14145.7	5317.5	12004.0	3936.7
Hamming	16221.4	230.6	18195.8	405.4	20086.5	344.0	46798.7	1357.8	109280.0	2995.0
Packing	1298.1	45.4	1493.2	255.3	2242.2	427.9	4141.2	1117.3	8801.5	3396.9
Magic-square	19770.6	40.6	24614.3	6689.6	27564.2	10060.2	22158.6	772.3	50978.9	2512.3
Partition	20248.8	17.7	22448.0	2127.2	26657.1	9579.9	39789.2	18271.4	93509.1	34716.1

Table 7.1: Average runtime and standard deviation for parallel search using re-computation for backtracking

The re-computation based Golomb ruler results show exactly the kind of speedup that is available for doing parallel search. The runtime is reduced as more threads are used in the search. In the beginning the speedup is high, but as more and more threads are being run the performance gain from each thread is reduced. The main reason for this is that there are only 4 processors available in the system. Adding more threads only allows the system to make jumps between different subtrees, which can help find better solutions faster.

The Golomb ruler tests are not as sensitive to the problem with a racing condition. The reason for this is that the platform needs to test for optimality. Meaning it has to examine each and every node in the search-tree, to guarantee

that there are no better solutions. The racing condition is most apparent in the beginning of the execution. Since as the system has found the first solution and the work of finding a better one starts all threads already have nodes to work on. At the end the racing condition comes into play again. However since at the end of a branch and bound search the search-engine mostly discards spaces, the racing condition doesn't affect search in the same degree as in the beginning. Discarding spaces is much cheaper than calculating new ones.

Threads	1		2		4		8		16	
	mean	std	mean	std	mean	std	mean	std	mean	std
Golomb	49485.4	36.5	56465.7	555.3	37244.4	527.7	94455.4	13684.8	184885.5	18439.1
Hamming	15766.3	72.0	18408.0	477.5	21387.4	323.5	51368.0	1363.5	125744.7	3174.0
Packing	1043.7	44.9	1243.5	91.3	1344.6	155.1	2903.9	1362.0	6032.5	2070.4
Magic-square	17665.0	19.6	29433.3	7382.6	35549.3	11914.82	34444.8	18151.5	61065.3	3936.7
Partition	17313.7	283.3	16767.3	3044.1	23802.4	9268.3	40462.4	14693.9	83345.1	28048.8

Table 7.2: Average runtime and standard deviation for parallel search using copying for backtracking

In contrast to the re-computation based Golomb ruler test, for the copying based test the runtime increased in all steps except for one. From figure 7.2 and table 7.2 we can also see that the speedup benefit is not a linear function. This is alarming as the average runtime should be linear to the amount of threads. A possible theory for the sudden speed increase when running 4 threads, could be that running a search with 4 threads on this search-tree gives a good chance to discover the optimal solution early on. This however should also be the case for when running more threads than 4. As can be seen in both the graph and the table, the runtime for running more than 4 threads increases greatly. Both comparing to running 2 and 4 threads on the same test. Another explanation could be the bugs in the GECODE system prior to version 1.0. Apparently prior to version 1.0 of the GECODE system, there were some errors in the functions for doing copying-based search. These tests were run using a earlier version of GECODE and due to the parallel execution the effects of these bugs could have a great effect on this particular test.

Finally this analysis section ends with with a table showing the speed-up for all the tests compared to the original sequential runtime. These figures can be seen in Table 7.3

Speedup	Sequential	1	2	4	8	16
Packing Recomputation	1.0	0.96	1.10	1.66	3.06	6.50
Packing Copying	1.0	0.99	1.18	1.27	2.75	5.70
Hamming Recomputation	1.0	0.98	1.09	1.21	2.82	6.58
Hamming Copying	1.0	1.01	1.18	1.37	3.30	8.07
Magic-square Recomputation	1.0	1.08	1.35	1.51	1.21	2.79
Magic-square Copying	1.0	1.07	1.78	2.15	2.08	3.69
Golomb Recomputation	1.0	1.02	0.60	0.37	0.24	0.20
Golomb Copying	1.0	1.01	1.15	0.76	1.92	3.76
Partition Re-computation	1.0	1.00	0.97	1.39	2.36	4.86
Partition Copying	1.0	1.01	1.12	1.33	1.98	4.66

Table 7.3: The speedup for all of the tests

7.6 Improvements

The above sections state that there is still room for improvement in the parallel platform. By re-compiling and adjusting the platform to use the latest version of GECODE, I believe that the problems with the non-linear runtime of the copying based tests could be solved.

The runtime could be significantly improved if the racing condition could be removed. There are several possible solutions for removing the racing condition:

- Checking if a worker has work to share is in fact a read-only operation on the workers stack. As it is implemented in the system this is done in the sharing function. That is the worker locks the mutex which controls the stack just to check for more data. By moving this check out into its own function and only lock the mutex if there truly is data to share, the overhead caused by the racing condition could be severely reduced.
- Currently the function for finding work goes in a strict order over the worker when looking for work. It first starts to check with worker number 1, then 2, etc. This strict order places a high load on worker number 1, especially in the start of the search, when it is the only worker who actually has work to share. By changing the find-function to check with all the workers, but in a random order. The amount of threads disturbing worker number 1 is reduced, and thus the overhead could be reduced as well.
- A solution which is a bit more advanced is to divide the stack of each worker into two parts. One from which data can be shared without locking access to the other part of the stack, and one from which data is shared only in worst case scenarios. Such a splitting of the search stack has been done using a cut-off depth[6]. This cut-off depth is a depth in the stack where the owner cannot go above without locking the stack, and the thread looking for work cannot go below. A similar order could be introduced to the platform to reduce the overhead caused by the racing condition.
- In the current implementation the controller checks all the workers for work, instead of doing this the platform could be modified so that the workers report to the controller when they have work to share. The controller would then keep an array of workers which have work to share. This would improve the performance of the find-function as well as ensuring that the workers are not disturbed when they do not have enough work to share. Implementing this requires allot of reworking of the platform. It also requires the developer of a parallel search-script to keep track of when there is work to share, and to report this to the worker from the search-script. While this solution could improve the performance of the system, I believe that it cannot be implemented in a way which would keep the separation between the search-scripts and workers in the way I have tried to achieve.

I believe that by implementing the first three possible solutions for the racing-condition, the performance of the system will increase tremendously.

Chapter 8

Conclusion

This thesis has covered the concepts of implementing a parallel search-engine for a constraint programming system. It should be apparent to the reader that the implementation of a parallel search-engine is somewhat of a balance act. There are alot of decisions which need to be made about the nature of how threads communicate and distribute work.

I've developed a platform for parallel search on which different search-engines easily could be built. I have also implemented two types of search-engines on this platform; depth-first search and branch and bound. Each with different memory handling techniques; re-computation and copying. These search-engines can act as examples of how to implement other search-engines using the same platform for thread handling and sharing.

I aimed at making the platform as generic as possible. The goal was to allow several types of search-engines to use the same structure. However this platform cannot accommodate for more specialized search techniques, such as using optimistic branch and bound for example. In some cases the problem could be how GECODE handles the models of problems, in other cases more specialized communication was required. This platform however should allow the vast majority the different types of search-engines to use the parallel capacity offered.

The results from the analysis of the platform I have implemented show poor performance for all but one test. This test however shows promising speedup and it is my belief that the platform can be modified to perform to a degree where it is beneficial to use it.

Given the poor results of this platform I am still confident that its performance numbers can be improved. Even though this implementation does not perform well it proves that creating a generic platform for doing parallel search is possible, and an implementation like this can achieve speedup. As the tests show; the branch-and-bound engine using re-computation for backtracking reduced the runtime to 20 percent of the original sequential runtime when running 16 threads for the computation.

Bibliography

[1] Anath Grama and Vipin Kumar. State of the art in parallel search techniques for discrete optimization problems. *International Journal of Parallel Programming*, 11(1), 1999.

[2] T.K. Ralphs L. Ladnyi and L.E. Trotter Jr. Branch, cut, and price: Sequential and parallel. 2001.

[3] Laurent Perron. Search procedures and parallelism in constraint programming. In Joxan Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360, Alexandria, USA, October 1999. Springer-Verlag.

[4] Laurent Perron. Practical paralleism in constraint programming. 2002.

[5] Steven Prestwich and Shyam Mudambi. Improved branch and bound in constraint logic programming. In Ugo Montanari and Francesca Rossi, editors, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes in Computer Science*, pages 533–548, Cassis, France, September 1995. Springer-Verlag.

[6] V. Nageshwara Rao and Vipin Kumar. Parallel depth first search. Part I. Implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.

[7] Christian Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, November 1999. The MIT Press.

[8] Christian Schulte. Parallel search made simple. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, Franois Laburthe, Eric Monfroy, Tobias Mller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, number TRA9/00, pages 41–57, 55 Science Drive 2, Singapore 117599, September 2000.

[9] Christian Schulte. *Programming Constraint Services*. Doctoral dissertation, Universitt des Saarlandes, Naturwissenschaftlich-Technische Fakultt I, Fachrichtung Informatik, Saarbrcken, Germany, 2000.

[10] Guido Tack and Didier Le Botlan. Compositional abstractions for search factories. In Peter Van Roy, editor, *International Mozart/Oz Conference*, volume 3389 of *LNCS*, pages 211–232. Springer Verlag, 2004.

[11] Ananth Y. Grama Vipin Kumar and Vempaty Nageshwara Rao. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1), 1994.

[12] T. K. Ralphs Y. Xu and M. J. Saltzman. Alps: A framework for implementing parallel search algorithms. In *The Proceedings of the Ninth INFORMS Computing Society Conference*, 2004.