KTH Information and Communication Technology

# Parallel Portfolio Search for Gecode

ANTON FROM

Master's Thesis. Stockholm, July 2, 2015.
School of Information and Communication Technology
KTH Royal Institute of Technology

Supervisor: Roberto Castañeda Lozano
Examiner: Christian Schulte

TRITA-ICT-EX-2015:75

# Abstract

Constraint programming is used to solve hard combinatorial problems in a variety of domains, such as scheduling, networks and bioinformatics. Search for solving these problems in constraint programming is brittle and even slight variations in the problem data or search heuristic used can dramatically affect the runtime. But using portfolios of search engines on several variants of a problem by adding randomness to the heuristic used has been proved to counter this problem. A portfolio is defined as a collection of assets that combined gives it a desired return and risk. Unfortunately not all constraint programming systems have implementations of portfolio search, such as Gecode. Therefore were two portfolio search prototypes, sequential and parallel, designed and implemented for Gecode. The design is not system dependent and could easily be implemented in other constraint programming systems.

The design and implementation is tested by both validity and performance tests to ensure its soundness. Validity is tested by finding all possible solutions on a moderately difficult combinatorial problem known as the N-Queens problem. Performance is tested by finding the first solution on a very difficult combinatorial problem known as the Latin Square Completion problem with different numbers of search engines. To compare against the same validity and performance tests were run with just one search engine.

Results show that the design and implementation of portfolio search is sound. The parallel variant of portfolio search finds solutions faster with more search engines and outperforms the sequential variant. The sequential variant finds solutions about as fast as running with just one search engine.

Successfully designing and implementing portfolio search in Gecode will help researchers and companies who use Gecode to save both time and money as they are now able to find better solutions faster by using this portfolio search. It may also contribute to the research within this area and the continued development of Gecode.

**Keywords** Constraint programming, Gecode, Parallel portfolio search

# Referat

## Parallel Portföljsökning för Gecode

Villkorsprogrammering används till att lösa svåra kombinatoriska problem inom en mängd områden, såsom schemaläggning, nätverk och bioinformatik. Men sökning för att lösa dessa problem inom villkorsprogrammering är skör och även små variationer i problemets data eller använd sökheuristik kan dramatiskt påverka körtiden. Men att använda portföljer av sökmotorer på flera varianter av ett problem genom att införa slumpmässighet i sökheuristiken har bevisats kontra detta problem. En poftfölj är definierad som en samling tillgångar som kombinerad ger den en önskvärd avkastning och risk. Olyckligtvis så har inte alla villkorsprogrammeringssystem implementationer av portföljsökning, såsom Gecode. Därför designades och implementerades två portföljsökningsprototyper, sekventiell och parallell, för Gecode. Designen är inte systemberoende och kan enkelt implementeras i andra villkorsprogrammeringssystem.

Designen och implementationen är testad av både validitets och prestandatest för att försäkra dess sundhet. Validiteten testas genom att finna alla möjliga lösningar för ett lagom svårt kombinatoriskt problem känt som N-Queens problemet. Prestandan testas genom att finna första lösningen för ett väldigt svårt kombinatoriskt problem känt som Latin Square completion problemet med olika många sökmotorer. För att jämföra mot så kör en ensam sökmotor samma validitets och prestandatest.

Resultaten visar att designen och implementationen av portföljsökning är sund. Den parallella varianten av portföljsökning hittar lösningar snabbare med fler sökmotorer och överträffar den sekventiella varianten. Den sekventiella varianten hittar lösningar ungefär lika snabbt som en ensam sökmotor.

Att framgångsrikt designa och implementera portföljsökning i Gecode kommer hjälpa forskare och företag som använder Gecode att spara både tid och pengar när de nu kan hitta bättre lösningar snabbare genom att använda denna portföljsökning. Det kan också bidra till forskningen inom det här området och den fortsatta utvecklingen av Gecode.

**Nyckelord** Villkorsprogrammering, Gecode, Parallel portföljsökning

# Contents

**List of Figures**

**List of Tables**

**1 Introduction** **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Proposed Solution . . . 2
1.4 Methodology . . . 3
1.5 Limitations . . . 3
1.6 Ethics and Sustainability . . . 4
1.7 Outline . . . 5

**2 Background** **7**
2.1 Constraint Programming . . . 7
2.2 Search in Constraint Programming . . . 9
2.3 Portfolio Search . . . 10
2.4 Gecode . . . 11
2.4.1 Search in Gecode . . . 11
2.4.2 Stop objects in Gecode . . . 11
2.5 Case Studies . . . 12
2.5.1 Case Study: N-Queens . . . 12
2.5.2 Case Study: Latin Square . . . 14

**3 Designing Portfolio Search** **17**
3.1 Design of Sequential PBS . . . 18
3.1.1 PBS:next() . . . 19
3.2 Design of parallel PBS . . . 19
3.2.1 Parallel PBS:next() . . . 21
3.2.2 Parallel Communication Protocol . . . 22

**4 Implementing Portfolio Search** **25**
4.1 Implementation of Sequential PBS . . . 25
4.1.1 PBS:next() with 2 Search Engines for One Solution . . . 26

4.1.2 PBS:next() with 2 Search Engines for All Solutions . . . . . . 26
4.1.3 PBS:next() with N Search Engines . . . . . . . . . . . . . . . 28
4.1.4 Global Stop Object for PBS . . . . . . . . . . . . . . . . . . . 28
4.1.5 Other Methods in PBS . . . . . . . . . . . . . . . . . . . . . . 30
4.2 Implementation of Parallel PBS . . . . . . . . . . . . . . . . . . . . . . 31
4.2.1 Run Wrapper . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2.2 PBS:next() with 2 Search Engines for One Solution . . . . . . 33
4.2.3 Control Stop Object for Search Engines . . . . . . . . . . . . 33
4.2.4 Parallel Queue for Solutions . . . . . . . . . . . . . . . . . . . 34
4.2.5 PBS:next() with 2 Search Engines for All Solutions . . . . . . 36
4.2.6 PBS:next() with N Search Engines . . . . . . . . . . . . . . . 37
4.2.7 Global Stop Object for PBS . . . . . . . . . . . . . . . . . . . 41
4.2.8 Other Methods in PBS . . . . . . . . . . . . . . . . . . . . . . 42
4.2.9 Avoiding Deadlocks . . . . . . . . . . . . . . . . . . . . . . . . 42

**5 Experimental Evaluation 45**
5.1 Case Study: N-Queens . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.2 Case Study: Latin Square . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.2.1 Measuring Overhead . . . . . . . . . . . . . . . . . . . . . . . . 48
5.2.2 Measuring Performance . . . . . . . . . . . . . . . . . . . . . . 53

**6 Conclusions 63**
6.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**A Case Study Extras 65**
A.1 Server Specifications . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
A.2 Branching Implementations & Tie-Breaking . . . . . . . . . . . . . . 66

**B Source Code of Portfolio Search 69**
B.1 Source Code for Sequential Portfolio Search . . . . . . . . . . . . . . 69
B.2 Source Code for Parallel Portfolio Search . . . . . . . . . . . . . . . 74

**Bibliography 83**

# List of Figures

2.1 Figures showing queen movement & solution for 8-Queens. . . . . . . . 13
2.2 Small Latin Square example. . . . . . . . . . . . . . . . . . . . . . . . 14

3.1 Diagram illustrating abstraction layers & flow of sequential PBS. . . . . 18
3.2 Diagram illustrating abstraction layers & flow of parallel PBS. . . . . . 20
3.3 Diagram illustrating the communication protocol of parallel PBS. . . . . 23

4.1 Code segment of base case of sequential `PBS:next()`. . . . . . . . . . . . 27
4.2 Code segment of second iteration of sequential `PBS:next()`. . . . . . . . 28
4.3 Code segment of N-search engines in sequential `PBS:next()`. . . . . . . 29
4.4 Code segment of final sequential `PBS:next()` with global stop object. . 31
4.5 Code segment of first version of the `run_wrapper` class. . . . . . . . . . 32
4.6 Code segment of base case of parallel `PBS:next()`. . . . . . . . . . . . . 33
4.7 Code segment of base case of the modified stop object class. . . . . . . . 35
4.8 Code segment of second parallel `PBS:next()`, with commands. . . . . . 35
4.9 Code segment of second version of the `run_wrapper` class with solution queue. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.10 Code segment of parallel `PBS:next()` for all solutions. . . . . . . . . . . 38
4.11 Code segment of second version of the `run_wrapper` class. . . . . . . . . 39
4.12 Code segment of N-search engines in parallel `PBS:next()`. . . . . . . . . 40
4.13 Code segment of the modified stop object class with global stop object. 41
4.14 Code segment of final parallel `PBS:next()` with global stop object. . . . 42

5.1 Graphs showing execution times of first group measuring overhead . . . 50
5.2 Graphs showing execution times of second group measuring overhead . . 51
5.3 Graphs showing execution times of third group measuring overhead . . 52
5.4 Graphs showing execution times for $N$=15 measuring performance . . . 56
5.5 Graphs showing execution times for $N$=20 measuring performance . . . 57
5.6 Graph showing results after 1000 runs with parallel PBS . . . . . . . . . 60

A.1 Server specifications. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
A.2 Code segment of the original branching implementation. . . . . . . . . . 66
A.3 Code segment of the modified branching implementation. . . . . . . . . 67
A.4 Code segment of the tie-breaking limit functions. . . . . . . . . . . . . . 68

# List of Tables

2.1 Table showing number of solutions for $N$-Queens puzzle . . . . . . . . . 13

5.1 The table shows the test setup for the $N$-Queens puzzle . . . . . . . . . 46

5.2 The table shows the test results for the $N$-Queens puzzle . . . . . . . . 47
5.3 The table shows the test setup for overhead measurement on the Latin Square problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.4 Table showing problem instances for overhead measurement on Latin Square problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.5 Table showing problem instances & result for choosing hard problems . 54
5.6 Table showing test setup for performance measurement on Latin Square problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.7 Table showing test setup for the additional experiment . . . . . . . . . . 59
5.8 Table showing min & max values from results for the additional experiment 60

# Chapter 1

# Introduction

This chapter gives a first presentation of this thesis and provides important information about the thesis itself. Section 1.1 presents a short background and motivation for the thesis and section 1.2 presents the problem statement. Furthermore, it presents a proposed solution to the problem statement in section 1.3 followed by the methodology description in section 1.4. Then it states the limitations in section 1.5. It also presents section 1.6 that covers the ethics and sustainability for this thesis and finally gives an outline of the thesis in section 1.7.

## 1.1 Motivation

Constraint programming is a powerful paradigm that is used to solve combinatorial search problems and is currently used with success in a variety of domains, such as scheduling and bioinformatics [1]. But search for solving these constraint problems in constraint programming today is brittle. Even slight variations in either the problem data or the heuristic used during search can have a huge impact on the time it takes for search to find a solution [2]. However, there exist known techniques to exploit this brittleness, such as randomized restart-based search or portfolio search. This thesis is concerned with *portfolio search*.

A portfolio is defined as a collection of assets that combined gives it a desired return and risk [3, 4]. The idea of portfolio search is to run a whole portfolio of *search engines* (which are implementations of search) for several variants of a problem, either in parallel or in a round-robin fashion. The problem variants can be obtained by adding a slight amount of randomness in the search heuristic or even using different heuristics. The first solution that is found by a search engine is reported as the solution of the portfolio. Using a portfolio of search engines combines their individual strengths while covering for their individual weaknesses to better tackle constraint problems and find solutions faster than by using any of the search engines by itself [5].

More and more specialized heuristics are developed in constraint programming to solve specific types of problems even better than before. But because of the

specializations they also become more brittle. So the optimal solution would be to be able to run a lot of search engines with different heuristics on a single problem to counter the brittleness as well as taking advantage of the emerge of multi-core processors during the last decade. However, not all constraint programming systems have implementations of portfolio search. This forces the user[1] to either construct his or her own crude version of a portfolio search or manually run each search engine on the same problem which is cumbersome. One such constraint programming system lacking an implementation of portfolio search is Gecode[2]. Gecode is an open, free, efficient constraint solving toolkit built on C++ [6]. For further explanation on the reason to why Gecode was chosen, see section 1.5, and for a more thorough explanation on what Gecode is, see section 2.4 in chapter 2.

Addressing these problems would make it easier to run large sets of search engines with different heuristics on a single constraint problem. It would take advantage of the multi-core architecture in today's processors and also be able to decrease the runtime. The decreased runtime saves time which in turn saves money. Take a shipping company as an example which need to fill their shipping containers with as many packages as possible. With decreased runtime the company can use portfolio search (instead of only one search engine) to find more solutions before the deadline and then pick the best of the found solutions. This saves money for the company as they can now fill their containers better than before.

Therefore, developing portfolio search with which the user can easily perform these tasks will save both time and money as well as contribute to the research in this area.

## 1.2 Problem Statement

As motivated in section 1.1 an exploration of portfolio search and its uses is needed. This encompasses researching, designing, implementing and testing a prototype portfolio search engine, both for the comparison and evaluation against other constraint solving techniques as well as for the general research in this area and related fields. More specifically the problem statement for this thesis is to find a satisfactory answer to the following question:

*What is a good design and implementation of portfolio-based search in Gecode?*

## 1.3 Proposed Solution

The object of this master's thesis is to research, design, implement and test a prototype portfolio search as a response to the problem defined in section 1.2. The

[1] A user is a person that uses Gecode, such as researchers, developers, companies and hobbyists.

[2] Verified by Christian Schulte when discussing the thesis outline in the very beginning. He is one of the main developers of Gecode

solution is presented in two variants, one sequential and one parallel:

- The sequential variant is based on a round-robin architecture to give each search engine in the portfolio the same amount of runtime in each runtime cycle.
- The parallel variant is based on a master-slave architecture where each search engine in the portfolio runs in its own slave thread while the controlling part of the portfolio runs in the master thread and synchronizes them.

## 1.4 Methodology

First a thorough research of the topic is performed to give a good foundation for the design of portfolio search. After the design is finished the implementation of the portfolio is performed. This is all done to explore and evaluate different design and implementation choices in order to find an answer to the question stated in the Problem Statement section.

The chosen approach is iterative development that starts with the smallest most basic problem. For each iteration additional functionality and features are added on top of the work from the previous iteration. This ensures that the work is stable, functional and debugged before the next iteration begins. It also helps keeping the complexity in check by only adding a few new functions and features for each iteration. The sequential variant is designed and implemented first so that the more complex parallel variant can reuse and build onto common parts of the design and implementation. After both prototypes are finished and stable the verifications and experiments are performed. This is done in the form of two case studies.

The first case study tests the validity by verifying that no solutions are ever lost. The setup is that the prototypes run with several search engines that are to find all possible solutions to a chosen problem. Then all found solutions are counted and compared to the solutions found by running one search engine without portfolio search to ensure that no solutions were lost or invalid.

The second case study tests the performance of the prototypes by comparing their execution times to the execution time of running with one search engine without portfolio search. The setup is that the prototypes run with different amounts of search engines and are to find the first solution. This case study is divided into several parts which use different settings and problems to focus on different aspects of the performance. Some of the performance experiments are similar to earlier experiments performed by other authors such as [7] in order to compare the results.

## 1.5 Limitations

Because constraint programming is a broad area with many different libraries and languages implementing it, the scope needs to be decreased down into a realistic manageable size. Therefore, this thesis only focuses on one constraint programming

system. The design of portfolio search is made on a higher abstraction layer and is thus not completely system dependent[3]. But the implementation of this portfolio search is specific to the chosen system.

The chosen constraint programming system for this thesis is Gecode. Gecode already has randomized restart-based search implemented but does not have an implementation of portfolio-based search. This makes Gecode suited for this thesis as experiments comparing the efficiency of randomized restart-based search and portfolio-based search could be performed in the future, although this thesis does not do it due to time constraints.

This thesis does not develop a fully tested and error-free portfolio-based search, but rather aims to provide two prototypical implementations. It does not do comparisons with randomized restart-based search. It also does not do extensive testing and experiments on the prototypes.

It does perform experiments validating the soundness of the design and implementation. It also does perform initial experiments for measuring the performance compared to that of a single search engine.

## 1.6 Ethics and Sustainability

This thesis strives to adhere to the IEEE Code of Ethics [8]. It gives credit for contributions of others and does not claim others work as its own. The author rejects all forms of bribery and is honest about stating claims based on the available data. The author tries to make decisions consistent with the safety, health and welfare of the public, even though it is difficult to know how the research in this thesis could endanger anyone.

Researching, designing and implementing portfolio-based search in Gecode is important from a sustainability point of view. Researchers and companies that use Gecode to solve their hard combinatorial problems face the problem with search being brittle and sometimes does not manage to find a solution within a reasonable amount of time. This slows down research progress and prevents companies to improve their business in terms of optimizing shipping space, scheduling and vehicle routing to name a few, which are all very resource and time consuming. To provide these Gecode users with a tool to find solutions to their problems in a more efficient and time-saving manner benefits not just them or the company they work for, but all of us. Researchers might be able to find solutions to global environment problems faster, find new cures to diseases or improve the way Internet routing works. Companies can save both time and money while at the same time decrease the impact on the global climate by filling out their shipping space more efficiently, schedule their production better or calculate more efficient routes for their vehicles.

[3]Although the system needs to use search engine abstractions in order to port the design straight off.

## 1.7 Outline

The rest of the thesis is organized as follows:

Chapter 2 gives the reader the theoretical background needed to understand the contents of this thesis.

Chapter 3 is about the design of portfolio search and which design decisions were made and why. The design for portfolio search is system independent and should be possible to implement with constraint programming in open constraint solvers that uses search engine abstractions.

Chapter 4 is about the implementation of portfolio search and goes deeper into the actual coding part and problems that occurred. The implementation for portfolio search is system dependent as it is done in Gecode.

Chapter 5 presents the experimental evaluation and discusses the results and analysis of each case study as well as other interesting discoveries.

Chapter 6 presents the conclusion of this thesis and also describes the future work.

Appendix A contains coding specifics for the case studies.

Appendix B contains the complete source code for the implementations of the sequential and parallel variants of portfolio search.

# Chapter 2

# Background

This Chapter gives the reader a basic understanding of the area of the thesis. It first introduces the reader to constraint programming in section 2.1, and then continues with explaining how search works in constraint programming in section 2.2. After this the concept of portfolio search is covered in section 2.3 and then Gecode as an open constraint solver is explained in section 2.4. Lastly the backgrounds for the different experiments and case studies are covered in section 2.5.

## 2.1 Constraint Programming

Constraint programming is a powerful paradigm that is used to solve combinatorial search problems [1]. Constraint programming is currently used with success in a variety of domains. Examples are found in scheduling, vehicle routing, bioinformatics and networks [1]. Constraint programming in its most basic form is made up of two parts: a declarative part where the user states the constraints of the problem, and a solving part where the user gives a general purpose constraint solver the declared problem and lets it find the solutions. This two-part structure makes constraint programming different from regular programming where the user cannot declare constraints and relations between variables and also has to code the solving part of the program. Because of this constraint programming can be seen as a form of declarative programming as opposed to the more regular imperative and functional programming languages.

Variables in constraint programming initially consist of sets of possible values, so called *domains* [9]. When constraints are placed upon these variables as relations between them a constraint satisfaction problem (CSP) is created. As constraints are just relations that must hold in a solution [1], the solutions to these CSP's are those which have assigned a single value to each variable and do not violate any of the constraints of the problem. Therefore, a CSP is a container for all the variables and constraints and is used to model the problem. A problem model is an implementation of the CSP in a constraint programming system. The possible solutions to a problem model exist inside a *solution space* (also known as *search*

*space*). A solution space is often illustrated as a tree structure (hereinafter called search tree) where the top node contains all variables with their initial sets of possible values along with the implementation of the constraints. Each new node down the search tree contains a smaller sub-problem where the set of values of one or more variables have been reduced. Each leaf node in the search tree either contains a solution or a dead end. When the problem model is given to the constraint solver it searches after solutions in the search tree and returns what was requested if the constraint solver finds it. A request could be to return the first found solution, all found solutions or the best found solution, if any exists.

To find the requested solution(s) the constraint solver uses *constraint propagation* and *search* (more on *search* in section 2.2). Constraint propagation is when the constraint solver applies the constraints over the model's variables and prunes their sets of values by removing values that violate at least one constraint[9, 10]. It is called constraint propagation because it propagates the constraints over the variables in the problem model. When a value is removed due to constraint propagation the change can propagate to other variables, triggering additional removal (even in already pruned variables). Therefore, if any values were removed during propagation, the constraint solver will perform propagation again over all variables until no more values can be removed. After constraint propagation the constraint solver has reached one of four states:

1. At least one value was removed during the constraint propagation, therefore another constraint propagation is needed.
2. All variables have exactly one valid value left, therefore a solution is found.
3. At least one of the variables has no values left, therefore no solution exist.
4. No values were removed during the constraint propagation and at least one variable still has at least two possible values.

States 1-3 are illustrated in the two examples below. State 4 needs to use *search* to open up for more propagation, which is covered in section 2.2.

*Example 1:* Find a solution for the equation $x < y$ where $0 < x$ and $y < 3$. First the value sets of x and y are sufficiently large[1] {-100, ..., 100}. The three constraints are *a*) x is greater than 0; *b*) y is less than 3; and *c*) y is greater than x. Propagating the first constraint prunes the value set of x to {1, ..., 100}. Propagating constraint two limits the value set of y to {-100, ..., 2} in the same way. Now propagating the third constraint on variable x gives $x = \{1\}$ and then propagating the same constraint on variable y gives $y = \{2\}$. A solution is found!

*Example 2:* Find a solution for the equation $x = y$ where $0 < x$ and $y < 0$. First the value sets of x and y are sufficiently large {-100, ..., 100}. The three constraints are *a*) x is greater than 0; *b*) y is less than 0; and *c*) y is equal to x. Propagating the first constraint prunes the value set of x to {1, ..., 100}. Propagating constraint two limits the value set of y to {-100, ..., -1} in the same way. Now propagating

[1] The size limit of the domain is usually specified in the problem model.

the third constraint on variable x gives x = { }. Variable x has no values left and therefore no solution exists.

## 2.2 Search in Constraint Programming

When propagation alone is not enough to solve a constraint problem the constraint solver must use *search*. Search in constraint programming systems has two important dimensions. The first dimension is how the search tree should be described. This is usually done with *branching* or *labeling* [9]. The second dimension is how the search tree should be explored. This is usually achieved by a *search strategy* or *exploration strategy* [9]. Two often used search strategies are depth-first search (DFS) and branch-and-bound (BAB).

Branching is done when propagation cannot remove any more values. The constraint solver then decides which variable to branch on by applying the chosen search strategy. When a variable is chosen it is assigned one of its remaining values (randomly or by some selection rule). The removal of values during branching opens up for more propagation. When the constraint solver branches on a variable it is important that the state which it branches upon is saved so it can *backtrack* if no solution should exist in the chosen branch of the search tree.

The constraint solver runs propagation until no more values can be removed and then branching to open up for even more propagation. It continues in this pattern until one of three states is found:

1. All variables have been assigned a value that does not violate any of the posted constraints. A solution has been found.
2. One or more variables have no values left. This implies that a previous branching was incorrect which calls for a *restart* or *backtracking*.
3. The entire search tree is exhausted and no solution has been found. This proves that no solution exists for the problem.

If the constraint solver reaches a dead end it can either restart from the top of the search tree or backtrack to the state where the last branching was made. This thesis will not explain restart and backtracking further but the curious reader is directed to [11] for an extensive coverage of backtracking as well as restart. Below is an example illustrating how branching works.

*Example 3:* Find a solution for the equation 2x = y where $0 < x < 10$ and $0 < y < 10$. First the value sets of x and y are sufficiently large {-100, ..., 100}. The five constraints are *a*) x is greater than 0; *b*) x is less than 10; *c*) y is greater than 0; *d*) y is less than 10; and *e*) y is twice the size of x. Propagating the first constraint prunes the value set of x to {1, ..., 100} and the second constraint limits it even further down to x = {1, 2, 3, 4, 5, 6, 7, 8, 9}. Propagating constraint three and four limits the value set of y to {1, 2, 3, 4, 5, 6, 7, 8, 9} in the same way. Propagating the fifth constraint prunes all odd values from y, leaving it with y = {2, 4, 6, 8}, and prunes all values that doubled are greater than 8 (or 9, depending on which variable is pruned first) from x, leaving it with x = {1, 2, 3, 4}.

Now the constraint solver can no longer remove any more values by propagating the constraints and therefore needs to branch on a variable. By selecting variable x and assigning it to the value of $\{1\}$ the constraint solver is branching on the assignment of x = $\{1\}$. If this later turns out to lead to a dead end and backtracks, it will then try to assign one of the other values $\{2, 3, 4\}$ to x. After branching the constraint solver moves on and once again performs propagation. Now with x = $\{1\}$ propagating the fifth constraint leaves y with $\{2\}$. As the values of x and y still adhere to the constraints and also only have one value left in their sets a solution has been found!

## 2.3 Portfolio Search

*Portfolio search* is best understood by first defining and explaining what a *portfolio* is through a finance analogy. A portfolio is a collection of assets that each has its own return and risk and the portfolio aims to maximize the return while minimizing the risk [3, 4]. The risk is often measured as the standard deviation of the return. This is the fundamental functionality of a portfolio in any given situation. Using a portfolio gives the investor the possibility to tailor the return and risk to his or hers desires instead of putting all resources on one single asset [2]. Each added asset may influence the return and risk of the portfolio depending on the size of the asset and its own return and risk. When building a portfolio the investor needs to decide on three factors *a*) the risk tolerance; *b*) the time frame; and *c*) the investment objectives. Only then can a suitable portfolio be designed.

In the same way in constraint programming *portfolio search* aims to maximize the return while minimizing the risk, where return is the inverse of its execution time (the shorter the better) and its risk is the standard deviation of the return. Portfolio search is used to run several instances of a constraint solver, each with different heuristics and/or different random seeds, on the same problem. This way portfolio search is able to explore the search tree in different ways at the same time and the probability to find a solution faster is increased. There are two main types of portfolio search [7], *a*) running each instance of the constraint solver in parallel on a multi-core machine; and *b*) running them interleaved on a single-core machine. Both types are designed, implemented and evaluated in this thesis.

The most common way to select an algorithm for a constraint solver is to have several algorithms run over a given distribution of problems and take the algorithm which had the lowest average runtime [5]. This approach focuses on having just one instance of a constraint solver and does not take portfolio search into consideration. It also does not favor the algorithms with poor average performance, even if they present excellent performance on particular problems. But with portfolio search it becomes possible to use these poor-on-average algorithms and have them cover up each other's weaknesses and even outperform the best-on-average algorithms [5]. It has also been shown with concrete empirical results in [12] that even running multiple instances of the same algorithm just given different random seeds on a given

hard computational problem could improve performance significantly. It has also been shown in [7] that portfolio search works especially well for the more risk-taking algorithms.

## 2.4 Gecode

Gecode is an open, free, efficient constraint solving toolkit built on C++ [6]. Gecode implements problem models as spaces which contain the variables, propagators and branchers of the problem [13]. A propagator is the implementation of a constraint and a brancher is the implementation of a branching describing the search tree. The constraint solver takes the implemented problem model and then finds the solutions (if any exist). But for most problems only propagation is rarely enough which calls for an implementation of search with different search strategies.

### 2.4.1 Search in Gecode

Search in Gecode is implemented as and performed by *search engines*. Gecode has implementations of both depth-first search (DFS) and branch-and-bound (BAB) as search engines [14]. Both DFS and BAB in Gecode use backtracking. To use search in Gecode the user must first create an object of the problem model which is then given to the chosen search engine during its initialization. After the search engine has been created the user can start using it to find solutions.

All search engines in Gecode are derived from the same base class *EngineBase* which provides them all with the same interface. Therefore search engines in Gecode have (at least) the following methods [14]:

- next(): A method that finds the next solution and returns it to the user. If no more solutions exist NULL is returned.
- statistics(): A method that return a statistics object of the search tree. It contains the number of nodes expanded, number of restarts, number of failed nodes in the search tree, number of no-goods posted and the maximum depth of the search stack.
- stopped(): A method that returns true if the search engine has been stopped by a stop object, or false if it stopped when it found a solution (or exhausted its search tree).
- nogoods(): A method that returns no-goods constraints. These constraints are created from failures during search and can be used when restarting search to avoid going down the same path in the search tree again.

### 2.4.2 Stop objects in Gecode

In Gecode there exist so called stop objects that can be used to limit the search engine's execution [14]. These stop objects are called by the search engines to perform a validity check if the set limit is reached or not before the search engines

expand a new node in the search tree. If the limit is reached it will stop the search engine and a NULL solution is returned to the user.

There exist three predefined stop object types in Gecode [14], *a) NodeStop*, which limits the number of nodes that can be expanded; *b) FailStop*, which limits the number of failed nodes that can be in the search tree; and *c) TimeStop*, which limits the execution time the search engine can explore the search tree. These three stop object types are all derived from the base class *Stop* which provides them all with the same interface. Therefore stop objects in Gecode have (at least) the *stop()* method which performs the validity check and returns *true* if the limit is reached, or *false* otherwise [14].

## 2.5 Case Studies

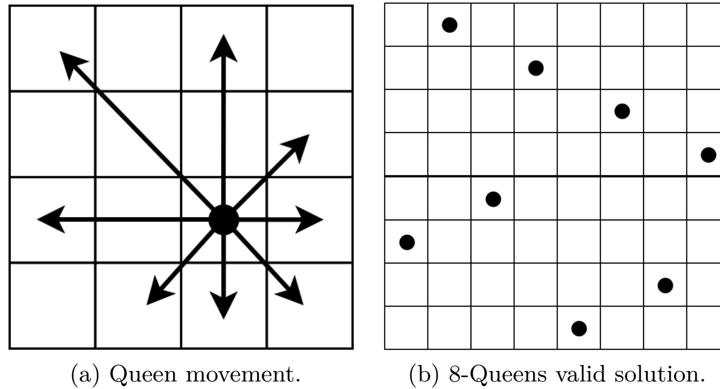
To verify the soundness and performance of the design and implementation of sequential and parallel portfolio search two case studies are performed. Each one cover and test different aspects of the implementations. The first case study for the two portfolio search variants is testing the soundness of the design and implementation. The chosen problem for the first case study is the *N-Queens puzzle* and is described in more detail in subsection 2.5.1. The second case study of the two portfolio search variants is testing the performance compared to that of a single search engine. The chosen problem for the second case study is the *Latin Square Completion problem* and is described in more detail in subsection 2.5.2. This section provides the reader with sufficient theoretical knowledge of the different chosen problems for the case studies. But how the actual case studies are performed is described under their respective sections in chapter 5 where the experimental evaluations are presented and discussed.

### 2.5.1 Case Study: N-Queens

This case study focuses on the soundness of the design and implementation of the two portfolio search variants. It tests that all solutions can be found and that no solutions are ever lost. To be able to efficiently test this, a moderately difficult combinatorial problem which could easily give a very large number of valid solutions on fairly small problem instances is needed. One such problem is the combinatorial problem known as the *N-Queens puzzle* which is a generalization of the *8-Queens puzzle*.

The 8-Queens puzzle is a problem where one must place 8 chess queens on a $8 \times 8$ chessboard without any one of them threatening the other [15]. Therefore a solution cannot have two queens on the same row, column or diagonal. For the original 8-Queens puzzle there exist 92 distinct solutions [16]. There are 12 fundamental solutions that commonly represent these 92 solutions. That means that a solution's reflections and rotations are all counted as one fundamental solution. However, the case study focuses on getting a very large number of solutions and therefore only 92

distinct solutions will not suffice. Figure 2.1 show the valid movements of a queen chesspiece (left) and a valid solution for 8-Queens (right).



(a) Queen movement. (b) 8-Queens valid solution.

Figure 2.1: Figures showing the valid movements of a queen chesspiece and a valid solution for 8 queens on a 8 × 8 chessboard.

The $N$-Queens puzzle is the same as the 8-Queens puzzle but instead of limiting it to just 8 queens on a 8 × 8 chessboard it can take any number $N$ of queens and place them on the corresponding $N \times N$ chessboard. There exist solutions for all natural numbers except for $N$=2 and $N$=3, due to the nature of the queens [16]. Table 2.1 show the number of solutions that exist for a given $N$ ranging from 9 to 20.

Table 2.1: Table showing the total number of solutions for a given $N$ for the $N$-Queens puzzle. The values are taken from the more extensive study in [16].

| *N* | *Nr of solutions* | *N* | *Nr of solutions* |
|---|---|---|---|
| 9 | 352 | 15 | 2’279’184 |
| 10 | 724 | 16 | 14’772’512 |
| 11 | 2’680 | 17 | 95’815’104 |
| 12 | 14’200 | 18 | 666’090’624 |
| 13 | 73’712 | 19 | 4’968’057’848 |
| 14 | 365’596 | 20 | 39’029’188’884 |

As seen in table 2.1 the number of solutions increases quickly with the increase of $N$, therefore making it a suitable test case for validation. Also Gecode already has a complete problem model of the $N$-Queens puzzle as an example that is ready to be used. Thus it guarantees that the implementation of the problem model is sound[2]. It is also worth noting that the $N$-Queens puzzle is one of the benchmarks used to compare backtracking algorithms [16].

[2]There is no need to verify that Gecode’s example programs are correct as they have surely been tested thoroughly before they were released.

### 2.5.2 Case Study: Latin Square

This case study focuses on the performance of the design and implementation of the two portfolio search variants. It tests the performance in terms of execution time and compares it with the execution time of a single search engine. To be able to efficiently test this, a combinatorial problem that is fairly difficult even on small problem instances and gets much harder as the problem size increases is needed. A combinatorial problem that could adjust the level of difficulty only by changing the given data set while keeping the same problem size would be ideal. One combinatorial problem with such qualities is the *quasigroup completion problem* for partially filled *Latin Squares*.

A *Latin Square* is an $N \times N$ array filled with $N$ different symbols. Each symbol is occurring exactly once in each row and exactly once in each column. An example of a Latin Square with $N$=3 is shown in figure 2.2.

| A | B | C |
|---|---|---|
| C | A | B |
| B | C | A |

Figure 2.2: A small Latin Square with $N$=3 where the symbols are A, B and C.

*Quasigroups* are strongly related to Latin Squares, but before their relation can be explained a definition of what a quasigroup actually is is needed. In [7] Gomes defines a quasigroup as follows:

> "A quasigroup is an ordered pair $(Q, \cdot)$, where $Q$ is a set and $(\cdot)$ is a binary operation on $Q$ such that the equations $a \cdot x = b$ and $y \cdot a = b$ are uniquely solvable for every pair of elements $a$, $b$ in $Q$. The *order* $N$ of the quasigroup is the cardinality of the set $Q$."

By creating the multiplication table of a quasigroup of order $N$ as defined by its binary operation results in a $N \times N$ table. Each element of the quasigroup occurs exactly once in each row and exactly once in each column in its multiplication table because of the constraints of the quasigroup. This property ensures that the multiplication table of a quasigroup is a Latin Square.

If a Latin Square is only partially filled it is called an *incomplete* or *partial Latin square*. Its $N \times N$ array is then only partially filled where each symbol occurs exactly once in each row and exactly once in each column.

The *quasigroup completion problem* [17] is to determine if a partially filled Latin Square can be filled in such a way that it becomes a complete Latin Square, which is also the multiplication table of the quasigroup. In [18] Colbourn proved that the quasigroup completion problem is actually NP-complete. This matches the desired qualities of the case study as it is easy to verify solutions (in polynomial time) to NP-complete problems but very difficult to find them.

In [17] Gomes shows that the difficulty of solving a quasigroup completion problem of order $N$ is heavily dependent on the number of preassigned values in its

multiplication table. By only changing the fraction of preassigned values in a partially filled Latin Square of order $N$ Gomes found that the difficulty peaks when the fraction of preassigned values is roughly 42%.

All the characteristics of the quasigroup completion problem make it a suitable test case for the performance of portfolio search. Also Gecode already has a complete problem model of the quasigroup completion problem as an example that is ready to be used. Thus it guarantees that the implementation of the problem model is sound[3]. However, the implementation in Gecode provides two different propagator settings and two different branching settings. It also gives the option to use different factors of tie-breaking as well as providing random seeds. In order to understand the experiments with the quasigroup completion problem all these options must be explained.

The two available propagator settings to the quasigroup completion problem in Gecode are *Binary* and *Distinct*. Binary applies disequality constraints on the variables which is unfortunately very weak [13]. Distinct applies distinct constraints (also known as alldifferent) which enforces that each variable takes pairwise distinct values from the other variables on each row and on each column in the Latin Square [13]. The Distinct propagator is rather strong compared to the Binary propagator.

The two available branching settings are *Size* and *AFC_Size*. Size branches on the variable which has the fewest values left in its set while the AFC_Size setting branches on a variable's AFC_Size [13]. AFC stands for *accumulated failure count* which is the number of times a propagator has failed during search. The AFC of a variable is the sum of AFCs of all propagators that depend on the variable (it is commonly known as the *weighted degree* of a variable). AFC_Size of a variable is its AFC divided by its domain (the size of its set of values).

Tie-breaking rules are needed when two or more variables are equally good during branching [13]. The default behavior for tie-breaking is to pick the first variable that satisfies the selection criteria. But often this is not good enough. Tie-breaking can have different selection criteria, such as most constrained variable or smallest/largest domain. If it is still a tie between two or more variables, tie-breaking can choose one of them either systematically by taking the first/last variable or do it randomly. Until now only exact ties have been considered, but with the use of *tie-breaking limit* functions the user is able to change that. A tie-breaking limit function receives the worst merit value $w$ and the best merit value $b$ and returns a value that determines which variables are considered as ties even if they are not exactly equal. This is useful when the user wants to introduce a bit more randomness in the branching strategy. The user specifies in the model how the returned value shall be calculated, for example as $\frac{w+b}{2.0}$ which returns the average merit value.

*Example*: Take the four variables $a$, $b$, $c$ & $d$ where $a$={1,2}, $b$={1,3,4,5}, $c$={2,3} and $d$={1,2,3,4,5,6} and where the tie-breaking is set to smallest domain with random selection. If no tie-breaking limit function is used the constraint solver

[3]There is no need to verify that Gecode's example programs are correct as they have surely been tested thoroughly before they were released.

will choose randomly between variable $a$ and $c$, as they both have only 2 values in their domains (merit value=2). But if it uses a tie-breaking limit function that returns the average merit value $\frac{6+2}{2.0} = 4.0$ the constraint solver would now choose randomly between variable $a$, $b$ & $c$ as all three of them have better or equal merit values compared to the calculated limit.

Random seeds can be given to the search engines to have them generate different random numbers, thus leading to different branching and selection choices between the search engines [13]. This way variants of a problem can be easily produced for several search engines when running with portfolio search.

# Chapter 3

# Designing Portfolio Search

This chapter guides the reader through the design process of portfolio-based search (hereinafter called PBS) in Gecode. It starts with describing what design decisions were made for the sequential round-robin variant in section 3.1 and then continues with the parallel multi-threaded variant in section 3.2.

PBS is designed as a meta-engine (an engine of engines) which acts as a portfolio. Its task is to monitor and control a given set of search engines. These search engines are assigned a problem model which they are to find solutions for. If one of them finds a solution, the search engine returns it up to PBS which in turn returns it up to the user. The search engines run either in a sequential round-robin fashion (on a single core) where each search engine shares the execution time or in a parallel multi-threaded fashion (on multiple cores) where each search engine has its own thread to execute in.

Both the sequential and parallel PBS variants have parts of their designs in common. The common design is more general and focuses on how the interface of PBS should work from a user perspective while sections 3.1 and 3.2 go more in depth on their internal designs. The common specification for the two PBS variants is as follows:

- PBS should have (at least) the same methods as *EngineBase*, but modified to account for the number of search engines. Thus having the same interface as normal search engines.
- PBS should take a set of search engines as an argument and use them to find solutions for the given problem.
- Search engines provided to PBS should already be created and provided with the problem model.
- PBS should provide each search engine with an equal amount of execution time.
- The user should not be able to modify or control the internal workings of PBS beyond what the methods of PBS allow and should be considered as a single search engine.
- The user should be able to provide PBS with a global *stop object*.

- The user should not try to access and/or modify search engines provided to PBS other than allowed by PBSs methods. By doing so it will result in undefined behavior.

## 3.1 Design of Sequential PBS

Sequential PBS is to run the provided search engines interleaved on a single core in a round-robin fashion. Figure 3.1 illustrates the three layers of abstraction. To the left is the topmost abstraction layer where the user has its program. It is here that the problem model, search engines and sequential PBS are all created and initialized. When the user calls the `PBS:next()` method (covered in subsection 3.1.1) the program enters the middle abstraction layer, which is that of sequential PBS. Internally sequential PBS handles the round-robin structure and sees to that its provided search engines run interleaved. When `PBS:next()` call one of the search engine's `next()` method the program enters the lowest abstraction layer, which is that of a search engine. Internally the search engine is searching for a solution and returns the result back up to PBS. The result is either a solution or a `NULL` value. When `PBS:next()` has received a solution it returns it back up to the user.

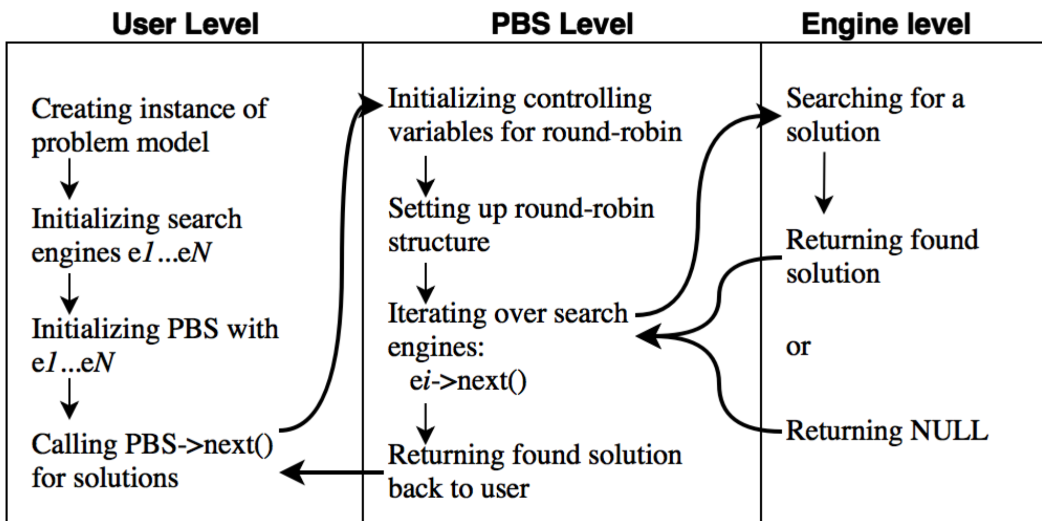



Figure 3.1: Diagram illustrating the three layers of abstraction of a program using sequential PBS as well as the program flow.

Only the design of the PBS:next() method is described because it is the only method that receives significant changes in sequential PBS. The rest of the methods are very similar to their single search engine counterparts and are therefore not of interest.

### 3.1.1 PBS:next()

In the sequential round-robin variant, each search engine gets its own equal slice of every run cycle. A slice is a fixed amount of nodes that each search engine is allowed to explore before stopping to let the next search engine in line run. When all search engines have run their slice the cycle starts over with fresh slices and the first search engine continues to run from where it first stopped. This continues until either a solution is found or every search engine has exhausted their whole search tree. A search engine can be stopped by four reasons:

1. It has explored the maximum number of nodes allowed in its slice.
2. It has found a solution and immediately stops and returns the solution to `PBS:next()` (even if it has unused nodes left in its slice).
3. It has exhausted its search tree and immediately stops and returns `NULL` to `PBS:next()` (even if it has unused nodes left in its slice).
4. When PBS itself is stopped by its own stop object. Then the currently running search engine is stopped mid-slice by `PBS:next()` and the other search engines are prevented to start running.

When `PBS:next()` receives a solution from the currently running search engine it will prevent the other search engines to resume running and immediately return the solution to the user. When `PBS:next()` receives `NULL` from the currently running search engine it will not return `NULL` to the user. Instead `PBS:next()` lets the next search engine in line resume running. Only when `PBS:next()` receives `NULL` from all search engines (which implies that all search trees are exhausted) will it stop and return `NULL` to the user[1]. This design decision was made in order to be able to check that all search engines can find all solutions, i.e. that no solutions are lost. The intention is that the user should be able to enable/disable this feature as it could be in the user's interest to run several different problems with portfolio search at the same time. Due to time constraints it has not been implemented in the prototypes but it should be implemented in a future version.

The `PBS:next()` method can be called again after returning a solution or after the stop object for PBS has been updated. When it is called again the search engine who last returned a solution or was stopped mid-slice will resume running. Because of this each search engine will use up its full slice each cycle and make the round-robin fair. The only exception is when a search engine has exhausted its search tree and can no longer search, it will then always return `NULL` when prompted for a solution simply because there are no more solutions to be found in its search tree.

## 3.2 Design of parallel PBS

The design of parallel PBS is more complicated. Parallel PBS is to run the provided search engines in parallel on multiple cores, each one in its own thread. Once the

[1] This design decision will have a negative impact only when running on a problem instance that do not have any solutions at all.

search engines are running in their threads PBS has no direct control over them contrary to the sequential variant. Because of this many new problems have to be solved. Some of the more crucial problems are:

- To get every search engine to run in parallel.
- To stop the running search engines from PBS.
- To find a good structure for the communication that ensures that no messages or solutions are lost.

Figure 3.2 illustrates the four layers of abstraction. To the left is the topmost abstraction layer where the user has its program. It is here that the problem model, search engines and parallel PBS are all created and initialized. When the user calls the `PBS:next()` method (covered in subsection 3.2.1) the program enters the upper middle abstraction layer, which is that of parallel PBS. Internally parallel PBS handles the parallel structure, creates threads and sees to that the provided search engines run in them. When the threads are started `PBS:next()` does not have any direct control over the threads (that is why their arrows are dashed) and therefore waits for a signal from one of the threads. The threads run in the lower middle abstraction layer where they call the search engine's `next()` method and communicate with `PBS:next()`. When a thread calls its search engine's `next()` method the program branch enters the lowest abstraction layer, which is that of a search engine. Internally the search engine is searching for a solution and returns the result back up to the thread. The result is either a solution or a `NULL` value and the thread signals `PBS:next()` accordingly. When `PBS:next()` has received a solution from the shared queue (covered in 3.2.1) it signals all other threads to stop and then returns the found solution back up to the user.

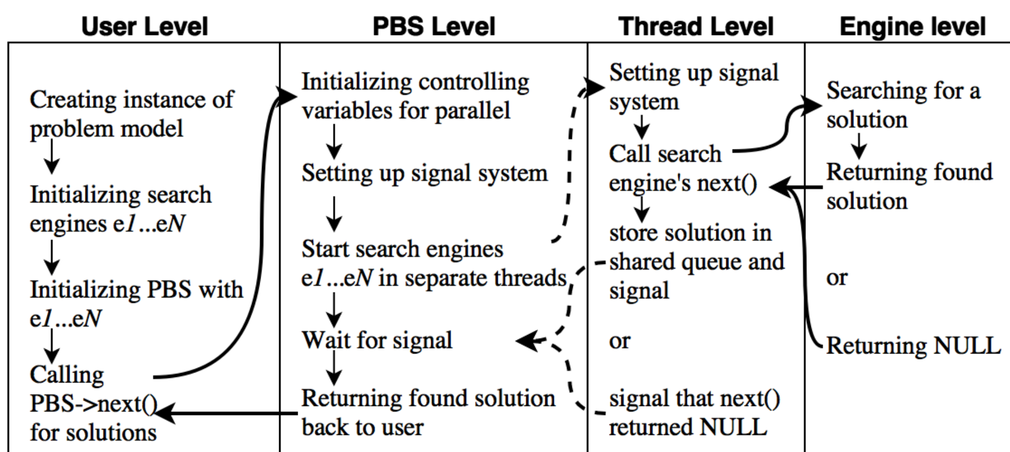



Figure 3.2: Diagram illustrating the four layers of abstraction of a program using parallel PBS as well as the program flow and communication.

Only the design of the PBS:next() method is described because it is the only method that receives significant changes in parallel PBS. The rest of the methods

are very similar to their single search engine counterparts and are therefore not of interest. The signal protocol is also described briefly in 3.2.2.

### 3.2.1 Parallel PBS:next()

PBS has a `next()` function which when called returns the next found solution (or `NULL` if no more solutions exist). In the parallel multi-threaded variant, each search engine get its own thread to run in. All search engines also share access to a shared protected queue where they put found solutions where `PBS:next()` can retrieve them later. All search engines run like this in parallel until one of them finds a solution or every search engine has exhausted their search tree. A search engine can be stopped by four reasons:

1. It has found a solution and immediately puts the solution in the shared protected queue where `PBS:next()` can retrieve it later. It then signals `PBS:next()` and terminates its thread.
2. It has exhausted its search tree and immediately signals `PBS:next()` without putting anything in the shared protected queue. It then terminates its thread.
3. It receives a stop signal from `PBS:next()` which means that another engine has already found a solution. It then signals back to verify and terminates its thread.
4. When PBS itself is stopped by its own stop object. Then naturally all running search engines are stopped mid-search by `PBS:next()` and they all signal `PBS:next()` and then terminate their threads.

When a search engine finds a solution it is stored in the shared protected queue where `PBS:next()` can retrieve it later. It then signals `PBS:next()` to alert it to the newly found solution and terminates its thread. `PBS:next()` then sends out stop signals to every search engine and then retrieves the solution from the shared protected queue and returns it to the user.

The `PBS:next()` method can be called again after returning a solution or after the stop object has been updated. When this happens `PBS:next()` first checks in the shared protected queue if there are any solutions there[2]. If there is, it simply returns the first solution in the queue. This continues for each successive call to `PBS:next()` until there are no more solutions in the queue, it then restarts all search engines in threads and waits for them to find a new solution and signal `PBS:next()`.

The exception is when a search engine has exhausted its search tree. It then signals `PBS:next()` without putting anything in the shared protected queue and then terminates its thread. `PBS:next()` then checks the queue and sees that it is empty and instead of returning `NULL` to the user it remembers that one search engine is finished and continues to wait for a solution. It is first when all search engines have exhausted their search trees, signaled `PBS:next()` and terminated their

[2] As all search engines run in parallel in separate threads it can happen that two or more threads receive found solutions from their search engines at the same time and put the solutions in the shared protected queue before receiving a stop signal from `PBS:next()`.

threads that `PBS:next()` knows that all search engines are finished. As the shared protected queue is still empty `PBS:next()` knows that there are no more solutions to be found and only then returns `NULL` to the user. This design decision was made in order to be able to check that all search engines can find all solutions, i.e. that no solutions are lost. The intention is that the user should be able to enable/disable this feature as it could be in the user's interest to run several different problems with portfolio search at the same time. Due to time constraints it has not been implemented in the prototypes but it should be implemented in a future version.

### 3.2.2 Parallel Communication Protocol

Because the PBS:next() method and the search engines execute in separate threads a communication protocol is designed. A master-slave structure is used. This means that each slave thread with a search engine communicates only with the master thread, which is `PBS:next()`, while the master thread communicates to all slave threads with search engines. A master-slave structure is easy to design and implement but could potentially become a bottleneck. But as it is highly unlikely that the normal user will have enough cores to run with so many search engines in separate threads that it actually becomes a problem it is completely outweighed by the simplicity of the structure.

Figure 3.3 illustrates the communication protocol with two search engines. It starts with the initialization of PBS and continues until `PBS:next()` returns the solution one of the search engines found. It then continues illustrating what happens when a second call to `PBS:next()` is made. As is shown in the figure `PBS:next()` creates the threads and then waits for a signal from one of them. When a signal is received `PBS:next()` signals the other thread to stop and wait for a verification signal. When `PBS:next()` gets the verification it can safely return the found solution to the user as all other threads have terminated, thus not leaving residue threads behind. When `PBS:next()` is called a second time it creates new threads again and the search engines pick up where they left (as it was just the threads that were terminated while the search engines kept their states in their own objects).

Actually Gecode keeps a pool of threads ready so the threads are not created and destroyed all the time but instead fetched from and released back to Gecode's own thread pool [19]. This speeds up the thread handling process significantly. Gecode provides abstractions for parallel programs to make it easier for the user. Gecode already has implemented classes for mutexes, locks, signaling events, thread handling and runnable interfaces (to name those that concern this thesis).

Mutexes are used for protecting critical areas in the code, for example access to shared data structures. Only one thread can have access to the mutex at any time, if other threads try to take it they are blocked until the thread holding the mutex releases it. Locks are a sort of add-on to mutexes which only makes sure that a mutex is released after the end of the current scope in the code

Signaling events are objects that threads can wait and signal. When the event receives a signal it increments a counter inside it by one. When a thread waits on

the event the same counter is decreased by one, but the counter can never go below zero. So if the counter is at zero and a thread waits on the event it is blocked until another thread signals the event.

For an object to be able to run in its own thread it must extend the `Runnable` class and have a `run()` method. When an object is given to a thread its `run` method is executed by the thread. When the `run` method is finished the thread destroys the object and is then released back into the thread pool.

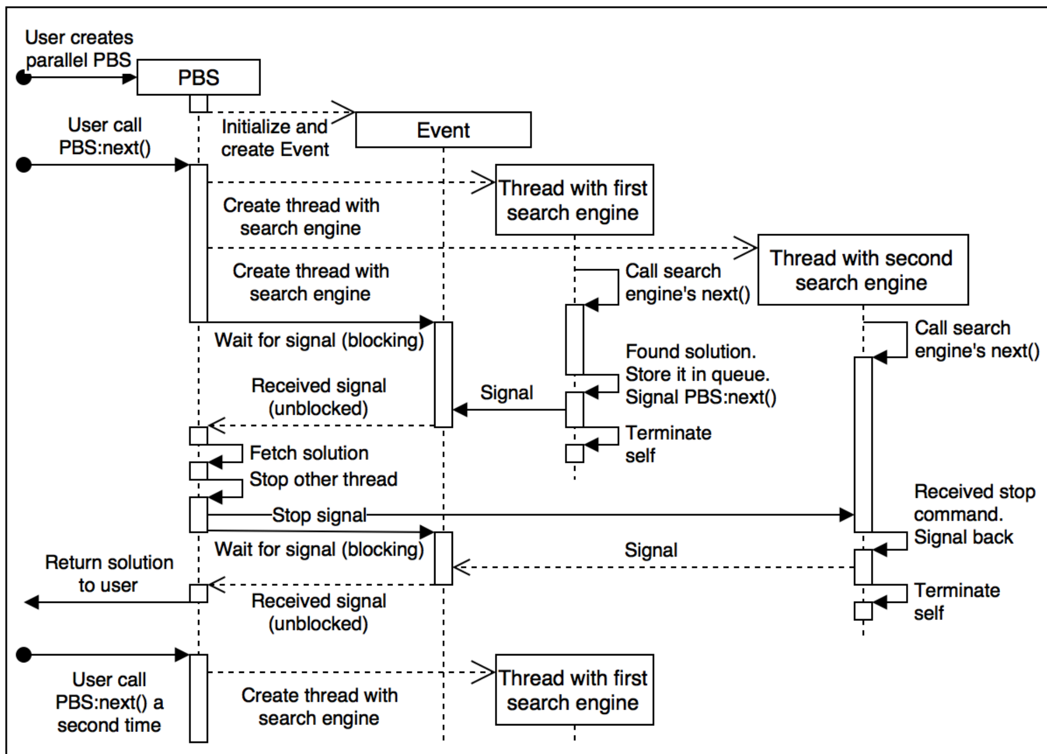



Figure 3.3: Diagram illustrating the communication protocol in action using parallel PBS with two search engines. It starts with the initialization of parallel PBS and continues until a solution is returned to the user. Then it continues illustrating what happens when a second call to PBS:next() is made.

# Chapter 4

# Implementing Portfolio Search

This chapter explains how the implementation of PBS was done in Gecode. First section 4.1 explains how the sequential variant of PBS was implemented and how it started with the most basic case. It then shows step by step in each subsection how the implementation evolved to its final state. Section 4.2 continues with explaining the implementation of the parallel variant of PBS in the same manner as the sequential one. All code segments illustrated in this chapter have been reduced to C++-like pseudo code in order to highlight the structure instead of the exact C++ syntax. As the chapter describes the implementation incrementally, code that has not changed between iterations are omitted with "..." to highlight only the changes which were made. The complete source code for both PBS variants can be found in appendix B.

## 4.1 Implementation of Sequential PBS

As the sequential PBS was designed first it was also implemented first. The subsections that follow explain in more detail the different steps and key-structures of the implementation. The first step was to implement the most basic case with only two search engines that worked in a round-robin fashion and could return the first found solution. In order to succeed with it a stop object of type NodeStop was created to control the round-robin structure. The second implementation step was to get the two search engines to find all solutions in their respective search trees. This was done to be able to verify that no solutions are ever lost. The third implementation step was to generalize it so that PBS can take any number of search engines as input and run round-robin with all of them. The fourth implementation step was to give PBS the option to have a global stop object. For this a stop object wrapper was created that encapsulates both the round-robin controlling *NodeStop* object as well as the optionally provided global stop object. The fifth and last implementation step was to look over the other methods that a search engine should have such as `statistics()`, `nogoods()` and `stopped()` and see that they worked properly with all earlier implementation steps.

### 4.1.1 PBS:next() with 2 Search Engines for One Solution

The round-robin functionality is implemented using a stop object of type *NodeStop*. Using a NodeStop object instead of FailStop or TimeStop is motivated in the two following sentences. NodeStop would be fairer than FailStop because FailStop is unpredictable and could lead to some search engines choosing the “right” branch and then get long runs while another search engine choosing a “bad” branch would hit failed nodes almost instantly and then have to stop. NodeStop would give finer resolution than TimeStop because with TimeStop the search engines would depend on how long they execute and on small problems they find solutions so quickly that it would be difficult to set so short execution times due to the imprecision of the timer while NodeStop would be able to control them down to one node each for every cycle.

The controlling NodeStop object is given to both search engines and is therefore able to control them. More specifically NodeStop objects check how many nodes have already been expanded in a search tree and compares it with the given limit before the search engine can expand the next node. If that number exceeds the limit NodeStop stops the search engine from expanding further nodes and makes the search engine return `NULL`.

**Note:** Any other stop object given to the search engines by the user prior to creating a PBS object are overwritten with the controlling NodeStop object. If the user were to change the stop object of one of the search engines from the outside later on the behavior would be undefined.

As each search engine is initialized with its own variant of the problem they all get their own individual search trees and can therefore not interfere with each other. If they were to share the same copy of a search tree the behavior would be undefined. Figure 4.1 shows the first implementation of the `PBS:next()` method where it returns the first found solution (not taking into account the scenario that no solutions exist). First search engine 1 (`e1`) runs by calling its `next()` method (where it searches for a solution in its search tree) and return the result. If a solution is found `e1` immediately returns it to `PBS:next()` which in turn returns it up to the user and stops. If `e1` is stopped by the NodeStop stop object (`so`) the returned value is `NULL`. In that case the second search engine (`e2`) runs. If it also is stopped before it finds a solution the execution cycle is finished. The NodeStop object then increases its limit by 10 nodes and the next execution cycle begins where `e1` continues to run. The NodeStop limit was chosen to be 10 nodes to get a fine granularity on the slices in each execution cycle.

### 4.1.2 PBS:next() with 2 Search Engines for All Solutions

In order to find all existing solutions for a given problem the user needs to repeatedly call `PBS:next()` and receive one solution at a time until he/she receives `NULL`. But for `PBS:next()` to be able to return `NULL` it is crucial for it to recognize when a search engine has exhausted its search tree. Otherwise it would just continue on

```
next() {
  returned = NULL;
  while(true){
    // Engine 1
    returned = e1->next();
    if(returned != NULL){
      return returned;
    }
    // Engine 2
    returned = e2->next();
    if(returned != NULL){
      return returned;
    }
    // Increasing NodeStop object limit by 10
    so->limit(10);
  }
  return returned;
}
```

Figure 4.1: Code segment illustrating the first implementation of the round-robin structure in sequential `PBS:next()` where the first found solution is returned (ignoring the scenario where no solutions exist). The NodeStop stop object (`so`) is created in PBSs constructor and given to the search engines.

forever asking `e1` and `e2` for solutions where no more solutions exist (as a search engine returns `NULL` if the search tree is exhausted).

Figure 4.2 shows the second implementation of the `PBS:next()` method where it can find all solutions (also taking into account the scenario that no solutions exist). Building onto the first implementation of `PBS:next()` by adding an additional clause right after `e1` and `e2` have run their execution slices solves this problem. The clause checks if *both* `e1` and `e2` have *not* been stopped by any stop object. If `e1` or `e2` had found a solution this clause would not have been reached as `PBS:next()` would have returned the solution and stopped. If neither `e1` nor `e2` found a solution (thus returning `NULL`) and did not get stopped by a stop object there are no more solutions to be found. Thus `PBS:next()` can be guaranteed that no more solutions exists and return `NULL` back to the user[1].

**Note:** When a second call is made to `PBS:next()` the search engine who found the previous returned solution continues to run. This is automatically taken care of due to the structure of the code. If `e1` found the previous solution it still has some of its execution slice left (unless it found the solution in its last node). So when `PBS:next()` is called again `e1` continues to run. But if it was `e2` who found the solution `e1` tries to run but is immediately blocked by the NodeStop object and

[1]Reminder: This design decision was made to later be able to verify that all search engines can find all solutions and that no solutions are ever lost in the communication.

```
next() {
  returned = NULL;
  while(true){
    ...
    // Are both finished?
    if(e1->stopped() == false and
       e2->stopped() == false){
      break;
    }
    ...
  }
  return returned;
}
```

Figure 4.2: Code segment illustrating the second implementation of the round-robin structure in sequential `PBS:next()` where all solutions can be found (also solving the scenario where no solutions exist).

then `e2` continues executing in its remaining execution slice.

### 4.1.3 PBS:next() with N Search Engines

For PBS to be able to run with any number of search engines the structure needs to be changed in several places. First the constructor that previously took two pointers to search engines now needs to take a vector of pointers to search engines. Second the `PBS:next()` method needs to be generalized for taking into account the unpredictable amount of search engines.

Figure 4.3 shows the third implementation of the `PBS:next()` method where it can find all solutions and also run any number of search engines in a round-robin fashion. A for-loop iterating over all search engines (`engines[i]`) lets each of them run its execution slice in turn. If a search engine finds a solution it immediately returns it to `PBS:next()` which in turn returns it to the user and stops as in the previous implementations. Furthermore if `PBS:next()` is called a second time the last active search engine continues to run. To check if all search engines have exhausted their search trees an additional for-loop is required to count how many of them have done so. If the number of exhausted search trees are equal to the number of search engines (`NR_ENGINES`) `PBS:next()` returns `NULL` back to the user.

### 4.1.4 Global Stop Object for PBS

Apart from the NodeStop stop object, the user should also be able to provide PBS with its own global stop object. This complicates matters as there are now two different stop objects that must be obeyed but a search engine can only have one. Furthermore the global stop object must have precedence over the round-

```
next() {
  returned = NULL;
  while(true){
    // Run each search engine in turn
    for( i = 0; i < NR_ENGINES; i++ ){
      returned = engines[i]->next();
      if(returned != NULL){
        return returned;
      }
    }
    // Check if all search engines have exhausted their
    // search trees
    for( i = 0, count = 0; i < NR_ENGINES; i++ ){
      if(engines[i]->stopped() == false){
        count++;
      }
      if(count == NR_ENGINES){
        return returned;
      }
    }
    ...
  }
  return returned;
}
```

Figure 4.3: Code segment illustrating the third implementation of the round-robin structure in sequential `PBS:next()` where any number of search engines can be run with PBS.

robin controlling NodeStop object as it controls the entirety of PBS. Therefore the creation of a stop object wrapper class is needed (hereinafter called *StopWrapper*). The `StopWrapper` class inherits from the base class *Stop* and is therefore treated as a regular stop object. It takes two stop objects as input parameters and then when it is called it first checks the global PBS stop object if it should stop (if PBS was provided with one). If not, it then checks the round-robin NodeStop object if it should stop. If not, it returns false. If any of them returns true it also returns true. This way PBS can use several stop objects on its search engines by putting them in the `wrapper` stop object, one for controlling round-robin and one optional for the entire PBS engine.

The `StopWrapper` class is a private internal class inside PBS and is therefore hidden from the user and protected from outside use. As the global stop object needs to make decisions on the statistics of the entire PBS the statistics of all search engines need to be summed up together and updated for each new node expanded. This requires some special methods that a normal stop object does not have in order to handle the complexity of two stop objects. A short explanation of

the methods is given in the following list.

- **setOldStats():** Takes the current statistics from the active search engine and copies them. This is to have a reference point in order to take the difference in statistics after each new expanded node.
- **update():** Takes the current statistics from the active search engine and adds the difference of the new and old statistics to the total statistics of PBS. It then calls the `setOldStats()` method to set the current statistics as reference point for the next expanded node.
- **resetstatus():** It resets the boolean status to false for both the NodeStop object and the global stop object. This is to ensure it starts with the right configuration when `PBS:next()` is called.
- **stop():** This is the only method a stop object must have. It is the method that is called right before a search engine expands a node to check if the search engine should be stopped or not. This method has been modified to account for the use of two stop objects, their precedence as well as updating the total statistics of PBS.
- **enginestop():** This method returns a pointer to the NodeStop object so `PBS:next()` can update the limit for the round-robin structure.
- **enginestopped():** Checks whether the NodeStop object stopped a search engine.
- **pbsstopped():** Checks whether the global stop object stopped a search engine. If so the entire `PBS:next()` method should stop and return `NULL` to the user.

Figure 4.4 shows the changes in `PBS:next()` to account for the `StopWrapper` object in the fourth implementation. In the beginning of `PBS:next()` the `StopWrapper` (`so` in the code) resets the status of the two stop objects inside it. Inside the for-loop right before the active search engine calls its `next()` method the `StopWrapper` must call its `oldstats()` method on it. Then directly after the `StopWrapper` must also update the statistics by calling the `update()` method. An additional clause is also present after these three lines of code. That clause checks if the global stop object stopped the search engine. If that is the case `PBS:next()` returns `NULL` back to the user. The last change is the line where the NodeStop object limit is updated. The result is the same as before even though the call is a bit longer.

### 4.1.5 Other Methods in PBS

The other methods that PBS has are `statistics()`, `nogoods()` and `stopped()`. The `statistics()` method just adds each of the search engines own statistics into one collected statistics object and then returns it up to the user. The `nogoods()` method was deemed unnecessary for this prototype. Therefore it has not been implemented so it just returns `NULL` whenever the user calls it. The `stopped()` method checks whether PBS has been stopped by its global stop object and returns the answer. If no global stop object has been given to PBS false is returned.

```
next() {
  so->resetstatus();
  returned = NULL;
  while(true){
    for( i = 0; i < NR_ENGINES; i++ ){
      so->setOldStats(engines[i]->statistics());
      returned = engines[i]->next();
      so->update(engines[i]->statistics());
      if(so->pbsstopped() == true){
        return returned;
      }
    }
    ...
    // Increasing NodeStop object limit by 10
    so->enginestop()->limit(10);
  }
  return returned;
}
```

Figure 4.4: Code segment illustrating the added changes in the fourth implementation of `PBS:next()` for using a stop object wrapper class to be able to use multiple stop objects in the search engines.

## 4.2 Implementation of Parallel PBS

As the parallel PBS was designed last it was also implemented last to take advantage of and build upon the common parts from the sequential variant. The subsections that follow explain in more detail the different steps and key-structures of the implementation. The first step was to make search engines able to run in their own thread. Therefore a `run_wrapper` class was created to encapsulate a search engine to give it a `run()` method so that it could run in its own thread. The second implementation step was to implement the most basic case with only two search engines that worked in parallel and could return the first found solution. The third implementation step was to make PBS able to stop the second search engine after the first one had found and returned a solution. In order for it to work a modified stop object class was created that contains a pointer to a control variable which the search engines check before exploring a new node. The fourth implementation step was to address the possibility of a race-condition when two search engines find a solution at the same time and both of them try to return it to PBS. In order to solve this, a shared solution-queue was created that is accessed through a mutex, therefore guaranteeing safety. The fifth implementation step was to get the two search engines to find all solutions in their respective search trees. This was done to be able to verify that no solutions are ever lost. The sixth implementation step was to generalize it so that PBS can take any number of search engines as input and run them all in parallel. The seventh implementation step was to give PBS the

option to have a global stop object. For this to work the modified stop object class had to be changed to account for the second optional stop object. The eighth and last implementation step was to look over the other methods that a search engine should have such as `statistics()`, `nogoods()` and `stopped()` and see that they worked properly with all earlier implementation steps. The last subsection explains briefly how this implementation avoids getting stuck in deadlocks.

### 4.2.1 Run Wrapper

Gecode provides a pool of available threads and takes care of all the management around it so the user does not have to bother with creating and destroying threads [19]. The only thing needed for an object to run in its own thread is that the object must have a `run()` method that the thread can execute. But since search engines normally do not have a `run()` method a `run_wrapper` class was created to solve that.

Figure 4.5 shows the first implementation of the `run_wrapper` class in its most basic form. The `run_wrapper` class is a private internal class inside PBS and is therefore hidden from the user and protected from outside use. The constructor takes a pointer to a search engine, a pointer to an event and a pointer to a shared solution variable and stores them in its own local variables. When a `run_wrapper` object is created and given to a thread the thread executes the `run()` method. In the `run()` method the search engine's `next()` method is called, which starts the search. When `next()` returns a solution (or `NULL`) the `run_wrapper` then puts the result in the `shared_solution` variable, signals the main thread through the event that a solution (or `NULL`) has been found and then terminates. This first implementation ignores the possible race-condition created by having an unprotected shared variable to keep it as simple as possible in the beginning.

```
class run_wrapper : public Support::Runnable {
  public:
    e;
    event;
    shared_solution;

    run_wrapper(e, event, shared_solution) {}

    run() {
      result = e->next();
      shared_solution = result;
      event->signal(); // Signal main thread
    }
};
```

Figure 4.5: Code segment illustrating the first version of the `run_wrapper` class.

### 4.2.2 PBS:next() with 2 Search Engines for One Solution

The implementation started out similar to the sequential variant. Two search engines were considered for the first prototype to keep the problem size small and manageable, which was even more important considering the devious nature of multi-threading. As each search engine is initialized with its own variant of the problem they all get their own individual search trees and can therefore not interfere with each other. If they were to share the same search tree the behavior would be undefined. Figure 4.6 shows the first implementation of the `PBS:next()` method where it returns the first found solution. Because of the parallel structure of the program, each search engine had to run in its own thread. With the help of the `run_wrapper` class each search engine got its own `run()` method and could therefore run in separate threads. When the threads have been started `PBS:next()` waits for one of the threads to signal the event which indicates that a solution has been found. When it is signaled the found solution is retrieved from the `shared_solution` variable and then returned to the user.

**Note:** As the solutions are complex objects it is actually the pointer to the solution that is stored in the `shared_solution` variable and not the solution object itself.

```
next() {
  returned = NULL;
  // Create run_wrappers for the engines and start
  // them in separate threads.
  Gecode::Support::Thread::run(
    new run_wrapper(e1, event, shared_solution) );
  Gecode::Support::Thread::run(
    new run_wrapper(e2, event, shared_solution) );

  // Wait for search engines to find a solution
  event->wait();
  returned = shared_solution;
  return returned;
}
```

Figure 4.6: Code segment illustrating the first implementation of the parallel structure in parallel `PBS:next()` where the first found solution is returned (ignoring the possible race-condition created by the unprotected `shared_solution` variable).

### 4.2.3 Control Stop Object for Search Engines

That `PBS:next()` could return the first found solution (or `NULL` if no solutions exist) was a good start. But it was really important that `PBS:next()` could stop the execution of the search engines. Because when one search engine has found a solution and notified `PBS:next()` about it through signaling the event, `PBS:next()` should stop

all other search engines before returning the solution back to the user. The combination of a modified stop object class (hereinafter called *moddedStopObject*), a global shared command variable (hereinafter called *PBS_Command*) as well as a global shared command mutex (hereinafter called *command_mutex*) turned out to be the best solution for this problem. The `moddedStopObject` class inherits from the base class *Stop* and is therefore treated as a regular stop object. The `command_mutex` is needed to protect `PBS_Command` during read/write operations in order to avoid race-conditions. The structure of this communication is that `PBS:next()` only writes commands to the `PBS_Command` and the search engines only read from it. Aside from the creation of the `moddedStopObject` class some minor changes were also needed in the `PBS:next()` method.

Figure 4.7 shows the first implementation of the `moddedStopObject` class where `PBS_Command` and `command_mutex` give the search engines the ability to read the commands. The `moddedStopObject` class is a private internal class inside PBS and is therefore hidden from the user and protected from outside use. Every time the `moddedStopObject's stop()` method is called by the search engine it first acquires the `command_mutex`, checks the current command, releases the `command_mutex` and then return true (if it should stop) or false (if it should continue) back to the search engine.

As the stop mechanism was implemented as a stop object every search engine checks if it should stop or continue before exploring each new node. This results in the `PBS_Command` being seen almost immediately when a new command is given. This structure also leaves the door open for future designs and implementations with additional commands.

In figure 4.8 only the changes done in `PBS:next()` are shown. By acquiring the `command_mutex`, changing the command in `PBS_Command` and then releasing the `command_mutex`, `PBS:next()` gets the ability to stop the search engines. After changing the command to 1, `PBS:next()` must wait for the second search engine to stop before it can return the solution to the user. This is because if it does not wait for the other search engine to finish and signal the event, that signal will be left on the event. The next time `PBS:next()` is called it will then get the residue signal from the last call which will result in undefined behavior.

### 4.2.4 Parallel Queue for Solutions

When `PBS:next()` could return the first found solution (or `NULL` if no solutions exist) and also could stop the other search engine the next step was to find all existing solutions. But in order to do that the existing implementation with an unprotected `shared_solution` variable is not enough as solutions could potentially be partly overwritten (resulting in garbage and undefined behavior) or completely overwritten (resulting in lost solutions)[2]. Adding a mutex for protecting the `shared_solution` variable only prevents the solutions to be partly overwritten.

[2] Actually the pointers to the solutions. Not the solution objects themselves.

```
class ModdedStopObject : public Search::Stop {
  protected:
    command_mutex;
    PBS_Command;

  public:
    ModdedStopObject(command_mutex, PBS_Command) {}

    bool stop(...) {
      command_mutex->acquire();
      /* 1 == stop / 0 == go on */
      if(*PBS_Command == 1) {
        command_mutex->release();
        return true;
      }
      command_mutex->release();
      return false;
    }

    ~ModdedStopObject(void) {}
};
```

Figure 4.7: Code segment illustrating the first implementation of the modified stop object class for giving `PBS:next()` the ability to stop the search engines through a shared `PBS_Command` variable protected by a shared `command_mutex`.

```
next() {
  ...
  // Change PBS_command to stop
  command_mutex->acquire();
  /* 1 == stop / 0 == go on */
  PBS_Command = 1;
  command_mutex->release();
  // Wait for other search engine to stop
  event->wait();
  return returned;
}
```

Figure 4.8: Code segment illustrating the second implementation of the parallel structure in parallel `PBS:next()` where the first found solution is returned and the other search engine is stopped (still ignoring the possible race-condition created by the unprotected shared variable).

But a solution could still be completely overwritten if two search engines write their solutions to `shared_solution` before the main thread get hold of the mutex and retrieves the solution. Therefore to counter the possibility of lost solutions a `queue` class was created. The `queue` class is called `PBS_queue` and it replaces the global `shared_solution` variable. But it still needs to be protected by a mutex in order to guard against race-conditions.

`PBS_queue` is a simple FIFO queue implementation with only two methods, `push()` and `pop()`, and is tailored for handling solutions. The queue has two node pointers to keep track of the first and last nodes in the queue. Each node in the queue contains a pointer to a found solution and a pointer to the next node. Should the queue be empty and a call to `pop()` is made the returned value is `NULL`. The `PBS_queue` class is a private internal class inside PBS and is therefore hidden from the user and protected from outside use. The complete source code for `PBS_queue` can be found in appendix B along with the rest of parallel PBS.

### 4.2.5 PBS:next() with 2 Search Engines for All Solutions

In order to find all existing solutions for a given problem the user needs to repeatedly call `PBS:next()` and receive one solution at a time until he/she receives `NULL`. For parallel `PBS:next()` to be able to provide this functionality there are several aspects that must be taken into account. First it needs to be able to store found solutions that have not yet been retrieved. This is accomplished by using the `PBS_queue` implemented in the previous subsection instead of the `shared_solution` variable. The calls to `PBS_queue` must also be protected with a mutex (hereinafter called `solution_mutex`) to ensure safety. `PBS:next()` also needs to keep track on how many of the search engines have exhausted their search trees. Lastly `PBS:next()` must be able to handle the scenario that a search engine returns `NULL` (indicating that its search tree is exhausted) and then keep waiting for the other search engine to return either a solution or `NULL` before returning it to the user. This is because according to the design decisions it should not return `NULL` back to the user before every search engine has exhausted its search tree[3]. To account for all these new requirements, changes were made in the `run_wrapper` class, in the `PBS:next()` method as well as in the global scope.

The `run_wrapper` class only got a minor change as illustrated by figure 4.9. The only change is the addition of a clause checking if the found solution is `NULL` or not. If it is not, then the solution is stored in the queue safely by acquiring the `solution_mutex` right before and releasing it just after.

There were some larger changes in the `PBS:next()` method this time. Figure 4.10 shows the third implementation of the `PBS:next()` method where it can find all solutions with two search engines. The first addition is in the beginning where it acquires the `solution_mutex`, pops a solution from the queue and then releases the mutex. If there is an already found solution in the queue, it returns it back

[3]Reminder: This design decision was made to later be able to verify that all search engines can find all solutions and that no solutions are ever lost in the communication.

```
class run_wrapper : public Support::Runnable {
  public:
    ...
    run() {
      result = e.next();
      if(result != NULL) {
        solution_mutex->acquire();
        queue->push(result);
        solution_mutex->release();
      }
      event->signal(); // Signal main thread
    }
};
```

Figure 4.9: Code segment illustrating the second version of the `run_wrapper` class with mutex-protected access to a solution queue. The `shared_solution` variable was replaced with a pointer to the `PBS_queue` and a `solution_mutex` was added to protect the access.

to the user and stops. If the queue was empty, `NULL` is returned and `PBS:next()` continues with creating run wrappers for the search engines and starting them in their threads. After that the second addition takes place. First `PBS:next()` waits for a signal on the event. When a signal is received it checks if a solution is in the queue. If yes, it retrieves it, tells the other search engine to stop and then waits for it to signal. Then it returns the solution to the user. If not, it now knows that one of the search engines has exhausted its search tree and `PBS:next()` must therefore wait for a signal from the other search engine. When the other search engine signals the event `PBS:next()` once again tries to receive a solution from the queue and then return it to the user. This time it does not matter if it is a solution or `NULL`. This way `PBS:next()` is guaranteed to only return `NULL` if both search engines have exhausted their respective search trees.

**Note:** When a second call is made to `PBS:next()` all search engines continue where they were when they were stopped by `PBS:next()` (or last found a solution).

### 4.2.6 PBS:next() with N Search Engines

For PBS to be able to run with any number of search engines the structure had to be changed in several places. First the constructor for PBS that previously took two pointers that point to search engines now had to take a vector of pointers that point to search engines. Second two global counters, `finished_threads` and `stopped_threads`, are necessary for `PBS:next()` to be able to keep track on how many search engines have finished and how many have been stopped. This calls for some minor changes in the `run_wrapper` class. Finally the `PBS:next()` method had to be generalized for taking into account the unpredictable amount of search

```
next() {
  ...
  // Check if there exist already found solutions
  // in queue. If yes, return said solution.
  solution_mutex->acquire();
  returned = queue->pop();
  solution_mutex->release();
  if(returned != NULL) {
    return returned;
  }
  ...
  event->wait();
  solution_mutex->acquire();
  returned = queue->pop();
  solution_mutex->release();
  // If solution found, just stop other search engine
  if(returned != NULL) {
    // Change PBS_command to stop
    command_mutex->acquire();
    /* 1 == stop | 0 == go on */
    PBS_Command = 1;
    command_mutex->release();
    // Wait for other search engine to stop
    event->wait();
  }
  // If not, wait for other search engine to return
  else{
    event->wait();
    solution_mutex->acquire();
    returned = queue->pop();
    solution_mutex->release();
  }
  return returned;
}
```

Figure 4.10: Code segment illustrating the third implementation of the parallel structure in parallel `PBS:next()` where all solutions can be found (also solving all race conditions).

engines.

The changes in the `run_wrapper` class are illustrated by figure 4.11. Other than the constructor for the `run_wrapper` class expanding to take pointers to the two global counters, a new if-else clause has been added after the existing one. The new clause checks whether the search engine was stopped or not and increments the corresponding global counting variable. Apart from that the calls to the `solution_mutex` have been moved out from the first clause to instead envelop all

of the clauses to ensure safety when accessing these shared variables.

```
class run_wrapper : public Support::Runnable {
  public:
    ...
    run() {
      result = e.next();
      solution_mutex->acquire();
      if(result != NULL) {
        queue->push(result);
      }
      if(e.stopped() == false) {
        finished_threads++;
      }
      else {
        stopped_threads++;
      }
      solution_mutex->release();
      event->signal(); // Signal main thread
    }
};
```

Figure 4.11: Code segment illustrating the third version of the `run_wrapper` class with mutex-protected access to a solution queue. Adding to that are two global counters, `finished_threads` and `stopped_threads`, that helps `PBS:next()` to keep track of how many threads have stopped for what reason.

Figure 4.12 shows the fourth implementation of the `PBS:next()` method where it can find all solutions and also run with any number of search engines in a parallel fashion. The first change is the creation of `run_wrapper` objects for each search engine and starting them in their threads that now happen inside a for-loop that iterates over the vector of search engines. When all search engines have been started in their own threads the main while-loop is entered. Inside it the `PBS:next()` method first wait for the event. When it receives a signal it checks the queue for a solution. Then it checks if all threads have either finished or been stopped. If that is the case `PBS:next()` resets the counting variables and returns the solution (or `NULL` if no solution exist, which indicates that all search engines have exhausted their search trees). But if not all threads have finished or been stopped `PBS:next()` goes to the next clause and checks if the found solution is in fact a solution or `NULL`. If it is a solution `PBS:next()` sets the `PBS_Command` variable to 1 indicating that all other search engines should stop. It then enters an inner while-loop where it waits for all search engines to stop before it returns the found solution to the user. This structure will leave residue signals on the event but the outer while-loop will just consume them during the next call to `PBS:next()` thanks to the else-clause inside it at the end, thus eliminating any potential harm they could otherwise have caused.

```
next() {
  ...
  // Create run_wrappers for every engine
  // and start them in separate threads.
  for( i = 0; i < NR_THREADS; i++ ) { ... }
  while(true) {
    event->wait();
    solution_mutex->acquire();
    returned = queue->pop();
    if( (finished_threads + stopped_threads)
        == NR_THREADS ) {
      // Reset values and exit
      finished_threads = 0; stopped_threads = 0;
      solution_mutex->release();
      break;
    }
    else if(returned != NULL) {
      command_mutex->acquire();
      PBS_Command = 1;
      command_mutex->release();
      solution_mutex->release();
      while(true) {
        solution_mutex->acquire();
        if( (finished_threads + stopped_threads)
            == NR_THREADS ) {
            // Reset values and exit
            stopped_threads = 0; finished_threads = 0;
            command_mutex->acquire();
            PBS_Command = 0;
            command_mutex->release();
            solution_mutex->release();
            break;
          }
        solution_mutex->release();
        Gecode::Support::Thread::sleep(10);
      }
      break;
    }
    else { solution_mutex->release(); }
  }
  return returned;
}
```

Figure 4.12: Code segment illustrating the fourth implementation of the parallel structure in parallel `PBS:next()` where any number of search engines can be run with PBS.

### 4.2.7 Global Stop Object for PBS

Parallel PBS also needs to be able to have a global stop object. And since it already has a modified stop object class of its own it makes sense to just build onto that instead of creating a new stop object wrapper class as in the sequential variant of PBS. In addition to changes in the `ModdedStopObject` class and some minor changes in the `PBS:next()` method a global shared variable called `PBS_stopped` is also created. This global variable is needed in order to communicate from inside the `ModdedStopObject` out to PBS that the global stop object has returned true and is stopping PBS.

Figure 4.13 highlights the added parts of the `ModdedStopObject` class. The new part consists of two nested clauses where the outermost one first performs a validity check that a global stop object actually exists. If so the global stop object's `stop()` method is called and if it returns true PBS should stop. If so it then sets the global shared variable `PBS_stopped` to true and then returns true to the search engine. If not, the `ModdedStopObject` return false to the search engine.

```
class ModdedStopObject : public Search::Stop {
  ...
  bool stop(const Search::Statistics& s,
            const Search::Options& o) {
    command_mutex->acquire();
    ...
    if(stop_object != NULL) {
      stopped = stop_object->stop(s,o);
        if(stopped == true) {
          PBS_stopped = true;
          command_mutex->release();
          return true;
        }
    }
    command_mutex->release();
    return false;
  }
  ...
};
```

Figure 4.13: Code segment illustrating the second implementation of the modified stop object class which gives PBS the ability to have an optional global stop object.

In figure 4.14 the changes to the `PBS:next()` method are highlighted. This part is added right after the `PBS:next()` method checks if there are any found solutions stored in the queue, and right before it creates all `run_wrapper` objects with search engines inside them and starting them in their threads. The added code resets the value of `PBS_stopped` to false so it always has the same starting conditions. Should PBS have been stopped by its global stop object and the user decides to call

on `PBS:next()` again without updating the global stop object it will return `NULL` because the global stop object will return true again when called before the search engines start to explore new nodes.

```
next() {
  ...
  // Set PBS stopped status to false.
  command_mutex->acquire();
  PBS_stopped = false;
  command_mutex->release();
  ...
}
```

Figure 4.14: Code segment illustrating the fifth implementation of the parallel structure in parallel `PBS:next()` where any number of search engines can be run with PBS with the addition of an optional global stop object.

### 4.2.8 Other Methods in PBS

The other methods that PBS has are `statistics()`, `nogoods()` and `stopped()`. The `statistics()` method just adds each of the search engines own statistics into one collected statistics object and then returns it up to the user. The `nogoods()` method was deemed unnecessary for this prototype. Therefore it has not been implemented so it just returns `NULL` whenever the user calls it. Thanks to the global shared variable `PBS_stopped` the `PBS:stopped()` method just returns the value of that variable. If no global stop object has been given to PBS false is returned.

### 4.2.9 Avoiding Deadlocks

The implementation is (hopefully) completely deadlock free as long as nothing from the outside interferes with the structure, such as the operating system suddenly terminating a slave thread which is currently holding a mutex or likewise. There is one signaling event and two mutexes, `command_mutex` & `solution_mutex`, in the code that could possibly cause deadlocks if not used correctly. Remember that if one thread holds a mutex and another thread tries to acquire it, the other thread will be blocked until the mutex is released. Following is a description focusing on how the event and mutexes are used to avoid deadlocks.

First the `PBS:next()` method acquires the `solution_mutex` to safely check the solution queue and then releases it again. If no leftover solutions were in the queue `PBS:next()` continues with acquiring the `command_mutex` to reset some global variables and then releases the mutex again. All this is done before any threads are started and should be completely safe (using the mutexes just in case some residue thread against all odds is still there). Then the threads are started and `PBS:next()`

quickly consumes all residue signals on the event, thanks to the code structure, before it waits for the threads to signal the event.

Each thread will only acquire and release the `command_mutex` during search (inside the controlling stop object). It is guaranteed by the code structure that the stop object releases the mutex before returning its result. When search stops and a solution (or `NULL`) is returned it will not try to acquire the `command_mutex` again. Instead the thread now acquires the `solution_mutex` once for storing the found solution in the solution queue, update one of the two global counting variables and finally release the `solution_mutex` again. After that the thread signals the event and terminates.

When `PBS:next()` is released from the event (by a signal from one of the threads), it acquires the `solution_mutex` and checks the solution queue. If no solution is present and not all threads are terminated it just releases the mutex and waits on the event again. If all threads have terminated it releases the mutex and returns the result back to the user. But if it received a solution *and* not all threads have terminated, `PBS:next()` will try to acquire the `command_mutex` and change the `PBS_Command` to 1 to stop all other threads. It then first releases the `command_mutex` and then releases the `solution_mutex`. After that it tries to acquire the `solution_mutex`, check if all threads have terminated, releases the mutex again and sleeps for 10 milliseconds. This is repeated until all threads have terminated. When that happens, `PBS:next()` can easily acquire both mutexes, reset some of the global variables, release both mutexes and finally return the result to the user.

This implementation prevents PBS from getting into a deadlock as all mutexes are guaranteed to always be released again eventually. Also all threads will eventually terminate as their search engines return a result, which leads to less competition on the mutexes for the rest of the threads.

# Chapter 5

# Experimental Evaluation

This chapter explains how the case studies described in subsection 2.5.1 and subsection 2.5.2 were conducted. It also presents the results and discusses them. The $N$-Queens case study is covered in section 5.1 where it presents the test setup, how the tests were performed and end with a discussion of the results. All to verify the soundness of the design and implementation of sequential PBS and parallel PBS. The Latin Square case study is then covered in section 5.2 where it presents the different test setups, how the tests were performed and end with a discussion of the results. All to find out the performance of sequential PBS and parallel PBS and compares their results with that of a single search engine variant that Gecode already has implemented.

**Note:** The test setup used in [7] influenced the design of the tests in this thesis. That is why the seemingly odd numbers of 2, 5, 10 & 20 search engines are used, instead of the more logical 2, 4, 8 & 16 that matches the number of cores in a CPU. This was done to be able to compare this thesis's results with the results presented in the paper.

**Note:** All search engines used in the case studies use the depth-first search strategy (DFS).

### Machine information

KTH (Royal Institute of Technology) provides all students with access to three servers called Atlantis, Colombiana and Subway. It were these three servers that all experiments were run on. Due to all students having access to these servers it was hard to guarantee that the experiments had full access to the required number of CPUs necessary without outside influence. But a best effort was made to try running the experiments when the servers were unoccupied. The servers are provided with GCC compilers of version 4.4.2. The servers operation system is Red Hat Enterprise Linux Client release 5.11 (Tikanga). Their brand & model are HP Proliant DL385 G7 with 64GB of RAM and each server has 2×AMD Opteron 6172, 2.1GHz CPUs (12 cores each). The full specifications of the cores are found in appendix A figure A.1.

## 5.1 Case Study: N-Queens

As told in subsection 2.5.1 the purpose of the $N$-Queens case study was to verify the soundness of the design and implementation of the two PBS variants. That is, to verify that no solutions are ever lost even when producing very large amounts of solutions fast. The case study used Gecode's own implemented model of the $N$-Queens problem to ensure that it was sound.

### Test Setup

The setup was that both sequential PBS and parallel PBS were to run on a large-enough problem to generate a very large amount of solutions. The problem size was decided to be $N$=14 which would give 365'596 solutions to find for each search engine.

Table 5.1 summarizes the setup of the conducted tests. First the $N$-Queens problem was run with $N$=14 with a single search engine (without portfolio search) using the already implemented variant in Gecode. As the program is sound only one single run was needed to get all possible solutions. Then each of the sequential and parallel variants of PBS were run ten times each with two, five and ten search engines respectively. Each test run had its own log file to which every found solution was printed in order to be able to count them later. As each search engine given to PBS is sound in itself it was only the structure of the PBS variants that could potentially drop solutions. All found solutions were later verified by comparing them to the found solutions of the single search engine run without portfolio search as well as counted to ensure that all solutions were found.

Table 5.1: The table shows the test setup for sequential PBS and parallel PBS together with the single search engine (run without portfolio search) for the $N$-Queens problem.

| | Number of Search Engines | Number of Runs | N-value |
|---|---|---|---|
| Normal | 1 | 1 | 14 |
| Sequential PBS | 2, 5 & 10 | 10 | 14 |
| Parallel PBS | 2, 5 & 10 | 10 | 14 |

### Results

Table 5.2 shows the test results of the $N$-Queens problem. In the case of the sequential PBS and parallel PBS, the total number of solutions found during the ten runs for each number of search engines (two, five and ten) have been summed up. Dividing the sum total with the number of runs (to get the average) and dividing the result with the number of solutions found by the single search engine (run without portfolio search), which should end up as the number of search engines used, if all solutions were found.

Table 5.2: The table shows the results of the $N$-Queens problem for sequential PBS and parallel PBS. One run with Gecode's single search engine variant was also done to compare against.

| | Number of Search Engines | Runs | Solutions Found | $\frac{Solutions\,Found}{Runs \times 365'596}$ |
|---|---|---|---|---|
| Normal | 1 | 1 | 365'596 | 1 |
| Sequential PBS | 2 | 10 | 7'311'920 | 2 |
| | 5 | 10 | 18'279'800 | 5 |
| | 10 | 10 | 36'559'600 | 10 |
| Parallel PBS | 2 | 10 | 7'311'920 | 2 |
| | 5 | 10 | 18'279'800 | 5 |
| | 10 | 10 | 36'559'600 | 10 |

**Discussion**

As you can see in table 5.2 the values in the second leftmost column are identical with the values in the rightmost column. This proves that no solutions were ever dropped during these tests. Thus the design and implementation of sequential PBS and parallel PBS is proved to be sound in the aspect of never losing solutions.

## 5.2 Case Study: Latin Square

As told in subsection 2.5.2 the purpose of the Latin Square case study was to test the performance of the sequential PBS and parallel PBS against an already implemented Latin Square example in Gecode (which was run without portfolio search). This case study is divided into two parts. The first part which is covered in subsection 5.2.1 examines the overhead of the two PBS variants by running them on a number of different tests of different sizes. The second part which is covered in subsection 5.2.2 examines the performance of the PBS variants by running them on problem instances that Gecode's single search engine Latin Square example has problems with. The case study used Gecode's own implemented model of the Latin Square problem to ensure that it was sound. Gecode's single search engine variant is hereinafter shortened down to the *normal* variant (which is run without portfolio search). All Latin Square tests use Gecode's implemented `Timer` class which represents a best-effort at measuring wall-clock time in milliseconds to measure the execution times.

**Note:** The actual problem's correct name is quasigroup completion problem (QCP) but the problem data is encoded as partially filled Latin Squares and the produced solutions are Latin Squares (as explained in subsection 2.5.2). Therefore all graphs are labeled with QCP because it is shorter and also the correct term for the actual problem while the case study itself is named after the use of Latin Squares.

### 5.2.1 Measuring Overhead

To give a good approximation of the overhead of the sequential PBS and parallel PBS they needed to run on many different problem instances with a varying number of search engines and then be compared to the normal variant.

#### Test Setup

In order to nullify the advantage of running multiple search engines the search engines used by sequential PBS and parallel PBS had to have the exact same search strategies and random seeds as the normal variant. Otherwise one of the search engines might be faster than the others due to a better random seed or branching strategy and that would give a false measurement of the overhead.

Table 5.3 illustrates the settings for the normal variant, the sequential PBS and the parallel PBS for each problem instance. The settings where chosen such that they would be as powerful as possible with no randomization at all. As explained in subsection 2.5.2 the Distinct propagator is stronger than the Binary one and the AFC_Size brancher is stronger than the one using Size. Therefore Distinct and AFC_Size were chosen. No tie-breaking or random seeds were used to remove randomness completely. The original branching implementations were used and can be found in Appendix A figure A.2.

Table 5.3: The table shows the test setup for sequential PBS and parallel PBS together with the normal variant for the Latin Square problem.

| | Search engines | Propagation strategy | Branching strategy | Tie-breaking factor | Random seeds |
|---|---|---|---|---|---|
| Normal | 1 | Distinct | AFC_Size | None | None |
| Sequential PBS | 2, 5 & 10 | Distinct | AFC_Size | None | None |
| Parallel PBS | 2, 5 & 10 | Distinct | AFC_Size | None | None |

The tests were selected and divided into three groups of three, where each group had a different problem size. In each of the groups three problem instances with different difficulty (relative to the problem size) were chosen, one easy, one medium and one hard to get a good coverage. Also the hard problem in each group had no solutions to further test the PBS variants and measure their overhead when they had to explore the entire search tree. Table 5.4 shows which problem instances were used for the overhead measurement tests. It also shows their problem size, the number of preassigned values and percentage thereof, if it has any solutions and also the difficulty (in terms of execution time). All of the problem instances were taken from the Fourth International CSP Solver Competition's problem instance

archive[1].

Table 5.4: Table showing the problem instances for sequential PBS, parallel PBS and the normal variant as well as the characteristics of each problem.

| Problem instance | Size of $N$ | % of preassigned values | Solution exist? | Relative Difficulty |
|---|---|---|---|---|
| 10-67-0 | 10 | 32% | Yes | Easy |
| 10-67-5 | 10 | 30% | Yes | Medium |
| 10-67-10 | 10 | 31% | No | Hard |
| 15-120-0 | 15 | 44,9% | Yes | Easy |
| 15-120-5 | 15 | 44% | Yes | Medium |
| 15-120-10 | 15 | 42,7% | No | hard |
| 20-187-0 | 20 | 46,75% | Yes | Easy |
| 20-187-4 | 20 | 48,75% | Yes | Medium |
| 20-187-10 | 20 | 47% | No | hard |

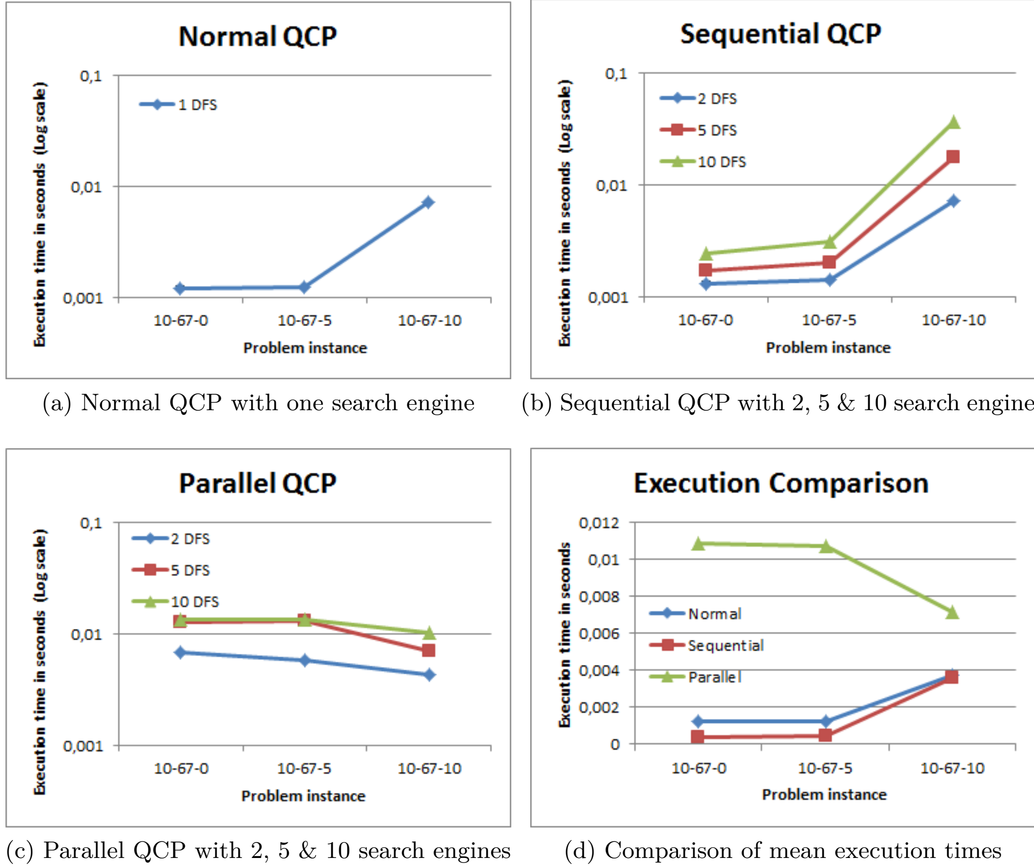
The expected results of the sequential variant of PBS are that the overhead is so small that it is barely detectable compared to the normal program. The expected results of the parallel variant of PBS are that the overhead for each search engine is small but increasing depending on the number of search engines used, as congestion is increased with more search engines.

### Results

The results from the first group where $N$=10 are shown in figure 5.1. Figure 5.1a shows the mean execution times of the normal variant's ten runs on each problem instance. Figure 5.1b and 5.1c shows the mean execution times of the sequential PBS and parallel PBS variants ten runs on each problem instance with 2, 5 & 10 search engines. Figure 5.1d shows the comparison in execution times between the normal and the combined mean execution times of the sequential PBS and the parallel PBS variants on the three problem instances. The combined mean was calculated from the mean execution times of running with 2, 5 & 10 search engines. Note that the execution times of the sequential variant have been divided by the number of search engines used before the combined mean was calculated in order to give a correct comparison. The same was not needed for the parallel variant because each of its search engines ran in parallel and did not affect the execution time from the others except for the overhead. It is worth noting that graphs 5.1a to 5.1c have their y-axes shown in Log scale while the y-axis in graph 5.1d is normal.

The results from the second group where $N$=15 are shown in figure 5.2. The four graphs are set up the same way as the ones for the first group with 5.2a showing

[1] Their website: `http://www.cril.univ-artois.fr/CSC09/` where the problem instances can be accessed.




(a) Normal QCP with one search engine

(b) Sequential QCP with 2, 5 & 10 search engines

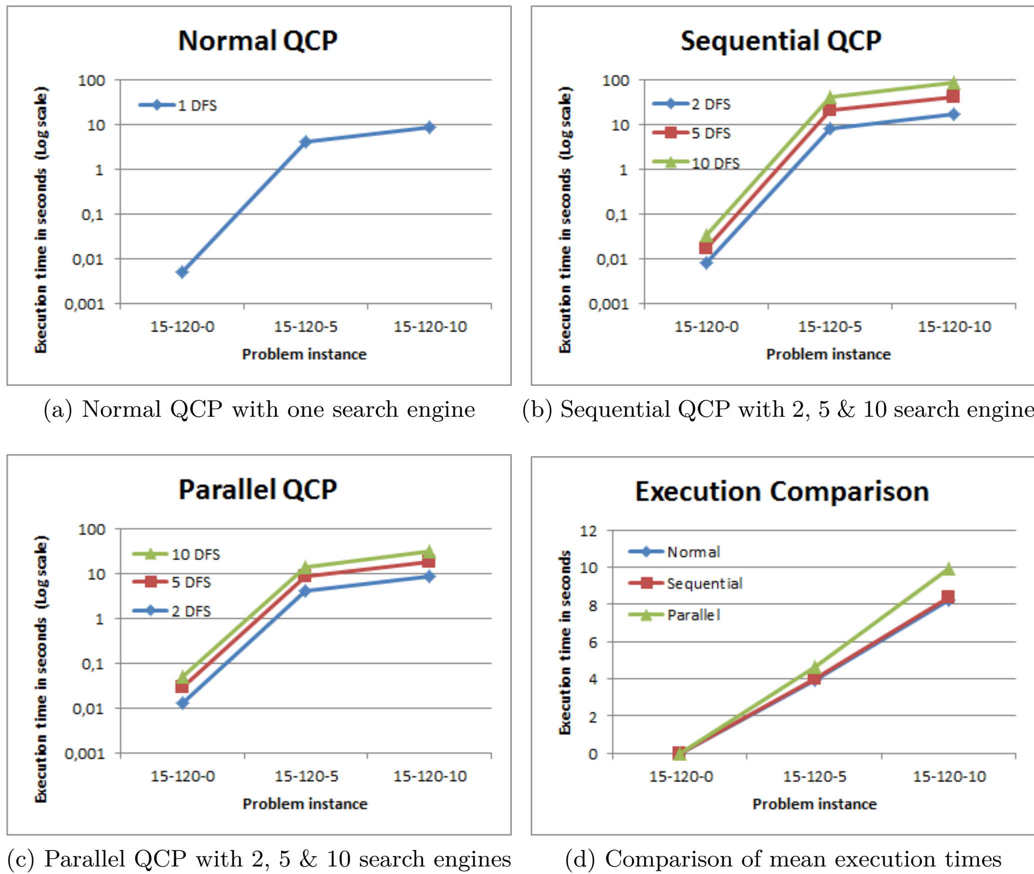
(c) Parallel QCP with 2, 5 & 10 search engines

(d) Comparison of mean execution times

Figure 5.1: Graphs showing the execution times for the first group with $N$=10. They are accompanied by a graph that compares their mean execution times where the sequential variant has been scaled accordingly to give a correct comparison.

the results for the normal variant while 5.2b and 5.2c are showing the results for the sequential and parallel PBS variants. Graph 5.2d shows the comparison of their combined mean execution times (calculated the same way as for the first group).

The results from the third group where $N$=20 are shown in figure 5.3. The four graphs are set up the same way as the ones for the first and second groups with 5.3a showing the results for the normal variant while 5.3c is showing the results for the parallel PBS variant. Graph 5.3d shows the comparison of their combined mean execution times (calculated the same way as for the first and second groups). Graph 5.3b only has parts of the results because some of the experiments had to run so long that the server terminated the experiments before they had finished every time. The data points in the graph that are created from actual data are all three points for the run with 2 search engines and all points for problem instance 20-187-0. The rest are synthesized from the other data points to give an estimate on




(a) Normal QCP with one search engine (b) Sequential QCP with 2, 5 & 10 search engines

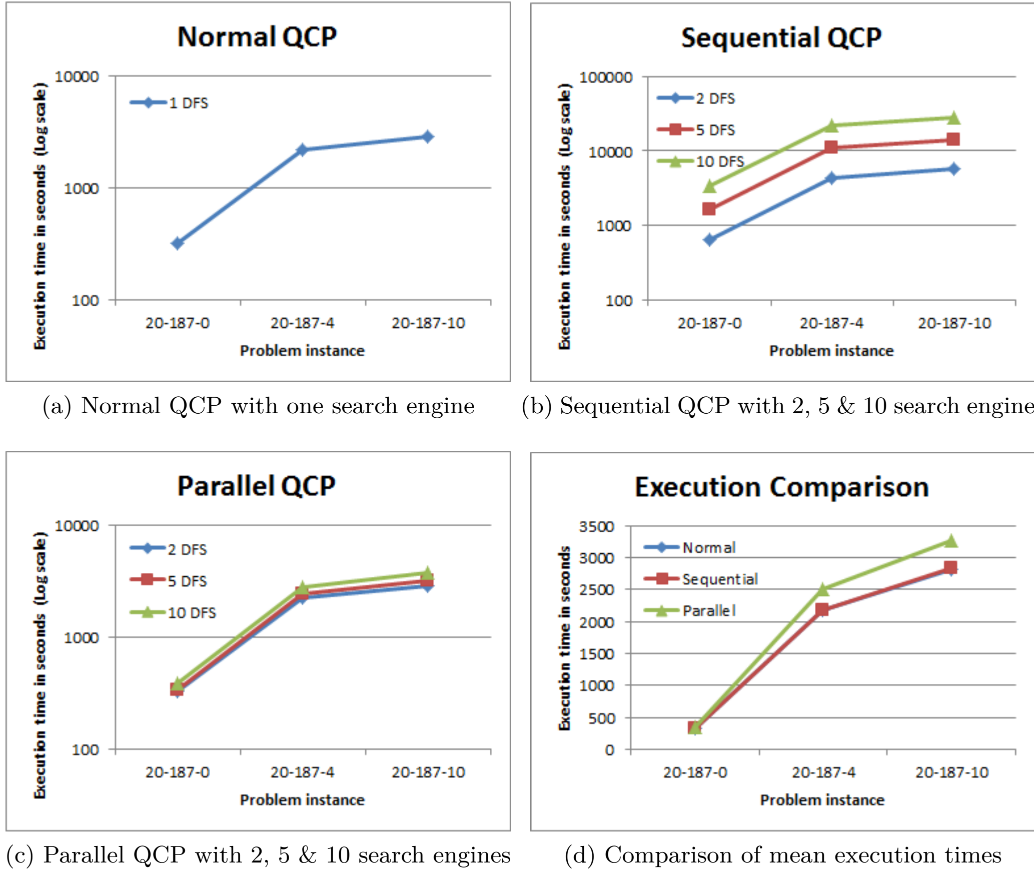
(c) Parallel QCP with 2, 5 & 10 search engines (d) Comparison of mean execution times

Figure 5.2: Graphs showing the execution times for the second group with $N$=15. They are accompanied by a graph that compares their combined mean execution times where the sequential variant has been scaled accordingly to give a correct comparison.

how sequential PBS was expected to behave. Therefore the combined mean values for the sequential variant in graph 5.3d are also based on the synthesized values from graph 5.3b.

### Discussion

The results of the first group were surprisingly odd, especially the results for the parallel variant. The most likely reason for this is that the problem instances were all too easy and had such short solution times (in milliseconds) that the actual time it took to solve the problem instance was overshadowed by the overhead, especially for the parallel variant. Also the execution time for the sequential variant was actually lower than the normal program, which should not have been possible in this setup with identical search engines, which suggests that only the first search




(a) Normal QCP with one search engine

(b) Sequential QCP with 2, 5 & 10 search engines

(c) Parallel QCP with 2, 5 & 10 search engines

(d) Comparison of mean execution times

Figure 5.3: Graphs showing the execution times for the third group with $N$=20. They are accompanied by a graph that compares their mean execution times where the sequential variant has been scaled accordingly to give a correct comparison.

engine got to run and found a solution during its first time slice. Adding to that was the precision of the timer used that measures wall-clock time in milliseconds, which could also be a source of small measurement errors. Thus no real conclusions can be drawn from the results from the first group.

The results from the second group were more promising as they were closer to the expected results. Even though the execution time is still very short it aligns with the expected results. Graph 5.2b shows that for the sequential variant of PBS when the number of search engines increase so does the execution time and the relation between them is approximately linear. Graph 5.2c shows that the parallel variant of PBS has more overhead than the sequential variant of PBS but seems to also be approximately linear. Looking at graph 5.2d strengthens the expected results for the sequential variant of PBS by showing that the overhead is so small that it actually starts to overlap the upper part of the normal variant's line, partly

covering it in the graph. As for the parallel variant of PBS it seems that it has a percentage of overhead related to the number of search engines and that that percentage increases as the number of search engines increases. This suggests a non-linear overhead depending on the number of search engines, which is to be expected due to the bottleneck in the master-slave structure.

The results from the third group continue to strengthen the expected results in the same way as the second group did. As expected for the sequential variant of PBS the execution time is linear to the number of search engines used for each problem instance, which is shown in graph 5.3b. Graph 5.3c shows the same trend as shown in the corresponding graph in the second group, that the overhead seems to be approximately linear for the parallel variant of PBS. Looking at grapg 5.3d continues to strengthen the expected results for the sequential variant of PBS by showing the same trend as in graph 5.2d. But looking at the parallel variant of PBS the same worrying trend can be seen with the overhead increasing with the number of search engines in a seemingly non-linear fashion.

Using the data from the three groups an initial attempt to calculate the overhead for the sequential variant and parallel variant was done. But due to the limited data points only estimated values were calculated. The overhead of the sequential variant was estimated to be around 0.5% independent of how many search engines were used. The overhead of the parallel variant was estimated to be around 1-3% per search engine, depending on how many search engines were used. This would (in theory) quickly add up to a significant amount of overhead and might have a negative impact on the performance. Further experiments are needed to accurately determine the relation between the overhead and the number of search engines for the parallel variant of PBS.

The high percentage of overhead in the parallel variant of PBS could be caused by the `command_mutex` used in the controlling stop object. As of now in the (prototype) parallel variant of PBS the `command_mutex` is acquired and released for *every node* explored in the search tree. To counter this an easy fix could be to modify it so that the controlling stop object only acquires and releases the `command_mutex` for every 100, 1'000 or even 10'000 nodes explored, thus reducing the congestion by the same factor (in theory). This would of course affect the time it takes for a search engine to see a new command sent from the master thread. Unfortunately due to time constraints there was no time left to test this in practice.

### 5.2.2 Measuring Performance

To measure the performance of the sequential PBS and parallel PBS variants they needed to run on problem instances that the normal variant found really hard. As problem instances with $N$=10 were way too easy only problem instances with $N$=15 and $N$=20 were considered. But to really see what the sequential variant and parallel variant of PBS is capable of the branching strategies had to be changed to introduce more randomness. Also the use of a tie-breaking factor would further randomize the branching strategies as well as using random seeds. The modified

branching implementations that were used can be found in Appendix A figure A.3.

### Preparation Tests

To find out which problem instances are hard for the normal variant it was run 5 times each over 10 different problem instances, both with $N$=15 and $N$=20. The results are summarized in table 5.5. These problem instances were also taken from the Fourth International CSP Solver Competition's problem instance archive.

Table 5.5: Table showing 10 different problem instances for $N$=15 & $N$=20 with their mean execution times for the normal variant. So the most difficult problems (that finish within one hour) can be selected.

| | 15-120 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Mean(s) | 0,005 | 0,03 | 3,1 | 0,005 | 2,1 | 3,9 | 2,3 | 0,1 | 0,2 | 22,3 |
| | 20-187 | | | | | | | | | |
| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Mean(s) | 1,6 | 88,1 | 3600 | 97,2 | 2135 | 18,6 | 3,0 | 655 | 18,7 | 207 |

Table 5.5 shows that the hardest problem instance with $N$=15 is number 9 with a mean execution time of 22,3 seconds. Therefore the chosen problem instance to use for $N$=15 was 15-120-9. As for the problem instance with $N$=20 there are two candidates, number 2 and number 4. Number 2 timed out after one hour and still had not found a solution. Number 4 was the hardest problem of which a solution could still be found within one hour of execution time. But to be able to correctly compare the results of the sequential and parallel runs the normal variant needed to be able to find a solution on the given problem instance within the given time limit. Therefore number 4 was chosen so the problem instance to use for $N$=20 was 20-187-4. Only one problem instance for each size were chosen due to time constraints, although more extensive tests would have been preferred.

### Test Setup

Table 5.6 shows the chosen settings for the search engines during the performance tests. Both the sequential PBS and parallel PBS variants ran with 5 respectively 10 search engines while the normal variant only ran with one search engine (without portfolio search). To optimize the effectiveness they all use Distinct for propagation strategy as well as both AFC_Size and Size with modified branchers for branching strategies. The modified branchers can be found in Appendix A figure A.3. Eleven different tie-breaking factor values were used ranging from 0.1 to 1.0 and also 0.0 (None) to give a good coverage of how much randomization was optimal for the two chosen problem instances. The implementations of the tie-breaking limit functions can be found in Appendix A figure A.4. A set of ten different random

seeds were generated[2] and used throughout all runs in the performance tests so that the only change was different tie-breaking factor values (for both AFC_Size and Size branching). This was done in order to measure how they performed with different tie-breaking factor values and to see if there was a "sweet spot". For each tie-breaking factor value the programs were run 5 times to give some measure of statistical support.

**Note:** Due to time constraints the three KTH servers' cores were 90-95% occupied with running all of the different performance tests at the same time and it was first after all tests were finished that the data was analyzed. That is why the obviously inferior Size branching strategy is present in all of the result graphs.

**Note:** The preferred way would have been to generate new random seeds for every search engine in each run on each of the configurations. But due to time constraints this was not feasible as it would require the tests to be run between 100-1'000 times for each setting in order to get statistical soundness.

Table 5.6: Table showing the test setup for sequential PBS, parallel PBS and normal single search engine program for the Latin Square problem for performance measurement on problem instances with $N$=15 & $N$=20.

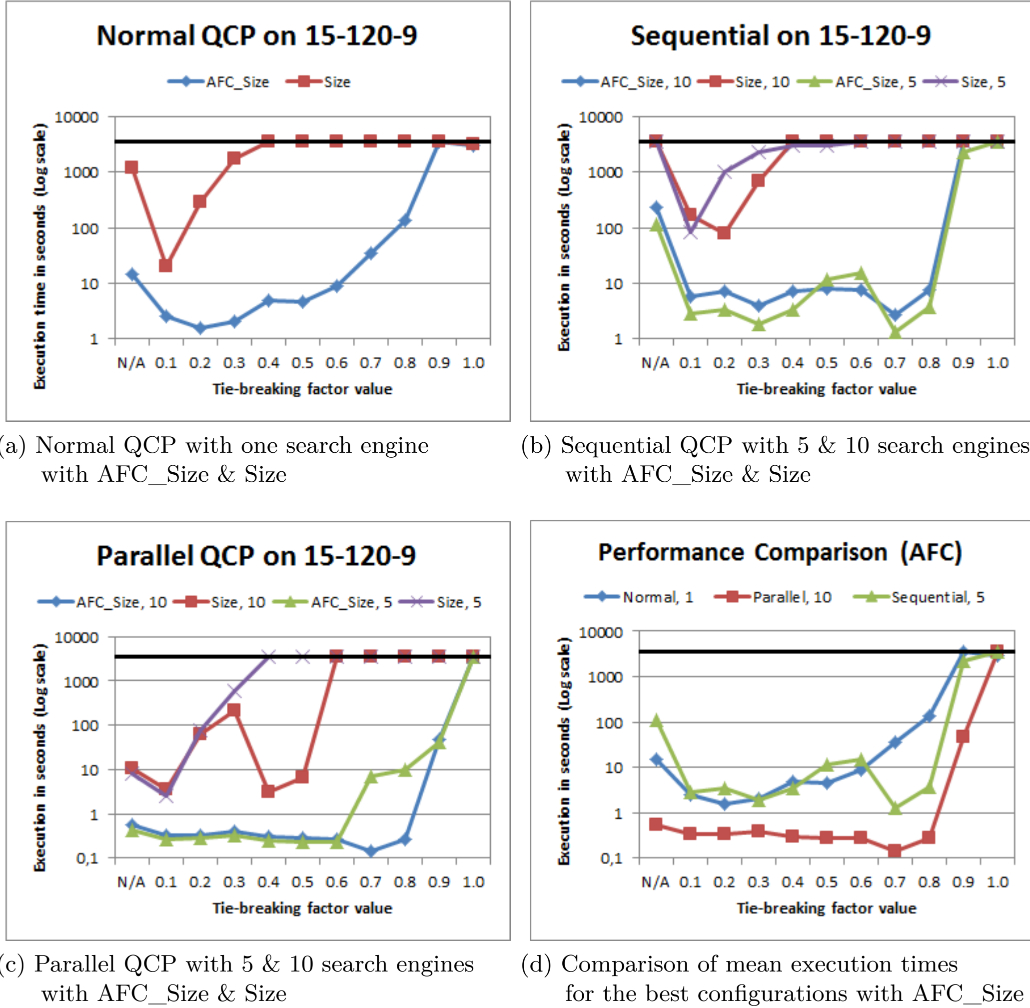
<table>
<tr><th></th><th>Search engines</th><th>Propagation strategy</th><th>Branching strategy</th><th>Tie-breaking factor values</th><th>Random seeds</th></tr>
<tr><td>Normal</td><td>1</td><td rowspan="3">Distinct</td><td rowspan="3">AFC_Size & Size</td><td rowspan="3">None, 0.1-1.0</td><td rowspan="3">Yes, using pre-defined seeds</td></tr>
<tr><td>Sequential PBS</td><td>5 & 10</td></tr>
<tr><td>Parallel PBS</td><td>5 & 10</td></tr>
</table>

### Results

The results from the tests running on problem instance 15-120-9 with $N$=15 are shown in figure 5.4. Figure 5.4a shows the mean execution times of five runs for the normal variant for each of the eleven tie-breaking factor values, both with AFC_Size and Size branching strategies. Figure 5.4b and 5.4c show the same for the sequential PBS and parallel PBS variants where they were run with 5 and 10 search engines on all different tie-breaking factor values, both with AFC_Size and Size branching strategies. A time-out of one hour execution time was used.

The results from the tests running on problem instance 20-187-4 with $N$=20 are shown in figure 5.5. The results are presented in the same manner as the previous results. Figure 5.5a show the execution results for the normal variant, using both

[2] The random seeds used were [1, 23, 102, 61, 77, 89, 59, 44, 143, 98] and were produced by the author randomly typing ten numbers on the keyboard. The normal variant used the first value, running with 5 search engines used the 5 first values and running with 10 search engines used all ten values.




(a) Normal QCP with one search engine with AFC_Size & Size

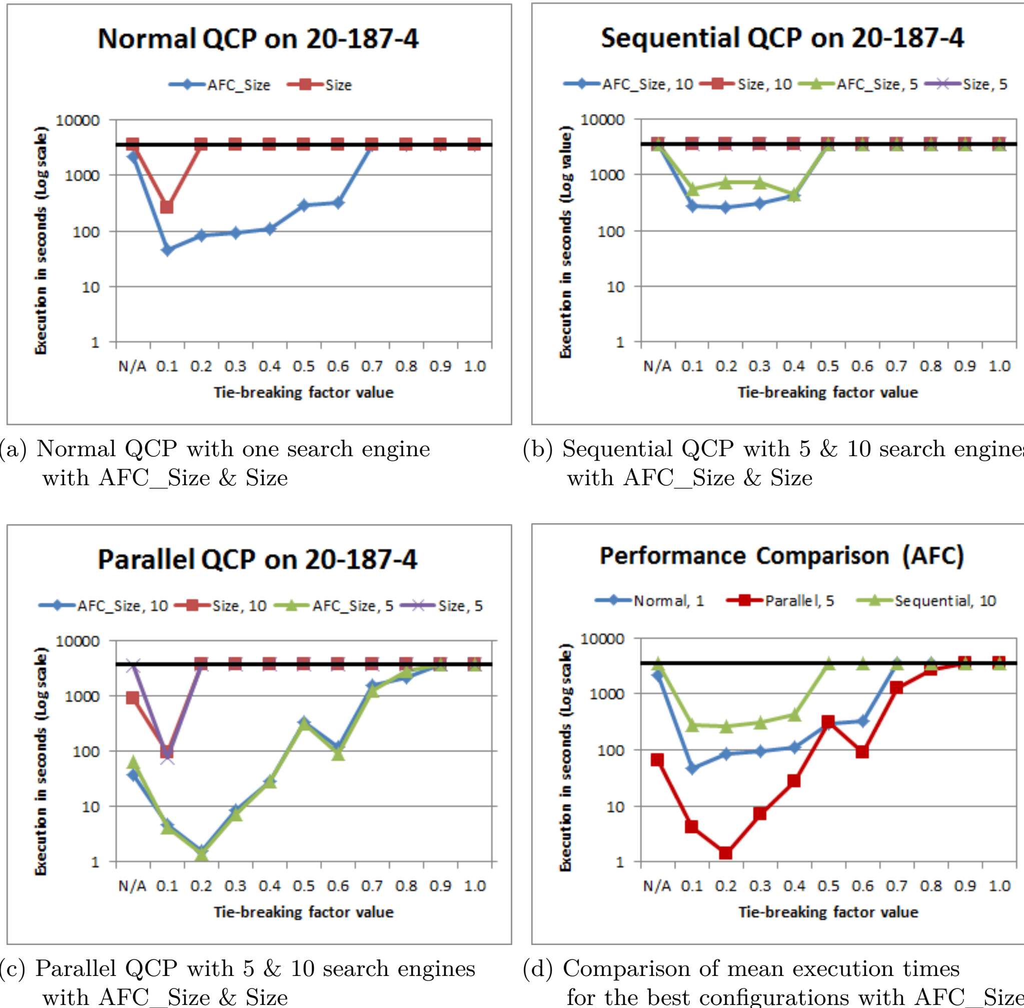
(b) Sequential QCP with 5 & 10 search engines with AFC_Size & Size

(c) Parallel QCP with 5 & 10 search engines with AFC_Size & Size

(d) Comparison of mean execution times for the best configurations with AFC_Size

Figure 5.4: Graphs showing the execution times for $N$=15. They are accompanied by a graph that compares the mean execution times of the PBS variants that performed the best. The black lines shows where the time limit is (3600s).

AFC_Size and Size for branching strategies. Figure 5.5b and 5.5c show the same for the sequential PBS and parallel PBS variants as they were run with 5 and 10 search engines using both AFC_Size and Size branching strategies on all tie-breaking factor values. A time-out of one hour execution time was used again, even though this problem was harder, in order to give a fair comparison to the results on the previous problem.




(a) Normal QCP with one search engine with AFC_Size & Size

(b) Sequential QCP with 5 & 10 search engines with AFC_Size & Size

(c) Parallel QCP with 5 & 10 search engines with AFC_Size & Size

(d) Comparison of mean execution times for the best configurations with AFC_Size

Figure 5.5: Graphs showing the execution times for $N$=20. They are accompanied by a graph that compares the mean execution times of the PBS variants that performed the best. The black lines shows where the time limit is (3600s).

### Discussion

First of all, as the two chosen problem instances have solutions these tests are not affected by the design decision that all search engines must exhaust their search trees before PBS can return `NULL`. Simply because it will not happen.

As can be seen in graphs 5.4a to 5.4c running with AFC_Size as branching strategy completely outclasses running with Size as branching strategy. It is interesting that the sequential variant of PBS running with 5 search engines and AFC_Size performs slightly better than with 10 search engines and that is why it is in the comparison graph. Graph 5.4d compares the best execution setup from graphs 5.4a

to 5.4c by plotting them together. It clearly shows that the parallel variant of PBS with 10 search engines is better than the normal variant or the sequential variant of PBS. Another observation is that the sequential variant of PBS with 5 search engines has approximately the same performance as the normal variant. However, this problem instance is still solved fairly quickly so the test runs does not show the true strength of the PBS variants.

Graphs 5.4a to 5.4c also show that the Size branching strategy timed-out when the tie-breaking factor value got larger than 0.3, indicating that there was too much randomness. The graphs also show that with tie-breaking factor value 1.0 all variants had huge problems with finding a solution inside the given time limit. The normal variants and the parallel variants with 5 search engines barely managed to find a solution with only minutes away from the time limit. This indicates that it was the search engine with random seed set to 1 that found the solution and that due to the overhead in the parallel variants only the ones running with 5 search engines managed to find it before the time limit, only because they had less overhead due to using fewer search engines. This also suggests that given more time the parallel variants running with 10 search engines would also soon find the solution as it was only the overhead that slowed them down. But it also indicates that they had almost reached total randomness and that even larger tie-breaking factor values would yield similar results.

As can be seen in graphs 5.5a to 5.5c running with AFC_Size as branching strategy still outclasses running with Size as branching strategy, even though the complete picture could not be given due to the time limit cutting off the graphs at the top. Looking closer on graph 5.5c you can see the interesting phenomenon that the parallel variants with 5 and 10 search engines using AFC_Size branching have almost exactly the same execution times. By looking at the raw data it is found out that the parallel variant running with 5 search engines perform around 5-10% better relative the parallel variant running with 10 search engines. This suggests that it is only the overhead of the additional 5 search engines that makes the difference and that it is one of the first five random seeds that lead to the solution. Graph 5.5d compares the best execution setup from graphs 5.5a to 5.5c by plotting them together. It clearly shows that both parallel variants of PBS with 5 & 10 search engines are better than the normal variant or the sequential variant of PBS. Another observation is that the parallel variant now outperforms the normal single search engine program on almost all tie-breaking factor values. Also the sequential variant of PBS with 5 & 10 search engines now have worse performance than the normal program.

Graphs 5.5a to 5.5c also show that the Size branching strategy timed-out when the tie-breaking factor value got larger than 0.1, clearly showing that the AFC_Size branching strategy is always stronger. The graphs also show the same trend as on the previous problem instance that, now with an even more difficult problem, regardless of branching strategy or number of search engines, none of the programs managed to find a solution in the given time limit when the tie-breaking factor value reached 1.0, indicating that they had almost reached total randomness and

that even larger tie-breaking factor values would yield the same results.

Due to the unexpected behavior of the parallel variant of PBS when measuring the overhead and the uncertainty of how much it affects the performance measurements, it was decided to perform one additional experiment focusing solely on the performance of the parallel variant of PBS with different numbers of search engines.

### Additional Test Setup

To be able to compare against the previous results in figure 5.5 as well as due to time constraints, it was decided to run the parallel variant of PBS on problem instance 20-187-4 again instead of a new problem instance. The configuration that was chosen was the most promising one from the previous tests and is summarized in table 5.7. The tie-breaking factor value was chosen based on how the normal variant had performed on the same problem to give an approximate comparison.

Table 5.7: Table showing the test setup for the additional experiment for the parallel variant of PBS.

| | Search engines | Propagation strategy | Branching strategy | Tie-breaking factor | Random seeds |
|---|---|---|---|---|---|
| Parallel PBS | 5, 10 & 20 | Distinct | AFC_Size | 0.1 | Total randomness |

Total randomness means that for each run new random seeds between 0-100 were generated for each search engine[3]. Each setting was run a 1000 times on the problem instance to give a good statistical foundation that would not be affected too much by "unlucky" random seeds.

### Additional Results

Figure 5.6 shows the results of running parallel PBS with the settings shown in table 5.7 a 1000 times. The primary axes (left & bottom) are for the mean and median curves for the execution time with its standard deviation while the secondary axes (right & top) are for the mean and median curves for the number of failures with its corresponding standard deviation. The curves of the execution time and the curves of the number of failures are overlayed to better show how they follow each other with different numbers of search engines.

Table 5.8 complements the results in figure 5.6 by showing the min and max values of the 1000 runs for each number of search engines, both for execution time and number of failures. This is to illustrate how much it can vary if one is extremely lucky (or unlucky) with the random seeds.

[3]The ten random seeds previously used were used as seeds for the random number generators used here. At the time the author was not aware of hardware generated "true" random seeds.

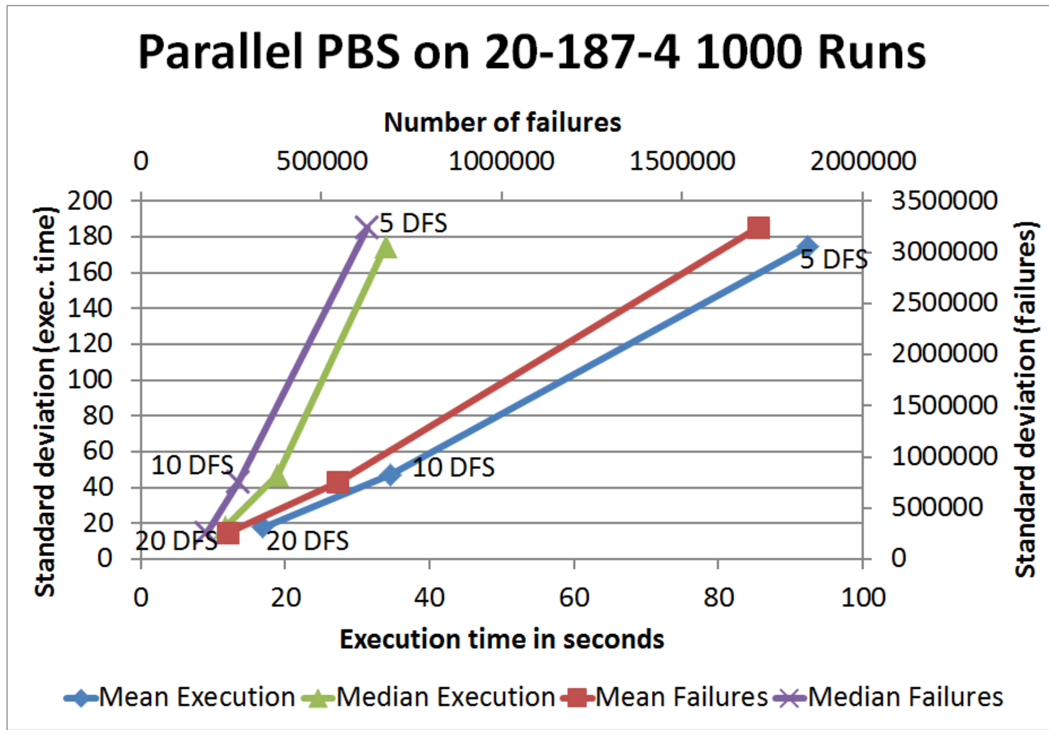



Figure 5.6: Graph showing the results of running parallel PBS a 1000 times with 5, 10 & 20 search engines on problem instance 20-187-4. The main x-axis (bottom) shows the execution time and the main y-axis (left) shows the corresponding standard deviation. The secondary x-axis (top) shows the number of failures and the secondary y-axis (right) shows the corresponding standard deviation. Completely random values were freshly generated for each run.

Table 5.8: Table showing the min and max values from the results for the additional experiment for the parallel variant of PBS.

| | Execution Time in seconds | | | Number of Failures | | |
|---|---|---|---|---|---|---|
| | 5 DFS | 10 DFS | 20 DFS | 5 DFS | 10 DFS | 20 DFS |
| Min | 1,73693 | 1,88137 | 1,96804 | 33386 | 33386 | 33386 |
| Max | 1619,34 | 399,327 | 173,479 | 29216707 | 5904173 | 2160718 |

### Additional Discussion

Figure 5.6 clearly shows that when using completely random seeds the performance improves when the number of search engines increases. Both the mean and median curves were plotted to show what can normally be expected but that there are still situations where the user is unlucky with the random seeds and get worse performance than that of the normal variant. When the results are compared to

the results of the normal variant (run only with random seed=1) which had an average execution time of around 46 seconds it shows that running with less than 10 search engines in parallel will most likely not improve the execution time (if not having one of the search engines run with random seed=1, thus guarding against bad random seeds). But the comparison is not quite fair as the normal variant did not run with different random seeds and is thus just approximative.

But as table 5.8 shows even running with only 5 search engines in parallel could improve the execution time but is more likely to worsen it. Looking at 20 search engines the expected improvement is about 3 times as fast in terms of execution time (read from the graph) while the worst and best case scenarios are about 4 times as slow and about 23 times as fast, in terms of execution time (read from the table). This makes it an attractive choice as the benefits are great and the worst possible backlash is “only” about 4 times worse (and highly unlikely).

But remember that this prototype of the parallel variant of PBS has some problems with the overhead (as mentioned earlier at the end of the overhead measurement discussion). By addressing the overhead problem by implementing the suggested solution the performance is expected to increase even further, possibly making it worth the time to run even with only 5 search engines.

It is worth mentioning that this is just run with a single problem with the same tie-breaking factor value to be able to approximately compare against the results of the normal variant (found in graph 5.5a). A more extensive study is necessary to generalize these findings and also to run the normal variant with random seeds in the same fashion as this additional experiment to get a more accurate comparison.

# Chapter 6

# Conclusions

This thesis aimed to answer the question "*What is a good design and implementation of portfolio-based search in Gecode?*". By researching and testing different designs an implementation was settled upon. The goal of the implementation was only to produce a prototype which then was tested in several aspects in chapter 5. The tests were designed to see if the design and implementation of portfolio-based search in Gecode was sound.

There are five major conclusions that can be drawn from the results presented in chapter 5 which are shown in the following list:

1. Design and implementation of portfolio-based search in Gecode is sound for both the sequential and parallel variants.
2. No solutions are ever lost during execution of sequential or parallel portfolio-based search.
3. Overhead in the sequential variant and the parallel variant of portfolio-based search is small enough to not affect the execution time in any significant manner.
4. The parallel variant of portfolio-based search is capable of finding solutions faster than a normal single search engine approach.
5. The sequential variant of portfolio-based search is not as effective as its parallel counterpart. It is recommended to consider using restart based search instead, or a combination thereof.

The first conclusion is by far the most important as it is the main topic of this thesis. The other four conclusions are different parts of the first, illustrating important features and discoveries. The second conclusion is a fundamental requirement for the design and implementation to be considered sound. If the implementation would occasionally drop solutions found by their search engines the design would have serious flaws. The third conclusion also strengthens that the design and implementation of portfolio-based search is sound. If the overhead would be too great it would diminish the benefits of using multiple search engines. While the fourth conclusion verify the assumption that running several search engines in parallel is

better than any single search engine the same cannot be said for the sequential variant of portfolio-based search.

## 6.1 Future Work

Future studies specifically following this thesis's steps would be to continue working on and improving the design and implementation of portfolio-based search in Gecode as it is still only a prototype. To fully integrate it with Gecode in the long run would need a lot of extra documentation and testing, which was a far-fetched last optional goal in this thesis. Considering to implement a branch-and-bound variant where solutions could be injected into a search engine from the outside so that search engines could share progress between each other is also an interesting area to research. Also looking at different ways to implement the use of sharing no-goods between search engines when running branch-and-bound or restart-based search could be interesting.

As the hardware industry continues to produce processors with more and more cores the future for portfolio-based search is bright and promising. Therefore it would be interesting to look at how the implementation of portfolio-based search in Gecode scales when run on a computer cluster with several hundreds of available cores and perhaps adapt the communication protocol to handle such vast numbers of threads.

# Appendix A

# Case Study Extras

## A.1 Server Specifications

```
processor         : 0               vendor_id         : AuthenticAMD
cpu family        : 16              model             : 9
model name        : AMD Opteron(tm) Processor 6172
stepping          : 1               cpu MHz           : 2100.053
cache size        : 512 KB          physical id       : 2
siblings          : 12              core id           : 5
cpu cores         : 12              apicid            : 43
fpu               : yes             fpu_exception     : yes
cpuid level       : 5               wp                : yes
flags             : fpu vme de pse tsc msr pae mce cx8 apic sep
    mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht
    syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm 3dnowext
    3dnow rep_good constant_tsc nonstop_tsc amd_dcm pni cx16
    popcnt lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a
    misalignsse 3dnowprefetch osvw ibs skinit wdt nodeid_msr
bogomips          : 4200.48         TLB size          : 1024 4K
    pages
clflush size      : 64              cache_alignment : 64
address sizes     : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate [8]
```

Figure A.1: Specifications for the three servers that all experiments were run on. The servers each have 2×AMD Opteron 6172, 2.1GHz CPUs (12 cores each). Each core has the specifications shown in this figure.

## A.2 Branching Implementations & Tie-Breaking

```
...
if (tbf > 0.0) {
 Rnd r(opt.seed());
 switch (opt.branching()) {
 case BRANCH_SIZE:
  branch(*this, e, tiebreak(INT_VAR_SIZE_MIN(&tbl_min),
                            INT_VAR_RND(r)),
         INT_VAL_MIN());
  break;
 case BRANCH_AFC_SIZE:
  {
    IntAFC afc(*this, e, opt.decay());
    branch(*this, e, tiebreak(
                  INT_VAR_AFC_SIZE_MAX(afc, &tbl_max),
                  INT_VAR_RND(r) ),
           INT_VAL_MIN());
    break;
  }
 }
} else {
 switch (opt.branching()) {
 case BRANCH_SIZE:
   branch(*this, e, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
   break;
 case BRANCH_AFC_SIZE:
   branch(*this, e, INT_VAR_AFC_SIZE_MAX(opt.decay()),
                                         INT_VAL_MIN());
   break;
 }
}
...
```

Figure A.2: Code segment illustrating the original branching implementation used during the first part of the Latin Square case study when measuring the overhead of the two PBS variants.

```
...
if (tbf > 0.0) {
  Rnd r(opt.seed());
  switch (opt.branching()) {
  case BRANCH_SIZE:
    branch(*this, e, tiebreak(INT_VAR_SIZE_MIN(&tbl_min),
                              INT_VAR_RND(r)),
           INT_VAL_RND(r));
    break;
  case BRANCH_AFC_SIZE:
    {
      IntAFC afc(*this, e, opt.decay());
      branch(*this, e, tiebreak(
                       INT_VAR_AFC_SIZE_MAX(afc, &tbl_max),
                       INT_VAR_RND(r)),
             INT_VAL_RND(r));
      break;
    }
  }
} else {
  Rnd r(opt.seed());
  switch (opt.branching()) {
  case BRANCH_SIZE:
    branch(*this, e, INT_VAR_SIZE_MIN(), INT_VAL_RND(r));
    break;
  case BRANCH_AFC_SIZE:
    branch(*this, e, INT_VAR_AFC_SIZE_MAX(opt.decay()),
                                         INT_VAL_RND(r));
    break;
  }
}
...
```

Figure A.3: Code segment illustrating the modified branching implementation used during the second part of the Latin Square case study when measuring the performance of the two PBS variants.

```
...
/// Tie-breaking minimum limit function
double _tbl_min(double w, double b) const {
  assert(w >= b);
  return b + (w - b) * tbf;
}
/// Tie-breaking minimum limit function
static double tbl_min(const Space& home, double w, double b) {
  return static_cast<const QCP&>(home)._tbl_min(w,b);
}
/// Tie-breaking maximum limit function
double _tbl_max(double w, double b) const {
  assert(b >= w);
  return b - (b - w) * tbf;
}
/// Tie-breaking maximum limit function
static double tbl_max(const Space& home, double w, double b) {
  return static_cast<const QCP&>(home)._tbl_max(w,b);
}
...
```

Figure A.4: Code segment illustrating the tie-breaking limit functions used during the second part of the Latin Square case study when measuring the performance of the two PBS variants.

# Appendix B

# Source Code of Portfolio Search

## B.1 Source Code for Sequential Portfolio Search

```
1  #include <gecode/search.hh>
2
3  namespace Gecode {
4
5    template<class T>
6    class PBS {
7
8      class StopWrapper : public Search::Stop {
9
10       protected:
11         // Might change e_nodestop to generic stop object
               later
12         Search::NodeStop* e_nodestop;
13         Search::Stop* pbs_stop;
14         bool e_stopped;
15         bool pbs_stopped;
16         // pbs_stats is the total statistics of PBS
17         Search::Statistics pbs_stats;
18         // old_stats is the previous statistics of the
               current running engine.
19         // Needed to take out the difference between the
               previous and current statistics.
20         Search::Statistics old_stats;
21
22       public:
23         StopWrapper(Search::NodeStop* _nodestop,
               Search::Stop* _pbs_stop)
24         : e_nodestop(_nodestop), pbs_stop(_pbs_stop),
25         e_stopped(false), pbs_stopped(false) {
26           //nothing more to do
27         }
28
```

```
29      /// Copy over values from s to old_stats
30      void setOldStats(const Search::Statistics& s) {
31        old_stats.fail = s.fail;
32        old_stats.node = s.node;
33        old_stats.depth = s.depth;
34        old_stats.restart = s.restart;
35        old_stats.nogood = s.nogood;
36      }
37
38      /// Update the total statistics for the meta engine.
39      void update(const Search::Statistics& s) {
40        // Update pbs_stats with difference of
41        // previous and current statistics
42        pbs_stats.fail += s.fail - old_stats.fail;
43        pbs_stats.node += s.node - old_stats.node;
44        pbs_stats.depth += s.depth - old_stats.depth;
45        pbs_stats.restart += s.restart - old_stats.restart;
46        pbs_stats.nogood += s.nogood - old_stats.nogood;
47
48        // Update old_stats to current s
49        setOldStats(s);
50      }
51
52      /// Reset status for engine and PBS when PBS:next()
53      /// is called again
54      void resetstatus(void) {
55        e_stopped = false;
56        pbs_stopped = false;
57      }
58
59      // Functions must be called in the following order:
60      // setOldStats(engine.statistics())
61      // engine.next()
62      // update(engine.statistics())
63      /// Check if engine or PBS should be stopped.
64      bool stop(const Search::Statistics& s, const
          Search::Options& o) {
65        // PBS is checked first because of higher precedence
66        if((pbs_stop != NULL)) {
67          // Update pbs_stats
68          update(s);
69
70          // This will always see so that pbs_stopped
71          // has the right value
72          pbs_stopped = pbs_stop->stop(pbs_stats,o);
73
74          // Stop if the stop object for the PBS says so
75          if(pbs_stopped) {
76            return true;
```

```
77              }
78            }
79
80            // This will always see so that e_stopped
81            // has the right value
82            e_stopped = e_nodestop->stop(s,o);
83
84            // Stop if the stop object for the engine says so
85            if(e_stopped) {
86              return true;
87            }
88            return false;
89          }
90
91          /// Return engine stop object pointer to be able to
                update/manipulate it
92          Search::NodeStop* enginestop(void) {
93            return e_nodestop;
94          }
95
96          /// Check if engine stop object is stopped
97          bool enginestopped(void) {
98            return e_stopped;
99          }
100
101         /// Check if PBS stop object is stopped
102         bool pbsstopped(void) {
103           return pbs_stopped;
104         }
105
106         ~StopWrapper(void) {
107             delete e_nodestop;
108             delete pbs_stop;
109         }
110     };
111
112   protected:
113     // Engines
114     std::vector<EngineBase<T>*>& engines;
115
116     // Stop object
117     StopWrapper* so;
118
119     // Constants
120     int BLOCK;
121     int NR_ENGINES;
122     // int BLOCK = so->enginestop()->limit();
123     // int NR_ENGINES = engines.size();
124
```

```
125   public:
126     // Create portfolio of engines e1 and e2
127     PBS(std::vector<EngineBase<T>*>& _engines, Search::Stop*
          stop,
128         const Search::Options&
129         o=Search::Options::def)
130       : engines(_engines),
131         so(new StopWrapper(new Search::NodeStop(10), stop)) {
132
133       BLOCK = so->enginestop()->limit();
134       NR_ENGINES = engines.size();
135         // Provide all engines with modded stop object that
              wraps the provided stop object.
136       for(int i = 0; i < NR_ENGINES; i++){
137         engines[i]->stop(so);
138       }
139     }
140
141     /// Return next solution (NULL, if none exists or search
          has been stopped)
142     virtual T* next(void) {
143       T* returned = NULL;
144       so->resetstatus();
145       bool all_finished = false;
146       // bool run = true;
147
148       while(true){
149
150         for(int i = 0; i < NR_ENGINES; i++){
151           so->setOldStats(engines[i]->statistics());
152           returned = dynamic_cast<T*>(engines[i]->next());
153           so->update(engines[i]->statistics());
154
155           // std::cout << "\nengine:" << (so->enginestopped()
                ? "true" : "false");
156           // std::cout << " | PBS:" << (so->pbsstopped() ?
                "true" : "false");
157           // std::cout << "\ne" << i << " stopped:" <<
                (engines[i]->stopped() ? "true" : "false");
158
159           if(so->pbsstopped()){
160             // std::cout << "\nPBS stop object stopped!";
161             return returned;
162             // run = false;
163             // break;
164           }
165
166           if(returned != NULL){
167             // std::cout << " : ";
```

```
168             // returned->print(std::cout);
169             return returned;
170             // run = false;
171             // break;
172           }
173 
174         }
175 
176         // if(!run){
177           // break;
178         // }
179 
180         for(int i = 0, count = 0; i < NR_ENGINES; i++){
181           if(!engines[i]->stopped()){
182             count++;
183             // std::cout << "\ne" << i << " is Done!";
184           }
185           if(count == NR_ENGINES){
186             all_finished = true;
187           }
188         }
189 
190         if(all_finished){
191           // run = false;
192           break;
193         }
194 
195         // std::cout << "\nLimit: " <<
              so->enginestop()->limit() << " increased to: "
196                   // << so->enginestop()->limit()+BLOCK <<
                        "\n";
197         so->enginestop()->limit(so->enginestop()
198                                   ->limit()+BLOCK);
199       }
200 
201       return returned;
202     }
203 
204     /// Return statistics
205     virtual Search::Statistics statistics(void) const {
206       Search::Statistics stats;
207       for(int i = 0; i < NR_ENGINES; i++){
208         stats += engines[i]->statistics();
209       }
210       return stats;
211     }
212 
213     /// Check whether engine has been stopped
214     virtual bool stopped(void) const {
```

```
215         return so->pbsstopped();
216       }
217
218       /// Reset engine to restart at space \a s (do nothing for
            now)
219       virtual void reset(Space* s)  {}
220
221       /// Return no-goods (for the time being nothing)
222       virtual NoGoods& nogoods(void) { return NoGoods::eng; }
223
224       /// Destructor
225       virtual ~PBS(void) {
226       }
227     };
228
229   }
```

## B.2 Source Code for Parallel Portfolio Search

```
1   #include <gecode/search.hh>
2   #include <gecode/search/support.hh>
3
4   namespace Gecode {
5
6     template<class T>
7     class PBS {
8     protected:
9
10      class ModdedStopObject : public Search::Stop {
11
12        protected:
13          Search::Stop* stop_object;
14          Support::Mutex* command_mutex;
15          int* PBS_Command;
16          bool stopped;
17          bool* PBS_stopped;
18
19        public:
20          ModdedStopObject(Search::Stop* _stop_object,
                Support::Mutex* _command_mutex, int* _PBS_Command,
                bool* _PBS_stopped)
21          : stop_object(_stop_object),
                command_mutex(_command_mutex),
                PBS_Command(_PBS_Command),
                PBS_stopped(_PBS_stopped), stopped(false) {
22            // nothing more to do
23          }
```

```
24
25          bool stop(const Search::Statistics& s, const
               Search::Options& o) {
26
27             command_mutex->acquire();
28             /* 1 == stop
29                0 == go on */
30             if(*PBS_Command == 1) {
31                // std::cout << "PBS_Command: " << *PBS_Command
                     << "\n";
32                command_mutex->release();
33                return true;
34             }
35
36             if(stop_object != NULL) {
37                stopped = stop_object->stop(s,o);
38                if(stopped) {
39                   // std::cout << "PBS stop object stopped!\n";
40                   (*PBS_stopped) = true;
41                   command_mutex->release();
42                   return true;
43                }
44             }
45             command_mutex->release();
46             return false;
47          }
48
49          ~ModdedStopObject(void) {
50                // delete e_nodestop;
51                delete stop_object;
52          }
53       };
54
55       // FIFO queue for PBS solutions
56       template<class U>
57       class PBS_queue {
58
59          struct Node {
60             U* solution;
61             Node* next;
62          };
63
64          public:
65          Node* first;
66          Node* last;
67
68          // constructor
69          PBS_queue() {
70             first = NULL;
```

```
71          last = NULL;
72        }
73
74        void push(U* result) {
75          // Copy/clone solution into new object to store in to
                move
76          // the entire object from thread stack onto master
                stack
77          if(first == NULL) {
78              first = new Node;
79              first->solution =
                    dynamic_cast<U*>(result->clone(false)); //
                    Does it need to be cloned?
80              first->next = NULL;
81              last = first;
82          }
83          else {
84            last->next = new Node;
85            last = last->next;
86            last->solution =
                  dynamic_cast<U*>(result->clone(false)); // Does
                  it need to be cloned?
87            last->next = NULL;
88          }
89        }
90
91        U* pop() {
92          // return first solution
93          U* returned;
94          if(first != NULL) {
95            Node* temp = first;
96            first = first->next;
97            returned = temp->solution;
98            delete temp;
99          }
100         else {
101           returned = NULL;
102         }
103         return returned;
104       }
105     };
106
107     template<class V>
108     class run_wrapper : public Support::Runnable {
109
110       public:
111       EngineBase<V>& e;
112       Support::Mutex* solution_mutex;
113       Support::Event* event;
```

```
114        PBS_queue<V>* queue;
115        int* finished_threads;
116        int* stopped_threads;
117
118        run_wrapper(EngineBase<V>& _e, Support::Mutex*
               _solution_mutex,
119          Support::Event* _event,
120          PBS_queue<V>* _queue,
121          int* _finished_threads,
122          int* _stopped_threads)
123          : e(_e),
124          solution_mutex(_solution_mutex),
125          event(_event),
126          queue(_queue),
127          finished_threads(_finished_threads),
128          stopped_threads(_stopped_threads) {}
129
130        virtual void run(void) {
131          V* result = e.next();
132
133          solution_mutex->acquire();
134          if(result != NULL) {
135            queue->push(result);
136          }
137          // std::cout << "Stopped: " << e.stopped() << "\n";
138          if(!e.stopped()) {
139            (*finished_threads)++;
140          }
141          else {
142            (*stopped_threads)++;
143          }
144          solution_mutex->release();
145          event->signal();
146        }
147      };
148
149      // Mutexes and event to protect critical areas.
150      Support::Event* event;
151      Support::Mutex* command_mutex;
152      Support::Mutex* solution_mutex;
153
154      // Shared variables for communication.
155      int NR_THREADS;
156      int finished_threads;
157      int stopped_threads;
158      int PBS_Command;
159      bool PBS_stopped;
160
161      // Queue
```

```
162     PBS_queue<T>* queue;
163
164     // Engine and run_wrapper vectors and stop object
165     std::vector<EngineBase<T>*>& engines;
166     std::vector<run_wrapper<T>*> runners;
167     Search::Stop* stop;
168
169   public:
170     // Create portfolio of engines e1 and e2
171     PBS(std::vector<EngineBase<T>*>& _engines, Search::Stop*
          _stop,
172         const Search::Options&
173         o=Search::Options::def)
174         : engines(_engines), stop(_stop) {
175
176       // Create mutexes, event, queue and set shared
            variables.
177       command_mutex = new Support::Mutex();
178       solution_mutex = new Support::Mutex();
179       event = new Support::Event();
180       queue = new PBS_queue<T>();
181       NR_THREADS = engines.size();
182       finished_threads = 0;
183       PBS_Command = 0;
184       stopped_threads = 0;
185       PBS_stopped = false;
186
187       // Provide all engines with modded stop object that
            wraps the provided stop object.
188       for(int i = 0; i < NR_THREADS; i++){
189         engines[i]->stop(new ModdedStopObject(stop,
              command_mutex, &PBS_Command, &PBS_stopped));
190       }
191     }
192
193     /// Return next solution (NULL, if none exists or search
          has been stopped)
194     virtual T* next(void) {
195       T* returned = NULL;
196
197       // Check if there exist already found solutions in
            queue.
198       // If yes, return said solution.
199       solution_mutex->acquire();
200       returned = queue->pop();
201       solution_mutex->release();
202       if(returned != NULL) {
203         // std::cout << "Master: Popped existing solution!\n";
204         // returned->print(std::cout);
```

```
205         return returned;
206       }
207
208       // Set PBS stopped status to false.
209       command_mutex->acquire();
210       PBS_stopped = false;
211       command_mutex->release();
212
213       // Create run_wrappers for every engine and start them
              in separate threads.
214       for(int i = 0; i < NR_THREADS; i++){
215         Gecode::Support::Thread::run(new
                run_wrapper<T>(*engines[i], solution_mutex, event,
                queue, &finished_threads, &stopped_threads));
216       }
217
218       // Master thread will "eat up" possible residue signals
              on the event by just going some extra laps in the
              do-while
219       while(true) {
220         event->wait();
221         solution_mutex->acquire();
222         returned = queue->pop();
223         // std::cout << "Master: finished_threads: " <<
                finished_threads << "\n";
224
225         if((finished_threads + stopped_threads) ==
              NR_THREADS) {
226           // Reset values and exit
227           finished_threads = 0;
228           stopped_threads = 0;
229           solution_mutex->release();
230           break;
231         }
232         else if(returned != NULL) {
233           command_mutex->acquire();
234           PBS_Command = 1;
235           command_mutex->release();
236           solution_mutex->release();
237           while(true) {
238             solution_mutex->acquire();
239             // std::cout << "stopped_threads: " <<
                    stopped_threads << "\n";
240             if((finished_threads + stopped_threads) ==
                  NR_THREADS) {
241               // Reset values and exit
242               stopped_threads = 0;
243               command_mutex->acquire();
244               PBS_Command = 0;
```

```
245             command_mutex->release();
246             finished_threads = 0;
247             solution_mutex->release();
248             break;
249           }
250           solution_mutex->release();
251           Gecode::Support::Thread::sleep(10);
252         }
253         break;
254       }
255       else {
256         solution_mutex->release();
257       }
258     }
259 
260     if(returned != NULL) {
261       // returned->print(std::cout);
262     }
263     else {
264       if(PBS_stopped) {
265         // std::cout << "PBS stopped\n";
266       }
267       else {
268         // std::cout << "Master: Solution is NULL...\n";
269       }
270     }
271 
272     return returned;
273   }
274 
275   /// Return statistics
276   virtual Search::Statistics statistics(void) const {
277     Search::Statistics stats;
278     for(int i = 0; i < NR_THREADS; i++){
279       stats += engines[i]->statistics();
280     }
281     return stats;
282   }
283 
284   /// Check whether engine has been stopped
285   virtual bool stopped(void) const {
286     return PBS_stopped;
287   }
288 
289   /// Reset engine to restart at space \a s (do nothing for
        now)
290   virtual void reset(Space* s) {}
291 
292   /// Return no-goods (for the time being nothing)
```

```
293        virtual NoGoods& nogoods(void) { return NoGoods::eng; }
294
295        /// Destructor
296        virtual ~PBS(void) {
297        }
298    };
299
300 }
```

# Bibliography


[1] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006. Referenced on chapter 1.

[2] Bernardo A Huberman, Rajan M Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[3] Investopedia. Portfolio. Retrieved on April 17th 2015. URL: `http://www.investopedia.com/terms/p/portfolio.asp`.

[4] Financial-Dictionary. Definition of portfolio. `http://financial-dictionary.thefreedictionary.com/portfolio`. Retrieved on April 17th 2015.

[5] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. A portfolio approach to algorithm selection. In *IJCAI*, volume 1543, page 2003, 2003.

[6] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Chapter 1: Introduction. In Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2010. Referenced on page 1. Corresponds to Gecode 4.3.3.

[7] Carla P Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1):43–62, 2001.

[8] IEEE. Code of ethics. Retrieved on April 8th 2015. URL: `http://www.ieee.org/about/corporate/governance/p7-8.html`.

[9] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006. URL: `http://www.gecode.org/~schulte/paper.html?id=SchulteCarlsson:CPH:2006`.

[10] Christian Schulte and Guido Tack. Programming propagators. In Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2015. Corresponds to Gecode 4.3.3.

[11] Peter Van Beek. Backtracking search algorithms. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 4, pages 85–134. Elsevier BV, Amsterdam, 2006.

[12] Carla P Gomes and Bart Selman. Algorithm portfolio design: Theory vs. practice. In *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, pages 190–197. Morgan Kaufmann Publishers Inc., 1997.

[13] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling. In Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2015. Corresponds to Gecode 4.3.3.

[14] Christian Schulte. Programming search engines. In Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2015. Corresponds to Gecode 4.3.3.

[15] EJ Hoffman, JC Loessi, and RC Moore. Constructions for the solution of the m queens problem. *Mathematics Magazine*, pages 66–72, 1969.

[16] Igor Rivin, Ilan Vardi, and Paul Zimmerman. The n-queens problem. *American Mathematical Monthly*, pages 629–639, 1994.

[17] Carla P Gomes, Bart Selman, et al. Problem structure in the presence of perturbations. *AAAI/IAAI*, 97:221–226, 1997.

[18] Charles J Colbourn. Embedding partial steiner triple systems is np-complete. *Journal of Combinatorial Theory, Series A*, 35(1):100–105, 1983.

[19] Gecode Documentation. Gecode::support::thread class reference. Retrieved on April 15th 2015. URL: `http://www.gecode.org/doc-latest/reference/classGecode_1_1Support_1_1Thread.html`.