Parallel and Distributed Search in Constraint Programming

Carl Christian Rolf
d00cr

June 9, 2006

Abstract

Constraint programming is a growing field of research and is increasingly interesting to the software industry. It is considered the leading method of solving discrete optimization problems, such as scheduling. The biggest issue with optimization problems is that they often have an exponential time complexity. Both parallelization and distribution can be used to make more problems solvable in acceptable time.

In this thesis parallel and distributed extensions to a constraint solver have been implemented and evaluated. We have used a solver written entirely in Java, which facilitated fast development. The main goal of this work was to explore whether such extensions are feasible with regard to the timeframe and if they are useful from a performance perspective.

The hardware development, at time of writing, has lately taken a turn towards processors with several cores. In order to harness the power of the next generation hardware, parallelization of constraint solvers are necessary. Further, the internet and improved network technologies provide a means of achieving high performance at low cost.

The issues that need addressing when parallelizing or distributing a program are the cost of communication between processes and load balancing. This thesis focuses on the cost of copying and serialization of objects in a modern language with automatic memory-management.

Contents

Chapter 1

Introduction and Background

The goal of this master thesis is to evaluate the performance gain that can be achieved through parallelization and distribution of a constraint solver. Previous work in this field has mostly dealt with solvers written in C or C++, where memory is managed manually by the programmer. *Disolver* (Disolver, 2006) is an example of such a program. In this thesis, however, a solver written entirely in Java, a language with automatic memory-management, will be used which affects the aim of increasing the performance of the solver.

1.1 Constraint Programming

1.1.1 History

Constraint programming first started to appear in the late 1960's (Marriot & Stuckey, 1998). In the beginning it mostly consisted of extensions to traditional programming languages that allowed the programmer to use constraint relations. The extensions, however, were not appealing enough since they relied on fairly simple methods that could rarely be used outside specific contexts. Later came languages such as *Prolog*, which used a declarative syntax that allowed the programmer to let a powerful solver take care of the complex operations.

1.1.2 Example

A good example of constraint programming is the *send more money*-problem, where values are assigned to the letters in a way that solves the equation.

```
          S   E   N   D
+         M   O   R   E
=   M     O   N   E   Y
```

This problem can be written in Prolog as depicted in Figure 1.

```
1       send_more_money(S, E, N, D, M, O, R, Y) :-
2               [S, E, N, D, M, O, R, Y] :: [0..9],
3               set_constraints([S, E, N, D, M, O, R, Y]),
4               labeling([S, E, N, D, M, O, R, Y]).
5
6       set_constraints([S, E, N, D, M, O, R, Y]) :-
7               S =\= 0,
8               M =\= 0,
9               alldifferent([S, E, N, D, M, O, R, Y]),
10              1000 *S + 100 *E + 10 *N + D    +
11              1000 *M + 100 *O + 10 *R + E    =
12              10000*M + 1000*O + 100*N + 10*E + Y.
```

Figure 1: Prolog version of the send more money problem.

In the program in Figure 1 the capital letters are variables that can assume any numeric value. On line 2 the variables are constrained to only hold a value between 0 and 9. On line 7 and 8 the first variable in the numbers that the words spell are set to not be equal to zero. On line 9 it is declared that all the variables must have different values. On the last three lines the relationships between the values are defined. The actual search for a valid solution is called on line 4. During the labeling process all the constraints are checked for consistency, given the values that the solver has assigned to the variables that have already been processed.

1.1.3 Constraint Solver

In this thesis we will assume that only integer values are allowed for variables. This limitation is not as strong as it may seem – the most important usages of constraint programming are discrete optimization problems. In constraint programming, integer-only variables are referred to as *finite domain variables* (FDVs). All such variables have a *finite domain* (FD) of values that the variable is allowed to assume.

The *constraint solver* finds solutions to the problem that has been declared. This is done through a depth-first search. On each level of the search a variable is chosen in some way and assigned one of the values from its domain. Then, all the constraints that the assignment can affect are evaluated to see if the assignment breaks any of the rules that the programmer has given, this is referred to as *consistency checking*. During this check *pruning* is performed, where illegal values are removed from the domains of the variables whose constraints were affected by the assignment. Because of pruning, the number of values that need to be tested per variable is reduced the deeper the search goes. If the assignment does not violate any constraints it is said to be *consistent* and the search can progress. If it is *inconsistent* then the solver has to *backtrack* by removing the assigned value from the domain of the variable and try another value.

1.1.4 Consistency Methods

There are several different ways of performing consistency checking: *node consistency*, *arc consistency*, *generalized arc consistency*, *path consistency*, and *bounds consistency* (Dechter, 2003). Generalized arc consistency and path consistency are rarely used since there are more efficient, often constraint specific, algorithms.

Node and arc consistency are named so because of their operation in a constraint graph, where a FDV is a node and constraints are arcs between nodes. Node consistency is the method used for *unary constraint*, i.e., constraints with only one FDV where the arc ends in the same node it originated from. In Figure 1 the constraints on line 7 and 8 are typical examples of unary constraints. Node consistency determines if a value is consistent or not, if it is inconsistent, then that value is removed from the domain of the variable.

Arc consistency is used for *binary constraints* – constraints with two FDVs. Given the FDVs X and Y the constraint graph is consistent if for every possible value of X there is a value of Y that satisfies the constraint between X and Y.

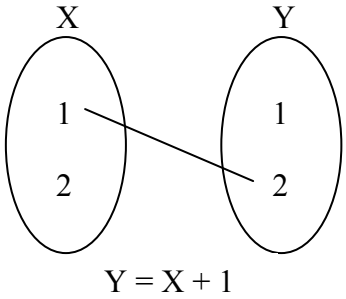


Figure 2: Arc-consistency.

As can be seen in Figure 2, if X and Y belong to the domain {1, 2}, only X = 1 and Y = 2 is a legal assignment given the constraint Y = X + 1. This will be discovered by the arc consistency algorithm, and the domains will be pruned to match.

Bounds consistency looks at the interval of values in the domains rather than the actual values in those domains. This is done for an arbitrary number of variables in a constraint. Bounds consistency is mostly performed on arithmetic constraints. By grouping several variables into the same constraint, bounds consistency becomes more efficient than using the equivalent set of binary constraints and solve those using arc consistency. In bounds consistency, the domains are approximated by the minimum and the maximum value. By finding two solutions, one minimal and one maximal, the remaining values in the domain will also be possible solutions. Since there can be holes in domains, a domain may not be describable simply by the lowest and highest value of the domain. Bounds consistency can be extended to use *domain consistency* that looks at the subintervals instead.

1.1.5 Search Methods

There are three different kinds of search: search for **one** solution, search for **all** solutions, and search for the **optimal** solution. Many problems in constraint programming are NP-complete, and thereby proving the optimality is NP-hard (Garey & Johnson, 1979). Finding a solution, however, usually takes exponential time. Since there is no way to know whether a particular branch of the search tree will be a solution or not, the progress through a branch has to be stored in order to allow backtracking. This means that a constraint solver will use a large amount of memory during the search process.

Search for one solution is just like an ordinary depth-first search. After a solution has been found all the variables that were to be labeled have been assigned a value, and the search terminates.

Search for all solutions will try all consistent combinations of values. In essence it will perform a large amount of searches for one solution.

Search for the optimal solution is done by using a *cost function* that describes the cost of the solution. The optimal solution is that which minimizes (or maximizes) the cost function. The most common method for searching for the optimal solution is *branch and bound* (BB) (Land & Doig, 1960; Lawler & Wood, 1966). The principle of BB is that after a solution has been found, there is no need to explore branches that can only render more expensive solutions, i.e., the search is bounded. Therefore the search for the optimal solution will usually be much faster than the search for all solutions.

Sometimes it is enough to find any solution at all, and if the problem is difficult to solve, one can use methods of *heuristic search methods*. Such methods are *incomplete* in the sense that there may be solutions even if a local search does not find it. One form of heuristic search is *credit search*, which spends a certain amount of its allotted credits in one part of the tree,

before backtracking to the start in order to proceed down into other branches using the remainder of the credits. These methods are especially good if one either wants a solution fast or not at all.

1.1.6 Backtracking

When the current solution is discovered to be inconsistent, the search has to backtrack. This can be a very costly procedure if the search gets stuck in one part of the tree. One reason for the search to get stuck is that it does not backtrack to the level that is the real cause of the inconsistency. Instead it backtracks only to the point where the inconsistency is essentially doomed to happen again. Getting stuck at processing the same part of the tree over and over again is referred to as *thrashing*.

In order to avoid thrashing one can use *look-ahead* or *look-back* techniques. Look-ahead refers to the process of trying to avoid situations where the solver can get stuck. Such methods require some meta-knowledge of the problem, however. Therefore, in practice, look-ahead is mostly done by using good methods for variable selection. Look-back on the other hand is less problem specific since it takes place later in the search-tree, making learning a possibility.

The first step in look-back is to find the cause of the inconsistency, after which one can either backtrack to the assignment that caused the problem, or backtrack and record the reason. Backtracking several steps is referred to as *backjumping*, since it jumps to the cause of the inconsistency. Finding the conflicting variable and performing the jump is an expensive task, therefore recording the reason in the form of a constraint is a good idea. Otherwise the performance gained, may be lost in other parts of the tree. Unfortunately learning is also expensive, especially in space. Despite the costs involved, however, look-back can significantly improve the performance of the search (Dechter, 2003).

1.1.7 Variable Selection

The depth first search can be made a lot faster by labeling variables in an intelligent order. The most common method is called *first fail*, it chooses the FDV with the smallest domain. This will quickly reduce the width of the search tree. Another method is *most constrained*, which picks the variable used in the most constraints. This will maximize the propagation and therefore improve pruning.

1.1.8 Value Selection

After a variable has been chosen a value has to be assigned to it. This value is taken from the domain of the FDV. If one is searching for the minimal solution then selecting the smallest value in the FD is the logical approach. Otherwise, one can select the medium value or any other value in the domain that is suitable for the solution one is looking for.

1.2 Parallel Search

Parallel search is the multithreaded version of search. By using several threads one can run search processes on several processors concurrently. In this thesis, *parallel* refers to the running of several processes concurrently where the processes share memory, and *distributed* refers to the situation where the processes do **not** share memory, i.e., they run on separate machines. The situation where one runs several processes on several machines, i.e., distributed-parallel processing, will be mentioned but not studied.

Parallel search has been an important research area for quite some time, Grama & Kumar (1995) sums up a lot of that research. The simple reason is that one can achieve a great speed-up through the usage of several processors. The basic principle is to create different processes for different parts of the search tree. The problem with such an approach is that some subtrees may be a lot smaller than others. In constraint programming, where the sizes of the domains change as constraints are enforced, there is no way to tell how big a subtree will be. Therefore, one has to implement *load balancing*, i.e., when a process that has finished can receive more work from those processes that are overburdened. There are several different approaches to achieve load balancing (Mitra et al., 1997). The reason there are so many methods is that the goal is to minimize communication while maximizing the processor usage. Obviously there is no solution that is optimal in all cases, rather the performance is dependent on the hardware. If the cost of communication is too high compared to the execution time, it may even be better not to parallelize the search at all.

In both parallel and distributed search there are three stages: initiation, execution, and end. In the initiation stage, the environment and search processes are started. During the execution, the load balancing tries to maximize the amount of useful work performed by the processors or machines. Finally, at the end stage, the search is almost finished and therefore the cost of communication is too high, and the processes begin to terminate.

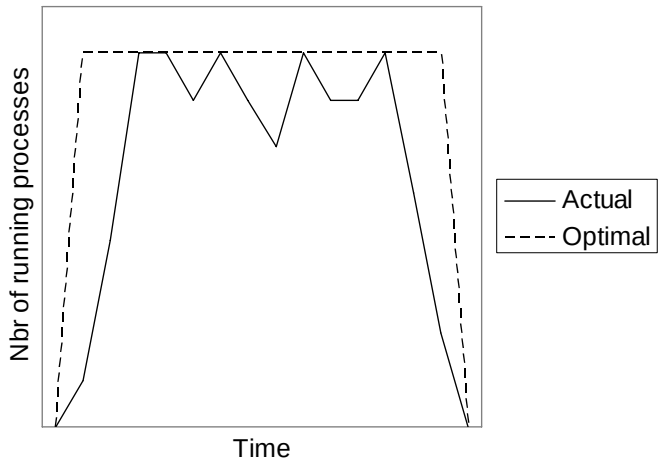


Figure 3: Load balancing

Figure 3 illustrates the actual load versus the optimal load. Ideally, all the available processes should execute simultaneously for the entire duration of the search. But instead, the initialization and end stages limit how long the processes are available to the search. Also, the load balancing is unable to keep the load at its maximum for the duration of the execution stage.

The efficiency measurement of a parallel program is called speed-up. There are four categories of speed-up: *super-linear*, *linear*, *sub-linear*, and *slow-down* (Mitra et al., 1997). The speed-up is calculated as the time it takes for a single process to finish, divided by the time it took for several parallel processes to finish: $S = T_{serial} / T_{parallel}$. If the measure is higher than the number of processes, the speed-up is super-linear. If it is lower, it is sub-linear, and if it is equal to the number of processes, the speed-up is linear. In the case of slow-down, the speed-up starts out as at least sub-linear and then shrinks as more processes are added.

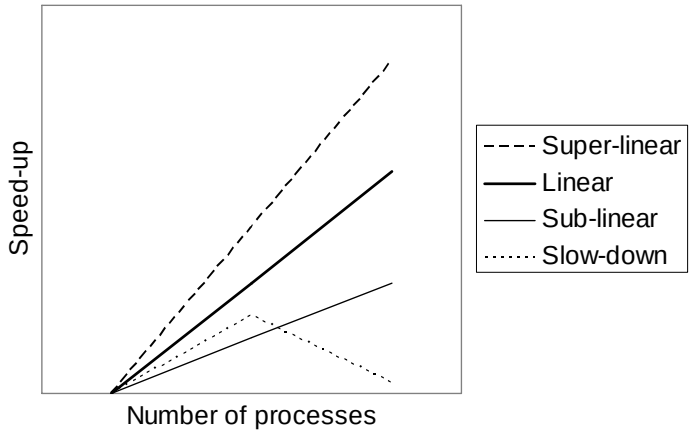


Figure 4: The different speed-ups

Amdahl's law (Amdahl, 1967) states that since the parallel program will contain non-parallel parts, the speed-up cannot be higher than sub-linear. In the case of branch and bound, however, this does not hold since strong bounds, discovered early can lead to a super-linear speed-up, see for instance Ashford et al. (1992). One effect that is almost always observed, though, is that the improvement in speed-up is reduced the more processes are added. The reason is that the more processors or machines there are, the more communication will take place in order to maximize the utilization. Since the communication is inherently serial, the non-parallel portion of the execution as a whole will increase.

1.3 Distributed Search

Distributed search works in the same way as parallel search with the difference that the processes run on separate machines. By *separate machines* one usually means machines that do not share memory. Hence, the communication between the processes has to take place over some dedicated communications channel, such as a network. Thanks to the development of the Internet, network technology has made great advances in the past decades, and is today superior both in cost and bandwidth to competing technologies. In this thesis it will be assume that the communication is done over some form of network.

In distribution, the minimizing of communication is more important than in parallelization. There are several reasons, one is that the bandwidth in a network is much lower than for a system-bus. Therefore, the messages sent, should be of minimal size. Another concern is that the information needs to be *serialized* in some way. This means that the contents of the messages have to be translated into a sendable form, and then translated back by the receiver. Hence, an effective translation is necessary, and a further requirement for small messages is added. The third issue is related to the structure of the network, the most common form of network today is a local area network (LAN) that has a tree structure. Just as for *hypercubes*, which have a direct connection in cubic form (Dehne et al., 1990), some computers are faster to reach than others. The biggest downside of using a LAN is that the bandwidth is dependent on both the throughput and the number of messages sent. Only one computer at a time can send a message to a specific receiver, all other computers have to wait before they can send their messages (Forouzan, 2001). There have been several attempts at minimizing communication, one successful method is to use a hierarchy, where a master controls the load balancing (Aida & Natsume, 2003).

Since networks have previously been quite slow, there has not been that much research into using distribution over a LAN as a means of improving the speed of search. The focus has instead been at creating algorithms for distributed variables, a situation called *distributed constraint satisfaction problem* (DisCSP). The most effective way to speed up the search process is to let all the computers access all the data, and to make the communication as efficient as possible. In the DisCSP situation, the goal is not to maximize speed, but rather to solve the problem without sharing data between the computers. In this thesis we will only briefly deal with distributed variables, see Yokoo et al. (1998), Yokoo & Hirayama (2000), and Hirayama & Yokoo (2004) for more on DisCSP. The advantage of not sharing information between processes is information security. By mostly sending *ok* and *not-ok* messages, little information would leak if someone listened in on the communications. Another method to improve security is to encrypt the information (Yokoo, 2005). Lately there has been some research into the combined situation where constraints are distributed, and several search processes are used in order to maximize the utilization of the available computers. This has been done to great success with regard to speed-up in the DisCSP case (Zivan & Meisels, 2006).

The latest development in distributed computing is *grid* technologies (Baker et al., 2002). Grid computing is a term describing the techniques involved in facilitating distribution over any given network. The technology involved is, e.g., standardized communication protocols. The idea is to utilize computer power that has been made available by its owners. To date, the most famous example of a grid-like effort is the SETI@home project (SETI, 2006).

1.4 Parallelization and Distribution in Java

Java has built-in support for both parallelization and distribution. These facilities can be somewhat cumbersome to use, however, therefore there are also several helper libraries. One such example is *cajo* (cajo, 2006), which provides a simpler means of calling functions on a different machine. Both synchronous and asynchronous calls can be made, both with the drawback that one has to use the *Object* class for return values and parameters. Although cajo was not used in this work, the means of using multicast messages to make the presence of an RMI-server known was inspired by it. The reason cajo was not used is that the design of the distribution made it very simple to use the standard RMI library in Java. In Java there is also support for *CORBA* (CORBA, 2006), which allows one to distribute programs between different programming languages. Since all the code was written in Java, using CORBA would only have created a further obstacle in the development

Parallelization in Java is quite simple to achieve, there are built-in thread objects and using the start method, those threads will be run separately from the main thread. Locks and synchronization are also features in the standard libraries and in the language itself. Therefore there is rarely any need to use third party libraries to achieve parallelization.

1.5 Motivation

There are two main reasons why this thesis is relevant, the first in that the solver that has been used was written in a *managed* language, i.e., the allocation and deallocation of memory is handled automatically, and the second reason is the current development in hardware towards *multi-core processors*.

The fact that the solver is written in Java instead of C or C++ has an impact for both parallelization and distribution. The Java runtime requires quite a lot of memory, and all objects have a much larger overhead than in C++. Given that some problems in constraint programming use several hundred constraints and variables, the overhead will have a significant impact on the time it takes to initialize processes. Another factor that needs to be studied in parallel search is the impact of the runtime. The garbage collector in Java 1.5 is capable of running as a separate process. Therefore a system with more than one processor core will automatically receive a performance boost. The level of performance increase from the parallel garbage collecting is dependent on how much memory the search uses.

In contrast to C and C++, Java has a built-in support for concurrency. This facilitates the rapid development of a parallel solver, allowing both parallelization and distribution to be studied in the timeframe of a master thesis. Java also has libraries supporting *remote method invocation* (RMI). The RMI classes provide a simple means of creating distribution, where one can handle a call to a function on another machine just like a local function call. The downside is that the communication is handled through automatic serialization, during which some of the overhead of the parameter objects will also be serialized. Hence, the serialization and communication needed to make a remote invocation takes at least an order of a magnitude longer than the copying of objects needed for a parallel call.

The second of the two main motives for this thesis, the current hardware development, is relevant since both parallelization and distribution is being studied. In 2005 the first *dual-core* processors were made available to the average desktop consumer. Multi-core technology is the term for putting several processor cores on the same *central processing unit* (CPU). One of the main reasons for this development is that the heat generated from energy loss became unmanageable. In a move to improve performance without increasing heat loss, Intel and AMD – the two biggest CPU manufacturers in the world at the time of writing – decided to put several cores on the same processor. In order to harness the power of several cores, however, programs must use several threads, running in parallel. Therefore the research and development of good techniques and tools for parallelization will become central to the development of end-user software.

Since a multi-core processors system will only need one communications channel to the memory and only one processor socket, it is likely to be cheaper to manufacture than an equivalent multi-processor system. Combined with the advent of grid computing, the future of large-scale servers and supercomputers probably lies in grids of multi-core systems. The amount of communication needed between computers in such machine constellations would be inversely dependent on the number of cores per processors, improving the case for existent network technologies such as LANs in supercomputing.

One of the main arguments against parallel and distributed search is that the problems are usually NP-hard, and finding a solution often takes exponential time (Roucairol, 1996; Grama & Kumar, 1995). Thus, one would need an exponential amount of processors in order to reduce the problem to polynomial time. However, processor development has followed Moore's law (Moore, 1965) quite well since its inception. Hence, the potential performance of a processor, as a function of time, is exponential. Given the development towards multi-core processors, it is therefore necessary to move to parallel search methods, or compile for parallelism (Hermenegildo, 2000), in order to continue to take advantage of the technological development.

Chapter 2

Managed Constraint Solver

2.1 Basics

The solver that this thesis uses is *JaCoP* (**Ja**va **Co**nstraint **P**rogramming Library), it is written entirely in Java 1.5 (JaCoP, 2006). In order to use the solver, one writes a regular Java program that uses the constraints that the JaCoP library provides. The most common constraints, such as $X \neq Y$, *alldifferent*, and *cumulative* are all available.

The search procedure in the solver will first check that the all the constraints are consistent, i.e., a solution is possible, then perform the search, and last it will assign specific values to the variables that were to be named. The algorithm for the search looks as follows:

```
boolean label(v)
        if empty(v)
                if consistent(store)
                        return true
                else
                        return false
        else
                if not consistent(store)
                        return false

                choose(v1 from v)
                choose(n1 from domain(v1))
                assign(n1 to v1)

                if consistent(store)
                        remove(v1 from v)
                else
                        remove(n1 from domain(v1))
                return label(v)
```

Figure 5: Basic JaCoP search algorithm for one solution.

Figure 5 shows the algorithm for finding one solution, the only change needed to find all solutions is to make the algorithm always return `false`, thereby all branches of the search tree will be explored. The algorithm also presumes that the assignments leading to a solution are stored somewhere, in JaCoP the solution is built as the recursion returns its way back through the tree.

How the variables and values are chosen, is defined by the programmer, first fail and choosing the middle value is a good default. The methods used in consistency checking and pruning are automatic, and depend on the constraint in question. Some constraints, like alldifferent, can be implemented using bounds consistency, which allows for better pruning. This can greatly benefit the speed of the search (Marriot & Stuckey, 1998).

2.2 Example

The send more money-problem could be implemented using JaCoP as below.

```
1  import java.util.ArrayList;
2  import JaCoP.*;
3
4  class SendMoreMoney
5  {
6         public static void main(String[] args) {
7                 FDstore store = new FDstore();
8
9                 FDV S = new FDV(store, "S", 0, 10);
10                FDV E = new FDV(store, "E", 0, 10);
11                FDV N = new FDV(store, "N", 0, 10);
12                FDV D = new FDV(store, "D", 0, 10);
13                FDV M = new FDV(store, "M", 0, 10);
14                FDV O = new FDV(store, "O", 0, 10);
15                FDV R = new FDV(store, "R", 0, 10);
16                FDV Y = new FDV(store, "Y", 0, 10);
17                FDV[] vec = {S, E, N, D, M, O, R, Y};
18
19                store.impose(new XneqC(S, 0));
20                store.impose(new XneqC(M, 0));
21                store.impose(new Alldiff(vec));
22                FDV sum = new FDV(store, "Sum", 0, 300000);
23                store.impose(new SumWeight(new FDV[]{S, E, N, D, M, O, R, E},
24                                new int[]{1000, 100, 10, 1, 1000, 100, 10, 1}, sum));
25                store.impose(new SumWeight(new FDV[]{M, O, N, E, Y},
26                                new int[]{10000, 1000, 100, 10, 1}, sum));
27
28                Search label = new SearchOne();
29                ArrayList<FDV> fdvs = new ArrayList<FDV>();
30                for (FDV v : vec)
31                        fdvs.add(v);
32                boolean result = JaCoP.Solver.searchOne(store, fdvs, label,
33                                new IndomainMiddle(), new Delete(new FirstFail()));
34
35                if (result)
36                        System.out.println("Solution found. Result: " + fdvs);
37                else
38                        System.out.println("No solution found.");
39        }
40 }
```

Figure 6: JaCoP version of the send more money problem.

On line 7, the *store* is declared, it is used to hold all the variables in the program. The finite domain variables that are declared on line 9 through 16 are put in the store, with a name and two values defining the domain of the variable. The constraints are imposed using the store as a facilitator. Then, on line 28, the search-object is declared, which defines how the search is to be performed in the call to the solver on line 32. On line 33 the method for choosing values in the domain is declared. So is the method used in variable selection, in this case it is first fail. If the result from the call to *searchOne* is `true` then there is a solution, in the example this is always the case. The output of the program will be:

```
*** Number of decisions: 10
*** Search depth: 10
*** Backtracks: 1
Solution found. Result: [S=9, E=5, N=6, D=7, M=1, O=0, R=8, Y=2]
```

The first three lines describe the search progress. The number of decisions are the number of assignments made on the way to the solution, the search depth is the depth reached in the search for the solution, and the number of backtracks is the number of times the search had to try another value for a variable.

2.3 Implementation

The downside of using a managed programming language is that the memory usage tends to be higher. This has to do with overhead used for, e.g., synchronization and other built in features in the language. In the case of Java, the overhead has been somewhat reduced by the introduction of generics in Java 1.5. Still, the solver uses a lot of memory, and some optimizations such as array-copying have been made in order to reduce the memory requirements. Even so, the traditional N-Queens problem, where one is to place n queens on an $n*n$ chess-board requires more than 128 megabytes of memory for $n \geq 300$.

During the search the value selected for the variable is taken from the domain of the variable. If the assignment is inconsistent that value is removed from the domain by finding the complement of the domain and then intersecting it with the original domain. If the domain consists of many intervals, creating the complement can take quite a bit of memory, and finding the intersection can take significant time.

The most memory-demanding part of the solver is maintaining the history of the search, without such information, backtracking would not be possible. The information is handled through a depth-level in the store, and a level-stamp on the finite domains. If the stamp is lower than the level of the store, the new domain will backtrack to the current domain. That way only the domains that change need to be stored.

When running consistency checking, the constraints where the variables have changed are evaluated. For most constraints the consistency checking is done by making sure that the domain of the variable is in the right range. Any domain-values outside the valid range are pruned by calculating the intersection between the range and the domain.

The last step of the search – the actual assignment of values – consists of making sure that all variables are in a singleton-range, i.e., the domain of the variable only has one value. If there is such a situation, and it is consistent, a unique solution has been found.

In Appendix A, the first action sequence diagram describes the program for a small problem using the non-extended solver.

Chapter 3

Extensions for Parallelization and Distribution

In order to test the parallel and distributed performance of the solver, it had to be somewhat modified. Given that the search algorithm is recursive, however, the modifications were quite small. An important part consisted of deciding which level of the recursion to place a remote call, and how to divide the workload. In order to perform tests of different forms of search, we extended three versions of the search processes: search for one solution, search for all solutions, and branch and bound search for the optimal solution. These will henceforth be known respectively as *searchOne*, *searchAll*, and *searchBB*.

3.1 Splitting the Workload

As mentioned in the introductory part, one of the most important aspects of parallelization and distribution is load balancing. Although there are several advanced methods for load balancing, using a more advanced method makes little sense when one uses a LAN with fairly few computers, or a multiprocessor computer (Xu et al., 1995). Instead we made the choice to rely on limiting not which machines or processes to call, but rather when to allow such calls.

Since the solver is limited to finite domains of integer values, the most obvious method to parallelize a program is to simply split the domain of the chosen variable. One part of the split is kept for the current process to continue with, the other part is sent to another process. By deciding when to split the domain and how much to split off one can achieve an acceptable load balancing.

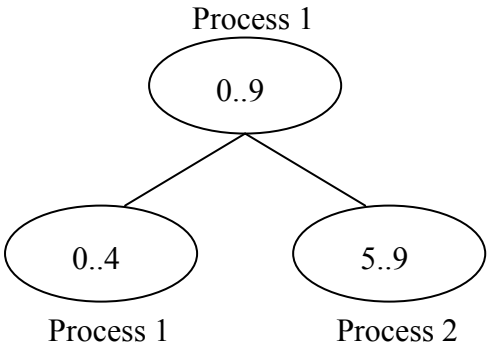


Figure 7: Domain splitting.

In Figure 7 the split has been made in the middle of the original domain. Therefore, if the search is a searchBB for the minimum solution, it is likely that Process 2 will finish executing before Process 1. One way to counter this effect would be to use an advanced method for assessing the work that needs to be done. However, during the course of this thesis we tested this with little success. The method that generated the overall fastest program was to simply look at the level of the store and allow splits to occur at about 85% of the maximum possible

depth. Although this may seem crude, using heavier mathematical computations often requires too much time.

Determining where to split was the other important factor in the load balancing. Tests showed that splitting off almost the entire domain was a good idea. This has to do with the limitation in the program that each search process can only split the domain of each variable once. By performing the split early in the domain, most of the work will go to the new search process, which can then split the domain further when calling other processes.

It is apparent that the optimal time and place of a split is dependent on the problem. If one is searching for a minimum value for instance, it makes little sense to let one process search a large portion of the lowest values in a domain. Other important factors include the size of the domain, the level of pruning that can be expected, and the total number of variables that need to be labeled. In this thesis, however, we used problem-independent values to decide when and where to split, since finding significantly better, problem-specific, values would require too much time.

3.2 Parallelization

Parallelization in Java is achieved by using threads, synchronization, and locks. A thread is a separate process that runs in the same runtime. Threads are just like any other objects, except that they have a start-function that initializes and starts a new process in the Java Virtual Machine (JVM). The thread can then operate independently from the other processes in the runtime.

The most difficult part of a parallel program is synchronization and ensuring mutually exclusive access to objects. Both can be achieved by using the Java-keyword *synchronized*, which locks the entire object that the synchronization is being applied to. For example, if a function is declared `synchronized` the object that the function is in will be locked so that no other thread can access the object until the synchronization is released, i.e., the function call is finished. Synchronization is a rather crude method to ensure mutual access, a better and more precise method is to use locks, which is why mostly locks were used in this project. Java provides a reentrant lock class which is very easy to use. Mutual access to specific objects or variables can be achieved by declaring a lock, locking it every time the variable is used, and then unlock when one no longer uses the variable. Since the lock is reentrant, functions that lock a variable can still run other functions that also want to lock the variable. This makes it a lot easier to write asynchronous programs where separate threads may call functions in each other simultaneously.

The drawback of locks and synchronization is *deadlock*, i.e., when a thread has a lock and is waiting for another thread that in turn is waiting for the lock that belongs to the first thread. Since both threads have locks, and are waiting for each other, neither of them can proceed. Although there are algorithms for “backing-up” in order to resolve the deadlock, the simplest method is to design the program so that no deadlocks can occur. Although such design can be difficult, trying to debug a program that gets stuck in deadlocks is even harder.

Just like threads in other programming languages, threads in Java can run in parallel, depending on the operating system. In the case of Linux, Mac OS, and Windows this is done automatically. Further, the JVM provides means of detecting the number of processors that are available to it. In the case of multi-core processors, the JVM will see them as separate processors. Figure 8, below, describes the process of parallelization, with two available processes.

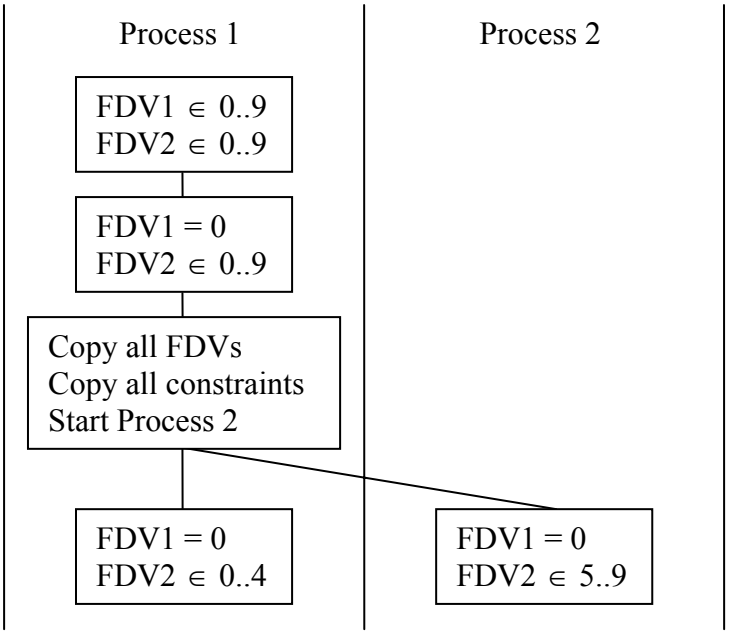


Figure 8: The parallelization procedure.

In the first box in Process 1 in Figure 8 the search process is initialized. In the following box FDV1 has been assigned a value of 0, this is the first level of the recursion. In the next box Process 1 has found that Process 2 is not currently executing, and proceeds to copy all the information needed to run Process 2 in parallel. Then, in the fourth stage, both search processes are running in parallel, in this case the domain of FDV2 was split in half.

For a more detailed description of the parallel search progression, see the second action sequence diagram in Appendix A. That diagram shows a problem similar to the one in Figure 8.

3.2.1 Threading

We have extended the search methods to include an array of threads, with a size of up to the number of processors available to the JVM. Each thread is linked to a corresponding search-object. Whenever a search process detects that one of the threads are free for execution it initializes that thread using the information that is needed to continue the search. This will be covered in detail in the section on copying. One limitation is that the main thread cannot be called by the threads it creates at the initialization. All other threads can call each other, however.

When a thread has been initialized it is started and the search continues, both in the original and the new thread. Since there will always be a main thread that initializes the other searches, it is enough to make sure that this *parent thread* keeps the solutions found during the search. This means that all threads share the same list of solutions, which then has to be locked whenever it is accessed. Further, in the case of searchOne all the threads are to terminate whenever a solution is found.

In searchAll and searchBB the parent thread needs to be kept running even after it terminates. The reason is that not all the possibilities that need to be tested may have finished. Therefore, there is a need to synchronize the last part of the execution, in order to make sure that all solutions are stored in the parent thread, without the JVM terminating. We achieved this by making the parent thread wait until all other threads have terminated. Before the wait

begins, a new thread is created, that the other threads can call. Otherwise one processor would be unused when the parent thread has completed its search.

Since Java 1.5 does not allow threads to be restarted, a new thread must be created whenever a search process has finished. Although threads require quite a lot of memory, needing to create new threads is not really a problem, since the memory of the previous thread will be reused.

3.2.2 Copying

The drawback of the parallel extension to the solver is that the entire store has to be copied, i.e., all variables and therefore all constraints have to be deep-cloned. Hence, three processes will require about three times as much memory as a single threaded search. Given that managed programming languages like Java already have a significant overhead, the speed-up is more limited than for solvers written in, e.g., C.

In order not to use unnecessarily much memory, only the information needed to continue the search is sent to the newly created process. This means that the search can only progress from the starting point and go downwards in the tree. Therefore, the search process will return instead of trying to backtrack, when all the values have been tested. The drawback is that the assignments made up to the point of splitting must also be stored and copied.

Although one could avoid creating new objects and instead use a more advanced representation of variables, domains, and constraints, this may not necessarily be beneficial. The price of minimizing copying is that one instead has to use locks or synchronization in order to ensure mutually exclusive access. And while the *new*-operation in Java is usually deemed as rather costly, the cost thread synchronization and locking can be just as high.

3.3 Distribution

Java provides a library for distributing programs, which allows almost seamless operations between computers. The only real difference is that the distributed object needs to be of a special type that makes it *exportable*. When an object is exported a *stub*, i.e., a short description of the object, is sent to the JVM that the function call is being made to. The function call itself is made through an automatic remote invocation procedure in the RMI (Remote Method Invocation) library. The parameters and return value of a function needs to be serializable, i.e., there has to be a way to translate the object to and from a network-sendable form. In this thesis, the default serialization methods in Java were used, this means that there will be some additional overhead in the communication.

The negative aspect of having lots of automated functions, which can be used with minimal effort, is that one cannot control the internal aspects of the system. The serialization for instance is called automatically when the RMI-call is made. This in itself is not a problem, but we designed the extensions to the solver to be *asynchronous*, which means that the automatic parts of the RMI system will run in a separate thread. Using synchronized remote operations is not an option, since the caller would then have to wait for the return of the remote function call before it could proceed. By placing the remote call in a separate thread, one can continue running the original process independently of the remote function. Then, however, one has to take the automatic parts of the RMI system into account when the remote call is placed, and when it returns.

One further complication when using RMI in Java is exception handling, the exceptions that occur on the remote machine will be sent to and reported on the machine that made the call. This makes it very hard to debug programs when several calls have been made in

sequence. For instance, if machine A has placed a remote call to machine B, and machine B subsequently calls a function on machine C. Then, if machine C gets an exception that terminates the function, the exception will be sent to machine B. If B cannot handle the exception and has to terminate, then the exception will be reported on and only on machine A. This chain of exception propagation can be quite hard to trace when inspecting the call stack of the exception. Especially since the network host names are not reported.

We have not used the full power of distribution in Java in this thesis. For example, Java can automatically transfer object between JVMs if the object is unknown to the JVM being called. Also, by distribution one often refers to the distribution of an object that is accessible to all the machines involved in the distribution. In the case of distributed search one could achieve this by making the search object itself distributed and allow the different machines to take over a part of the search process. This, however, would be unnecessarily complex, since the search process is recursive every level of the recursion is an obvious point for placing a remote call. Figure 9 describes the distribution process graphically.

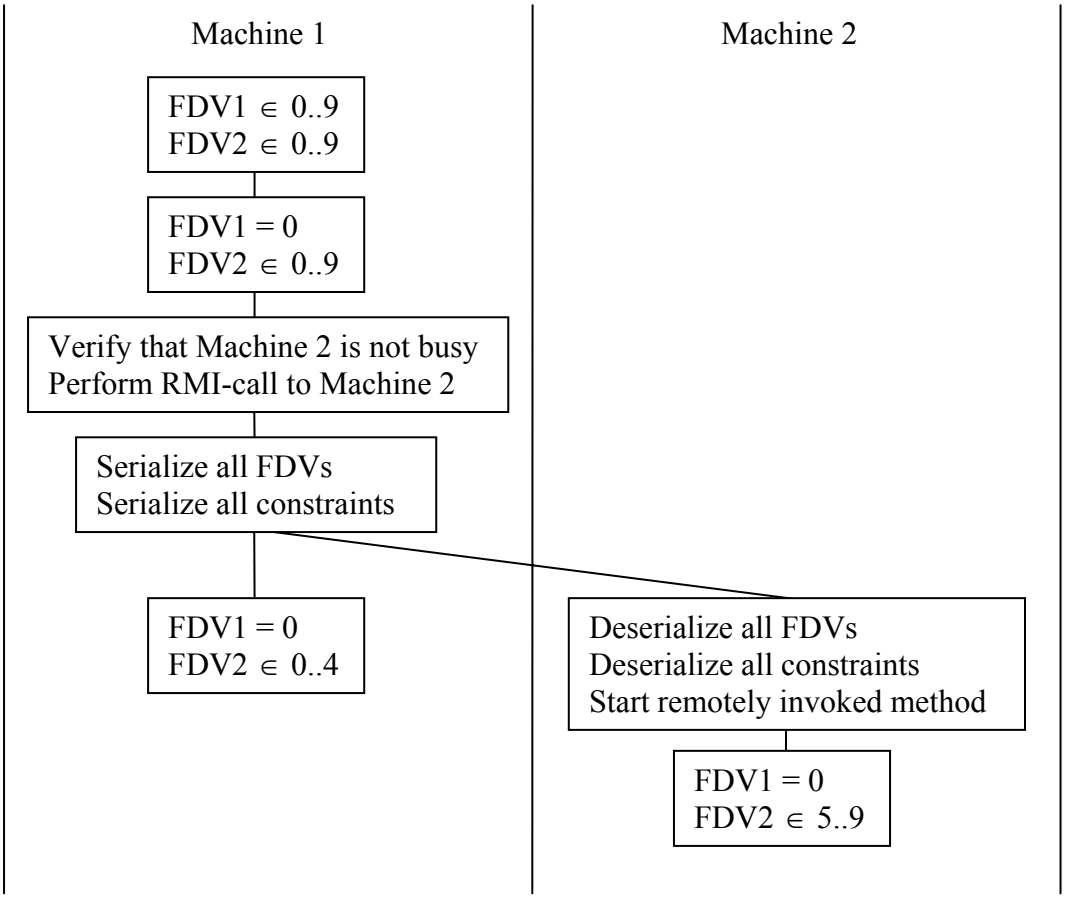


Figure 9: The distribution procedure.

In Figure 9 the same problem as in Figure 8 is described, only this time the search is not parallelized, but distributed. In the third stage Machine 1 finds that Machine 2 is not currently executing, and therefore places a RMI-call. In the fourth box in Machine 1, the serialization takes place, this is done automatically by the RMI-system in Java. After Machine 2 has deserialized the information, which is also done automatically, the search process starts. At this point the search is performed on Machine 1 and 2 concurrently. Just as in Figure 8, the domain of FDV2 was split in the middle.

The third action sequence diagram in Appendix A shows the distributed search in more detail, especially with regard to when to split the workload. The problem in that diagram, and the one for parallel search is similar to the problem in Figure 8 and Figure 9.

3.3.1 RMI Structure

Instead of using a distributed search object, the distribution is done in a client-server form.

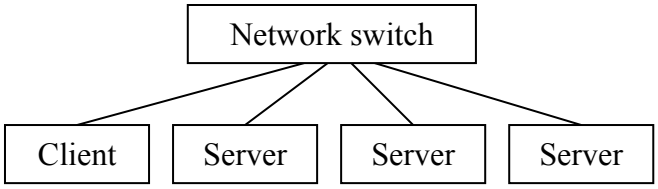


Figure 10: Client-server structure on a LAN.

Figure 10 shows how a client-server structure appears on a local area network. The switch in this case is not a computer but merely a communications hub that forwards the network messages between the computers.

The client is the computer where the constraint program is run, in Figure 6 this would be the `SendMoreMoney` class. The servers, on the other hand, are simply waiting for a remote call, which specifies the kind of search, and all the information needed to continue the search from the position where the client placed the RMI call. In order to achieve load balancing, the client can call all servers and the servers can call each other. The servers may not call the client, however. This poses a bit of problem if there are only a few machines and the client shares too little of its workload, but it makes the end synchronization a lot easier when performing searchAll or searchBB.

The servers have two threads for *multicast communication*, one thread for announcing its presence to the other machines and one for listening to such communications. The multicast broadcasts in Java sends a UDP (Universal Datagram Protocol) package to all the machines in a specific group which is defined by the *time to live* (TTL) of the package. Each server builds a list of which servers are running and which are shutting down. The server object is the only object that is distributable in the program. Whenever a new server is started and sends its announcement message the other servers retrieve a remote-reference to the new server. Since it is a reference, it has to be removed after the server shuts down, the same reference would be invalid if a new server on the same network address was started.

Each server has a list of threads for the remote calls that it has placed. After the RMI-call in the thread has returned, the thread is removed. The solutions found by a server are stored locally and are retrieved by the client at an appropriate time. When searching for all possible solutions the client has to wait for all the machines to terminate. This can be quite time consuming if the calls have been made in several steps between servers. In searchAll and searchBB the client waits until all RMI-calls have returned before retrieving the relevant solutions. In the case of searchOne the client terminates as soon as a solution has been found and communicated to the client. During searchBB the cost of the solutions that are found are reported to all the other computers.

3.3.2 Serialization

When placing a call to a remote function, the data sent as parameters needs to be sent along. In order to be able to send the data over a network the objects needs to be serialized. The default serialization in Java serializes all objects and variables that are not declared as *static* or *transient*. Static means that the object exists only in one form in the class in the entire JVM, and is therefore not valid in another JVM. Transient means only that the object should not be part of the default serialization. Although this gives a certain level of control over the default serialization procedures, there will still be a significant unnecessary overhead. Static and transient fields in a class will be initialized to their default value when the class is deserialized.

In order to be able to start the search, the same information as in the case of parallelization is necessary. The called machines needs to know what kind of search it is, all the variables, all constraints, the solution so far, and the variables left to be named. The serialization takes at least an order of a magnitude more time than copying. The number of variables is the most important factor for how much time the serialization takes. In this version of the extensions, all the needed information will be sent when placing the RMI-call, this creates a lot of unnecessary communication since a lot of the information will not change between calls.

Since the RMI calls are made in separate threads, the automatic serialization calls will be made independently of the original process. Therefore one has to synchronize on the last object to be serialized. In this case the last parameter in the remote call is the solution, until that object has been serialized the original process has to wait, otherwise some of the information might be out of phase, as the search will have time to proceed further down the tree.

Chapter 4

Experiments and Results

The experiments used to test the performance are two well known problems: finding the optimal Golomb ruler and finding one and all solutions to the N-Queens problem, both of which will be defined later. All tests were run on computers with fairly uniform performance and with some, but minimal, load outside the test program.

4.1 Experiment Setup

The computer used in testing parallelization is a Sun V40z with 4 AMD Opteron 850 processors. The processors have a speed of 2.4 GHz and they have 1 MB of L2 cache memory. Each processor has 4 DIMM (Dual In-line Memory Module) slots, each equipped with 1 GB of DDR400 memory, 16 GB in total, with an effective buss speed of 800 MHz. Each processor has a separate bi-directional link to the memory. The operating system is Red Hat Linux with the Linux kernel 2.4.21.

The hardware for distribution are 32 computers with AMD Athlon 64 3000+ (2.0 GHz) with 512 KB of L2 cache and 512 MB of RAM. All these machines ran Debian Linux 2.4.27. The machines are connected via a fully switched 100 megabit LAN. The Athlon machines are grouped in units of twelve, which are separated by at most three levels of switching.

4.2 Golomb Ruler

4.2.1 Problem Description

The Golomb ruler problem is defined as finding solutions to the list of n non-negative numbers where the absolute values of all the possible difference-pairs are unique. Formally:

$$a_1, a_2, ..., a_n \in Z^+$$
$$\left|a_i - a_j\right|, i \neq j \ \ is\ distinct$$

Since finding one possible ruler is simple, the experiment was done the traditional way by finding the optimal ruler, i.e., the one with the lowest cost. The cost is defined as the value of the highest number. In order to prove the optimality, one has to try all possible values that can lead to a cheaper solution. This is done by running a branch and bound search, using the last variable in the ruler as the cost function. The optimal Golomb ruler for n = 4 is [0, 1, 4, 6], and obviously has length (cost) 6. Sometimes there are several rulers with optimal cost, Golomb-5 for instance has optimal rulers [0, 1, 4, 9, 11] and [0, 2, 7, 8, 11], both with cost 11. The time complexity for finding the optimal Golomb ruler is exponential in n. There are

several research papers and websites on the matter (Atkinson et al., 1986; Lam & Sarwate, 1988; Distributed.net, 2006). At the time of writing the optimality has been proven up to n = 24, this was part of the Distributed.net project. Other research on Golomb rulers are focused on finding one solution, with no guarantee of optimality. This is done through mathematical techniques such as projective plane construction and affine plane construction. Through these method Golomb rulers of length 157 and 158 has been found (Shearer, 2006).<

4.2.2 Results with Parallelization

All tests were run at least three times, the tests were done for a value of n between 9 and 12. The execution times were as follows, all times are in seconds:

	Number of processes			
n	**1**	**2**	**3**	**4**
9	1.4	1.3	1.2	1.4
10	8.8	5.7	3.9	4.2
11	202	120	87.0	68.3
12	2397	2030	861.5	786.5

Table 1: Execution times in seconds for Golomb ruler.

As seen in Table 1, there is little time to be gained when the problem is small, therefore the resulting graphs presented below, will be of the larger problems where the time fluctuates less.

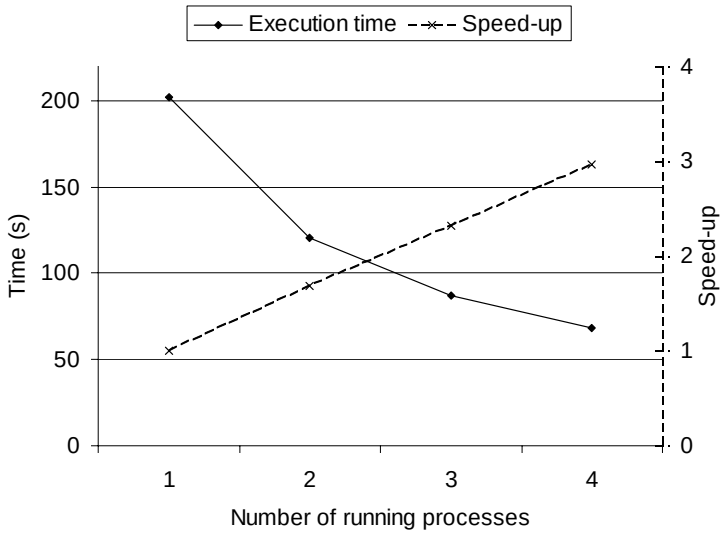


Figure 11: Execution time and speed-up of Golomb-11.

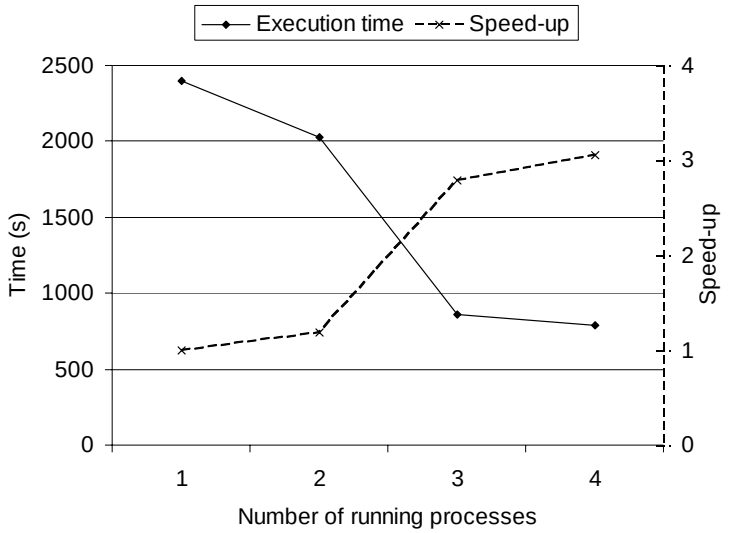


Figure 12: Execution time and speed-up of Golomb-12.

The results for Golomb-11, in Figure 11, show, as can be expected, that the gain in runtime is reduced for each processor, and that the speed-up is sub-linear. The case of Golomb-12, in Figure 12, is harder to explain. There is no obvious explanation for the strong difference between two and three processors. But given the lower gain for four the reason could be down to a lack of load balancing, where using two or four processors lead to a less uniform tree where processors are idle a lot longer.

The load distribution looks approximately the same for the different versions of Golomb, and the different amount of allowed processes. Again, the case of more processors is the most interesting due to more events.

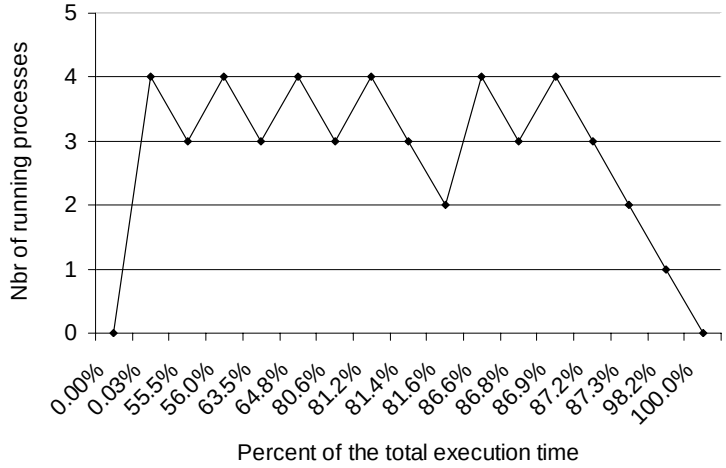


Figure 13: Processor load with four allowed processes for Golomb-11.

Figure 13 shows roughly what could be expected, the load reaches its maximum at 0.03% of the search since the parent thread uses all available threads as soon as possible. The second process is started on the first level of the search and the third and fourth are either started by the parent thread or by one of the child threads. Noteworthy is also that at 87.3% only two threads are left to perform the last part of the search.

4.2.3 Results with Distribution

The test of distribution were more restricted than for parallelization, the reason is the difficulty in ensuring that the machines had minimal load for the entire duration of the test. All the times are in seconds.

	Number of computers						
n	**1**	**2**	**3**	**4**	**8**	**16**	**32**
9	1.7	1.5	2.8	2.2	2.3	2.6	3.2
10	14	8.7	5.8	5.2	4.0	4.1	4.3
11	344	195	142	98.0	77.0	59.5	58.5
12	3928	3305	1417	1442	902.0	753.0	620.0

Table 2: Execution times for Golomb ruler with distribution.

Again the most interesting situations are $n = 11$ and $n = 12$, where the variation is smaller. The time and speed-up graphs for those situations look as follows:

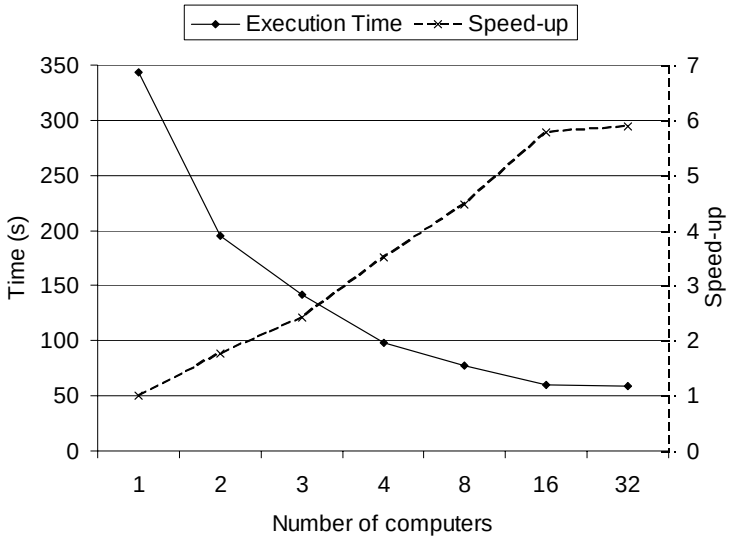


Figure 14: Execution time and speed-up for Golomb-11.

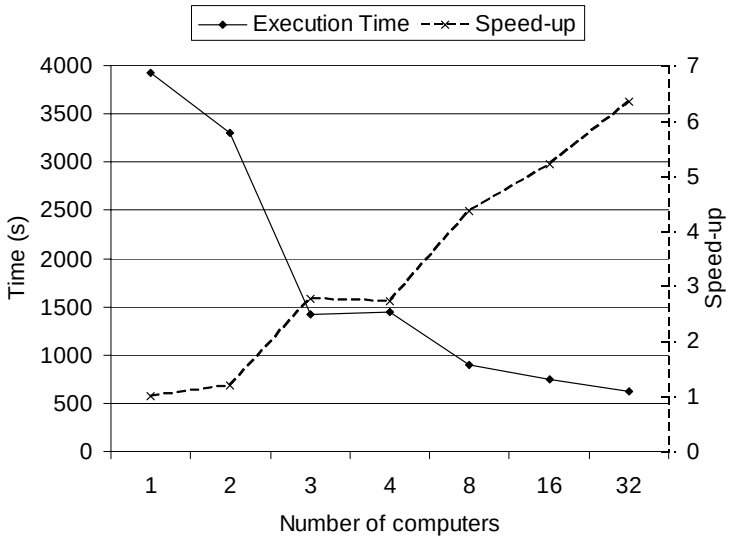


Figure 15: Execution time and speed-up for Golomb-12.

The distribution of Golomb-11, as shown in Figure 14, leads to an almost linear speed-up until the communications become too cumbersome. Just as in the parallel case the distribution of Golomb-12, in Figure 15, shows some strange results for certain numbers of search processes. This is possibly due to one large part of the tree that is split of possibly contains a lot of solutions, that another computer will invalidate when there are more than 4 machines available.

The load distribution for Golomb-11 is seen in Figure 16.

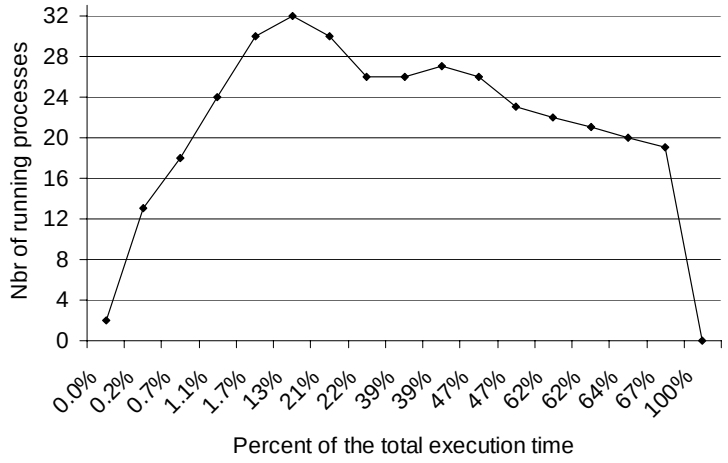


Figure 16: Load balancing for Golomb-11 when using 32 machines.

The load balancing only keeps the load full for a brief amount of time and the time-weighed average load is 55% of the maximum. This means that there is obviously room for improvement when using 32 processors.

4.3 N-Queens

4.3.1 Problem Description

The N-Queens problem is a chess-related problem that consists of finding a way to position n queens on an $n*n$ chess board in a way that no queen can strike another. Just as with the Golomb ruler, it is fairly easy to find one solution, therefore the tests were done on both searchAll and searchOne. The main difference between the Golomb and the N-Queens problem is the number of variables, constraints, and the structure of the domains. For N-Queens, copying in the case of $n = 200$, takes about twice as long as for $n = 12$, which in turn takes about twice as long as the copying in Golomb-10. For Queens-200 there are 598 FDVs and 401 constraints, for $n = 12$ there are 34 variables and 25 constraints. For Golomb-10 there are 55 FDVs and 102 constraints. The reason the copying takes longer for Queens-12 than for Golomb-10 is that the domains are pruned differently, in N-Queens the domains are split into many small intervals, whereas Golomb have fewer, but larger, intervals.

One complication with the N-Queens problem is the size of the domains of the FDVs, the size for all queens is initially n. Since the domains will remain quite large, but split into several smaller intervals, it can be difficult to achieve a good load balancing when relying on splitting of the domains. If the intervals in the domains are too small it may not be possible to split them in a useful way.

4.3.2 Results with Parallelization

Just as for Golomb the tests were run several times. Both the case of search for one solution and search for all solutions were tested, and gave the following results, times in seconds:

	Number of processes			
n	**1**	**2**	**3**	**4**
100	0.80	0.83	0.87	0.97
200	4.9	5.3	5.6	4.1
320	13	14	15	18
500	56	57	60	68

Table 3: Execution times in seconds when searching for one solution.

	Number of processes			
n	**1**	**2**	**3**	**4**
10	1.8	1.4	1.4	1.4
11	5.9	3.9	3.6	3.4
12	25	14	12	12
13	128	76.3	64.3	63.6

Table 4: Execution times in seconds in search for all solutions.

Table 3 and 4 shows the results when searching for one solution to the N-Queens problem, and when searching for all solutions respectively. As can be seen in Table 3 the parallel search actually takes longer in almost all versions of the problem. This happens because there are only a handful of backtracks in the entire search. Therefore there is no room for improvement, all that happens is that the processes may find solutions simultaneously. Hence the execution time is the same as with no parallelization plus the time it takes to start the other processes. The reason why $n = 320$ was chosen, is that for $n = 300$, and some other values, the search can get stuck in one inconsistent part of the tree where the actual cause of the inconsistency requires several levels of backtracking to correct.

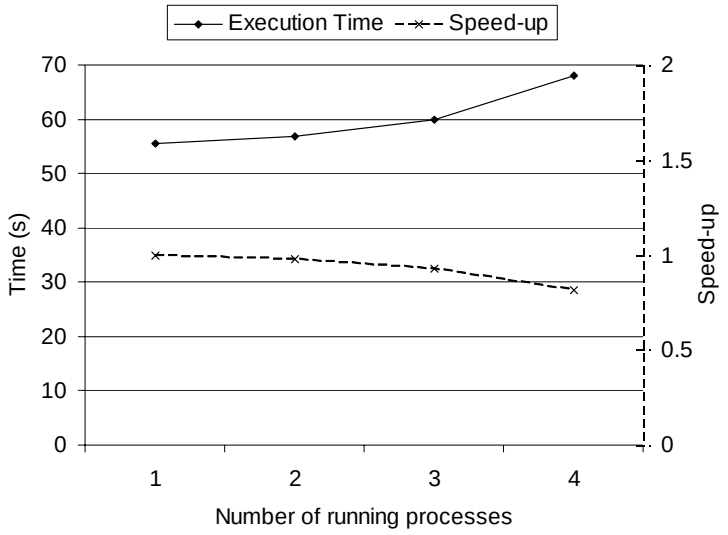


Figure 17: Execution time and speed-up when searching for one solution to Queens-500.

Given that finding one solution to the N-Queens problem leads to a slowdown the results are mostly of interest as a comparison to the search for all solution.

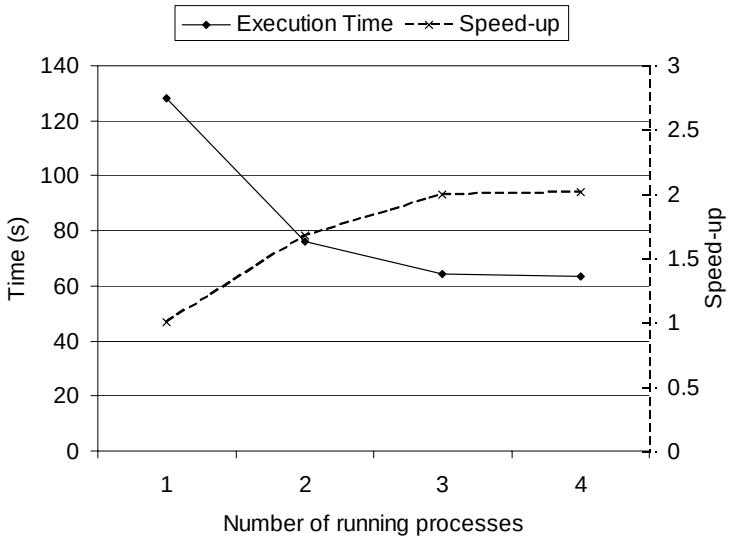


Figure 18: Execution time and speed-up of search for all solutions to Queens-13.

The gain of parallelization is much better when searching for all solutions compared to when searching for one solution. Just as has been predicted previously, however, the gain in searchAll of N-Queens is lower than for Golomb, the reason is the size of the information that needs to be copied.

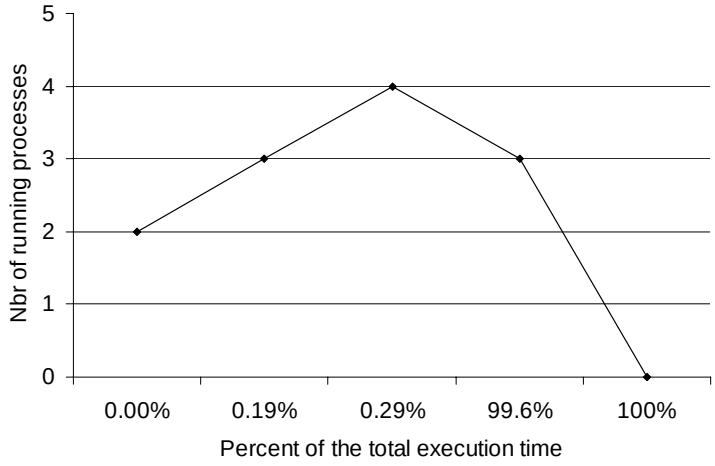


Figure 19: Load balancing when searching for one solution to Queens-500 using four processes.

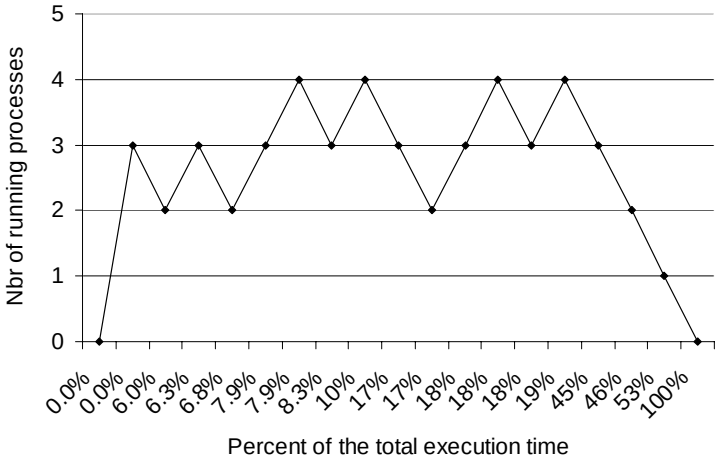


Figure 20: Load for search for all solutions to Queens-13 with four allowed processes.

Although the load balancing looks good in the situation in Figure 19 – all processors are used more than 99% of the time – the actual gain is, as Figure 17 showed, not good. The load in Figure 20, on the other hand is actually quite bad despite the fact that Figure 18 showed a speed-up factor of 2. More than 45% of the work is done by a single processor. The reason is most likely that there are many solutions in one small part of the tree, where the working processor is not allowed to send on the work to the other threads.

4.3.3 Results with Distribution

The test were run several times, all times are in seconds.

	Number of computers						
n	**1**	**2**	**3**	**4**	**8**	**16**	**32**
100	0.70	0.73	0.80	1.0	1.0	1.1	1.2
200	7.3	7.2	5.2	5.2	5.6	6.9	7.8
320	18	37	30	29	31	32	33

Table 5: Execution times when searching for one solution.

	Number of computers						
n	**1**	**2**	**3**	**4**	**8**	**16**	**32**
10	1.7	1.3	1.0	1.0	1.1	1.1	1.2
11	8.3	4.8	3.8	3.4	3.6	3.3	3.3
12	39	22	15	12	12	12	11
13	226	138	109	86.2	50.3	50.7	44.2

Table 6: Execution times when searching for all solutions.

As seen in Table 5, Queens-500 was not run. The reason is that for more than one machine, the test took too long to complete. The graphs for execution time in the case of $n = 320$ and $n = 13$, respectively, look as follows:

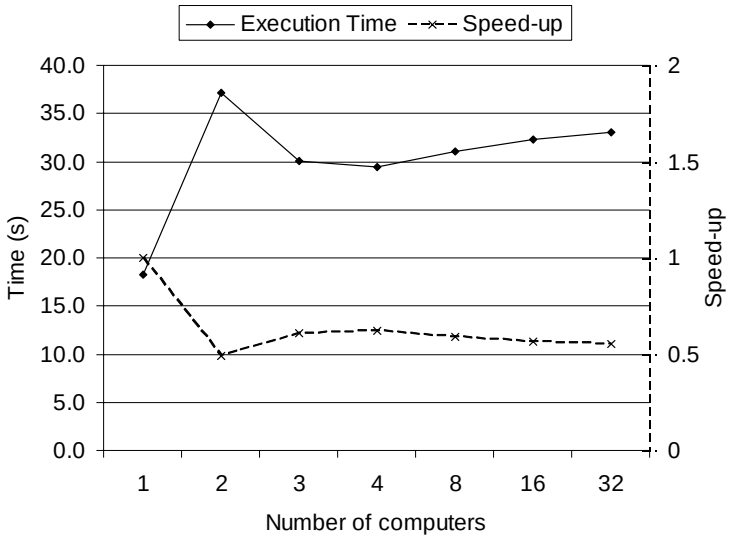


Figure 21: Execution time and speed-up in the case of searchOne with $n = 320$.

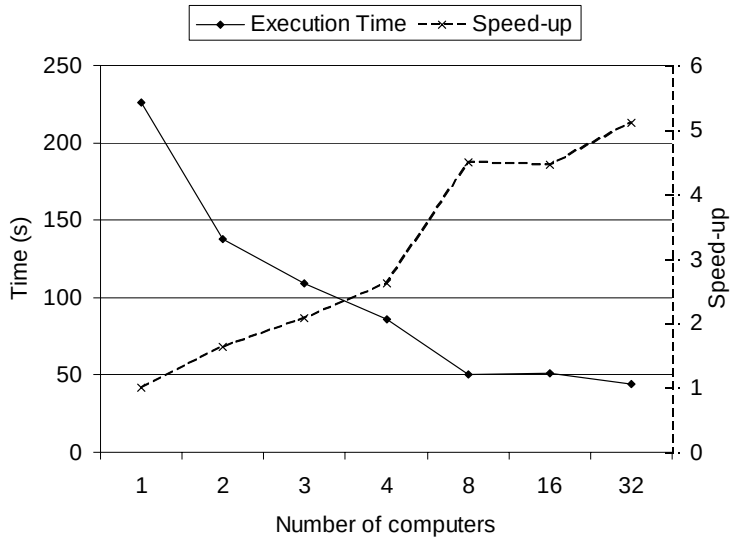


Figure 22: Execution time and speed-up for searchAll, $n = 13$.

Just as for the parallel search for one solution to N-Queens actually leads to a slow-down compared to the serial case, as seen in Figure 21. However, when searching for all solutions using distribution, the speed-up, as Figure 22 shows, is higher than when using parallelization. The speed-up is almost equal to parallelization up to 3 computers and threads, and then the distribution becomes significantly better for 4 processes/computers.

The load balancing in the case of searchOne is almost identical to the situation in Figure 19. When searching for all solution, however, it is quite different, the results are shown in Figure 23.

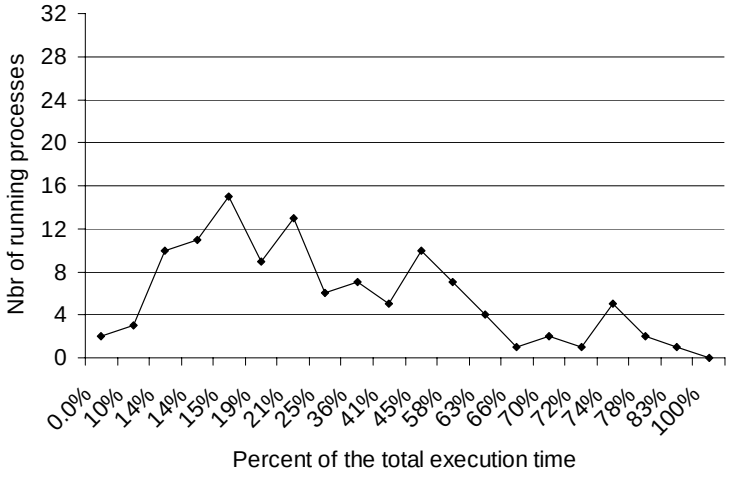


Figure 23: Load balancing during searchAll for Queens-13 with 32 computers.

The load balancing during parallel searchAll was, as Figure 20 showed, not very good. In the case of distribution it is even worse. As can bee seen in Figure 23, the load never reaches 50% of maximum, and the time-averaged load is only about 14%. This kind of result was almost to be expected, however, given the low gain in speed-up when using 32 machines compared to using eight machines. The load balancing performs significantly better when using fewer than 16 computers.

Chapter 5

Conclusions

In this thesis we have explored and tested how parallelization and distribution of a constraint solver can be performed. The solver was written in Java, and uses a recursive algorithm, this facilitated the development of the parallel and distributed extensions. The support in Java for multithreading and remote method invocation proved helpful and the test results were satisfactory.

There are two main conclusions to be made: parallelization and distribution is fairly easy to achieve in Java, and parallel and distributed programs often runs faster.

The first conclusion is obvious, all the work needed to modify the solver was performed by a single student within the timeframe of a master thesis, i.e., 20 weeks. The biggest difficulty was debugging the distribution, where exceptions in the RMI-system and the network communications slowed down the development.

The second conclusion is true in the case of search for all solutions and search for the optimal solution, but not when searching for just one solution. The reason why finding one solution to the N-Queens problem does not achieve a speed-up is that the search process only backtracks a few times, even when $n = 500$. Therefore the solution is found in a straight line down the search tree, leaving very little room for improvement.

Just as mentioned in the introduction, branch and bound search can achieve a super-linear speed-up, thereby outperforming the search for all solutions. The effect is more pronounced in the case of parallelization, which is most likely caused by the overhead of communicating the cost to the other machines. Although the load balancing in the distributed searchBB is a lot better than for searchAll, as Figure 16 and Figure 23 shows, this is not true when using fewer machines. Up to about eight machines, the average load is roughly the same in Golomb and N-Queens.

One further conclusion is that parallelization becomes limited sooner than distribution. As mentioned previously the effect is caused by the garbage collector running in parallel. It is conceivable that in some cases it is better to only use three processors instead of all four, leaving one free processor for the garbage collection. This would be especially true when searching for all solutions, since the speed-up is not as reduced in Figure 11 as in Figure 18.

The cost of communication being greater for a language with automatic memory-management starts to show when using 16 machines and more. The somewhat poor load balancing is of course related to the result, but the greater overhead in communication emphasizes the need further.

Despite the low speed-up when using many machines, parallelizing and distributing programs in Java is worthwhile. In the case of an already existent program the modifications can be quite small if it is easy to find when and where to split the workload. When distributing for a lot of machines, more efficient methods for serialization and load balancing may be necessary, however. And although the speed-up is quite good in parallelization, multi-core processors may benefit more from letting the garbage collector run on a free core. This holds especially if the cache memories are shared between cores, and the threads use separate copies of the data.

Future Work

There are several performance enhancing modifications that can be made to the program. The most important for distribution is to reduce the communication overhead and improve the serialization process. By sending a copy of the store as a multicast in the beginning, and then sending only the modifications made to that copy, one could severely reduce the overall communication. Also, by sending only values, instead of full objects, one could reduce the communication even further.

In the case of parallelization, the biggest room for improvement probably lies in finding a load balancing that takes memory usage into account, and just as in distribution try to copy only the necessary information. The benefit of copying as little information as possible is not nearly as high as for distribution, however. The overall memory needed to run all the threads is probably more important since it immediately reflects the need for garbage collection. Quite possibly, one could increase the performance significantly by writing a custom library for memory allocation and deallocation.

References

Aida, K. & Natsume, W. (2003). Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCCGRID'03)*.

Amdahl, G. M. (1967). Validity of the single processor approach to achieve large scale computing capabilities. *AFIPS Conference Proceedings*, 30, 483-485.

Ashford, R. W., Connard, P. & Daniel, R. (1992). Experiments in solving mixed integer programming problems on a small array of transputers. *Journal of Operational Research Society*, 43, 519-531.

Atkinson, M. D., Santoro, N. & Urrutia, J. (1986). Integer sets with distinct sums and differences and carrier frequency assignments for nonlinear repeaters. *IEEE Transactions on Communications*, 34, 614-617.

Baker, M., Buyya, R. & Laforenza, D. (2002). Grids and grid technologies for wide-area distributed computing. *Software: Practice and Experience*, 15, 1437-1466.

cajo (2006). *https://cajo.dev.java.net*.

CORBA (2006). *http://java.sun.com/developer/onlineTraining/corba/corba.html*.

Dechter, R. (2003). *Constraint Processing*. San Francisco, CA: Morgan Kaufmann Publishers.

Dehne, F., Ferreira, A. & Rau Chaplin, A. (1990). Parallel branch and bound on fine grained hypercubes multiprocessors. *Parallel Computing*, 15, 201-209.

Disolver (2006). *http://research.microsoft.com/~youssefh/DisolverWeb/Disolver.html*.

Distributed.net (2006). *http://www.distributed.net*.

Forouzan, B. A. (2001). *Data communication and networking - 2nd edition update*. New York, NY: McGraw-Hill Higher Education.

Garey, M. & Johnson, D. S. (1979). *Computers and intractability*. San Francisco, CA: Freeman.

Grama, A. Y. & Kumar, V. (1995). A survey of parallel search algorithms for discrete optimization problems. *ORSA Journal of Computing*, 7, 365-385.

Hermenegildo, M. (2000). Parallelizing irregular and pointer-based computations automatically: Perspectives from logic and constraint programming. *Parallel Computing*, 26, 1685-1708.

Hirayama, K. & Yokoo, M. (2004). The distributed breakout algorithms. *Artificial Intelligence*, 161, 89-115.

JaCoP (2006). *http://www.cs.lth.se/Education/LTH/EDA340/info/instruction/instruction.pdf*.

Lam, A. W. & Sarwate, D. V. (1988). On optimum time-hopping patterns. *IEEE Transactions on Communications*, 36, 380-382.

Land, A. & Doig, A. (1960). An automatic method for solving discrete programming problems. *Econometrica*, 28, 497-520.

Lawler, E. L. & Wood, D. E. (1966). Branch and bound methods, a survey. *Operations Research*, 14, 699-719.

Marriot, K. & Stuckey, P. J. (1998). *Programming with constraints: An introduction*. Cambridge, MA: The MIT Press.

Mitra, G., Hai, I., & Hajian M. T. (1997). A distributed algorithm for solving integer programs using a cluster of workstations. *Parallel Computing*, 23, 733-753.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics Magazine*, 38, 114-117.

Roucairol, C. (1996). Parallel processing for difficult combinatorial optimization problems. *European Journal of Operational Research*, 92, 573-590.

SETI (2006). *http://setiathome.berkeley.edu*.

Shearer, J. B. (2006). *http://www.research.ibm.com/people/s/shearer/grtab.html*.

Yokoo, M., Durfee, E. H., Ishida, T. & Kuwabara, K. (1998). The distribution constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and DATA Engineering*, 10, 673-685.

Yokoo, M. & Hirayama, K. (2000). Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3, 198-212.

Yokoo, M., Suzuki, K. & Hirayama, K. (2005). Secure distributed constraint satisfaction: reaching agreement without revealing private information. *Artificial Intelligence*, 161, 229-245.

Xu, C., Tschöke, S. & Monien, B. (1995). Performance evaluation of load distribution strategies in parallel branch and bound computations. *Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing*, 402-405.

Zivan, R. & Meisels, A. (2006). Concurrent search for distributed CSPs. *Artificial Intelligence*, 170, 440-461.

Appendix A

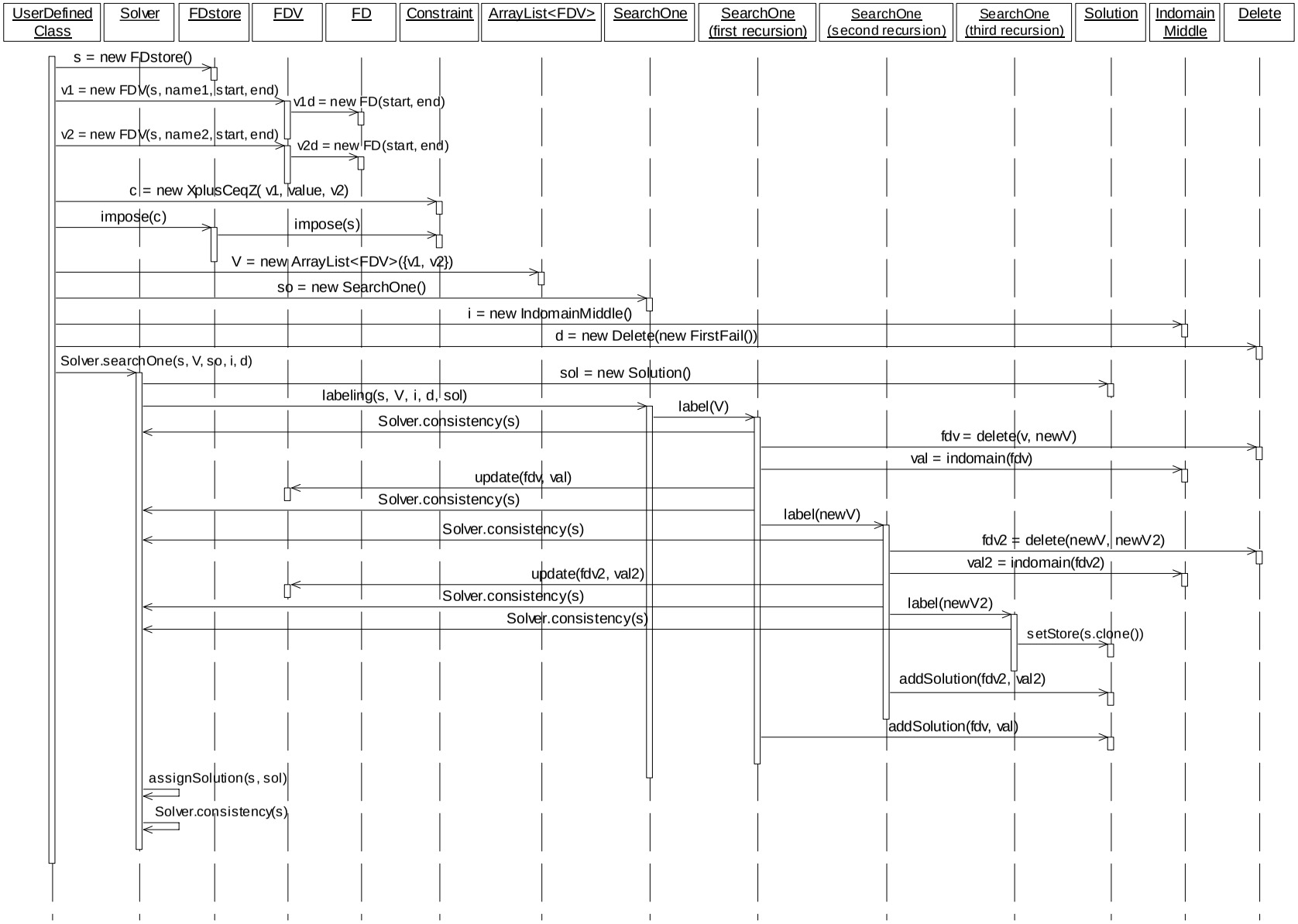
Action Sequence Diagrams

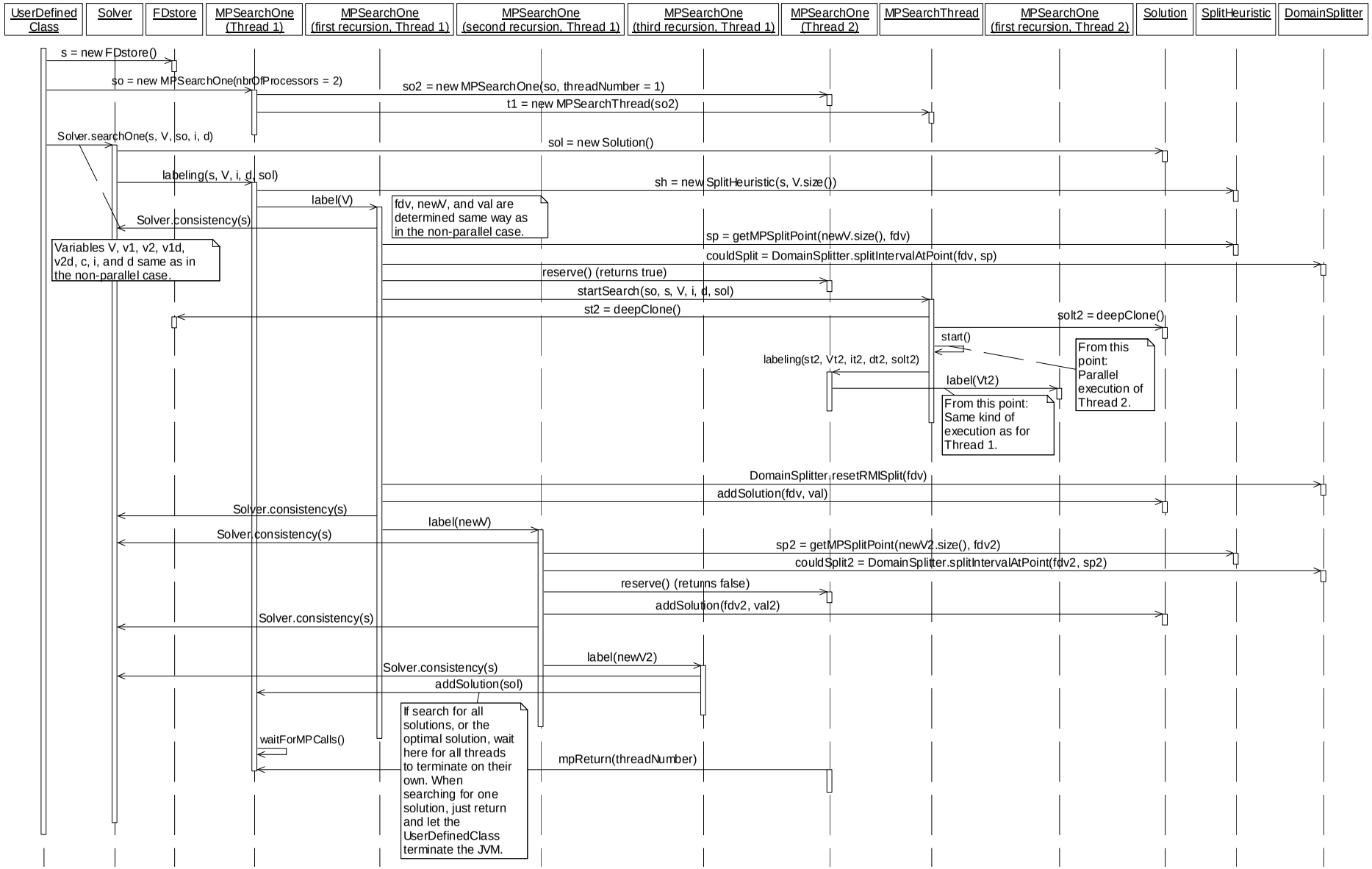
The action sequence diagrams in this appendix describe three versions of search. The first diagram shows the setup of a problem and how the search progresses in finding a solution. The search is the standard non-parallel search.

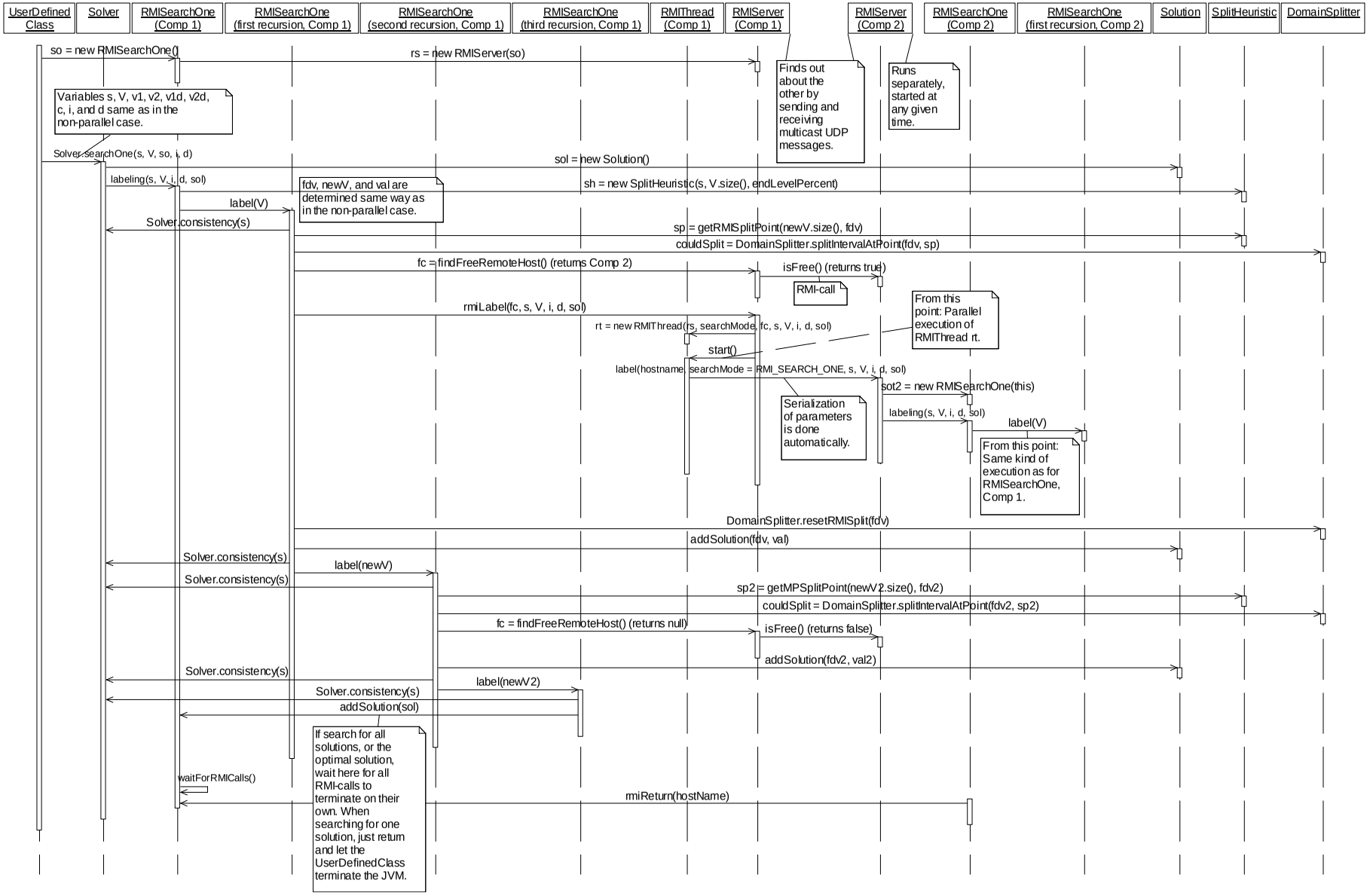
The second diagram describes the same problem as in the first image, but this time solved using parallel search.

The last diagram shows the same problem as in the two previous images, but solved using distributed search.

For the sake of readability some parts have been left out and the parallel searches only progress a single level, since the modeling tool did not support program parts running in parallel independently of each other.


UserDefined Class
Solver
FDstore
FDV
FD
Constraint
ArrayList<FDV>
SearchOne
SearchOne (first recursion)
SearchOne (second recursion)
SearchOne (third recursion)
Solution
Indomain Middle
Delete
s = new FDstore()
v1 = new FDV(s, name1, start, end)
v1d = new FD(start, end)
v2 = new FDV(s, name2, start, end)
v2d = new FD(start, end)
c = new XplusCeqZ(v1, value, v2)
impose(c)
impose(s)
V = new ArrayList<FDV>({v1, v2})
so = new SearchOne()
i = new IndomainMiddle()
d = new Delete(new FirstFail())
Solver.searchOne(s, V, so, i, d)
sol = new Solution()
labeling(s, V, i, d, sol)
label(V)
Solver.consistency(s)
fdv = delete(v, newV)
val = indomain(fdv)
update(fdv, val)
Solver.consistency(s)
label(newV)
Solver.consistency(s)
fdv2 = delete(newV, newV2)
val2 = indomain(fdv2)
update(fdv2, val2)
Solver.consistency(s)
label(newV2)
Solver.consistency(s)
setStore(s.clone())
addSolution(fdv2, val2)
addSolution(fdv, val)
assignSolution(s, sol)
Solver.consistency(s)


UserDefined Class
Solver
FDstore
MPSearchOne (Thread 1)
MPSearchOne (first recursion, Thread 1)
MPSearchOne (second recursion, Thread 1)
MPSearchOne (third recursion, Thread 1)
MPSearchOne (Thread 2)
MPSearchThread
MPSearchOne (first recursion, Thread 2)
Solution
SplitHeuristic
DomainSplitter
s = new FDstore()
so = new MPSearchOne(nbrOfProcessors = 2)
so2 = new MPSearchOne(so, threadNumber = 1)
t1 = new MPSearchThread(so2)
Solver.searchOne(s, V, so, i, d)
sol = new Solution()
labeling(s, V, i, d, sol)
sh = new SplitHeuristic(s, V.size())
label(V)
fdv, newV, and val are determined same way as in the non-parallel case.
Solver.consistency(s)
Variables V, v1, v2, v1d, v2d, c, i, and d same as in the non-parallel case.
sp = getMPSplitPoint(newV.size(), fdv)
couldSplit = DomainSplitter.splitIntervalAtPoint(fdv, sp)
reserve() (returns true)
startSearch(so, s, V, i, d, sol)
st2 = deepClone()
solt2 = deepClone()
start()
From this point: Parallel execution of Thread 2.
labeling(st2, Vt2, it2, dt2, solt2)
label(Vt2)
From this point: Same kind of execution as for Thread 1.
DomainSplitter.resetRMISplit(fdv)
addSolution(fdv, val)
Solver.consistency(s)
label(newV)
Solver.consistency(s)
sp2 = getMPSplitPoint(newV2.size(), fdv2)
couldSplit2 = DomainSplitter.splitIntervalAtPoint(fdv2, sp2)
reserve() (returns false)
addSolution(fdv2, val2)
Solver.consistency(s)
label(newV2)
Solver.consistency(s)
addSolution(sol)
If search for all solutions, or the optimal solution, wait here for all threads to terminate on their own. When searching for one solution, just return and let the UserDefinedClass terminate the JVM.
waitForMPCalls()
mpReturn(threadNumber)


UserDefined Class
Solver
RMISearchOne (Comp 1)
RMISearchOne (first recursion, Comp 1)
RMISearchOne (second recursion, Comp 1)
RMISearchOne (third recursion, Comp 1)
RMIThread (Comp 1)
RMIServer (Comp 1)
RMIServer (Comp 2)
RMISearchOne (Comp 2)
RMISearchOne (first recursion, Comp 2)
Solution
SplitHeuristic
DomainSplitter
so = new RMISearchOne()
rs = new RMIServer(so)
Finds out about the other by sending and receiving multicast UDP messages.
Runs separately, started at any given time.
Variables s, V, v1, v2, v1d, v2d, c, i, and d same as in the non-parallel case.
Solver.searchOne(s, V, so, i, d)
sol = new Solution()
labeling(s, V, i, d, sol)
fdv, newV, and val are determined same way as in the non-parallel case.
sh = new SplitHeuristic(s, V.size(), endLevelPercent)
label(V)
Solver.consistency(s)
sp = getRMISplitPoint(newV.size(), fdv)
couldSplit = DomainSplitter.splitIntervalAtPoint(fdv, sp)
fc = findFreeRemoteHost() (returns Comp 2)
isFree() (returns true)
RMI-call
rmiLabel(fc, s, V, i, d, sol)
From this point: Parallel execution of RMIThread rt.
rt = new RMIThread(rs, searchMode, fc, s, V, i, d, sol)
start()
label(hostname, searchMode = RMI_SEARCH_ONE, s, V, i, d, sol)
sot2 = new RMISearchOne(this)
Serialization of parameters is done automatically.
labeling(s, V, i, d, sol)
label(V)
From this point: Same kind of execution as for RMISearchOne, Comp 1.
DomainSplitter.resetRMISplit(fdv)
addSolution(fdv, val)
Solver.consistency(s)
label(newV)
Solver.consistency(s)
sp2 = getMPSplitPoint(newV2.size(), fdv2)
couldSplit = DomainSplitter.splitIntervalAtPoint(fdv2, sp2)
fc = findFreeRemoteHost() (returns null)
isFree() (returns false)
addSolution(fdv2, val2)
Solver.consistency(s)
label(newV2)
Solver.consistency(s)
addSolution(sol)
If search for all solutions, or the optimal solution, wait here for all RMI-calls to terminate on their own. When searching for one solution, just return and let the UserDefinedClass terminate the JVM.
waitForRMICalls()
rmiReturn(hostName)