university of groningen

faculty of science and engineering

Analysing Redundant Exploration of Parallel Search Algorithms

Bachelor Thesis

June 2022

Student: Derk Jan Pot

First supervisor: Dr. V. Degeler

Second supervisor: M. Medema

Abstract

This bachelor thesis aims to show the overhead of using parallel search for solving CSPs versus using sequential search. For our tests we used several problems from the MiniZinc Challenge 2021. We use the Gecode solver to solve these problems, which are divided in satisfaction problems and optimisation problems. We test if there are more or fewer nodes explored in the search space when we use more than 1 thread. Some of the satisfaction problems show that adding more threads results in more nodes to be explored in the search space, suggesting that some redundant exploration takes place. Other satisfaction problems however show the exact opposite results. Using nogoods from restarts often leads to more nodes to be explored, and therefore lead to greater overhead. However, there are some problems for which enabling nogoods from restarts lead to fewer nodes to be explored. Searching for all solutions rather than just the best solution shows us that more threads do not necessarily lead to more nodes to be explored, but enabling nogoods from restarts almost always results in more nodes to be explored compared to not using restarts. Quite a few of the satisfaction problems result in a timeout, making it harder to draw concrete conclusions. For the optimisation problems we get even more indecisive results. Adding more threads results either in more redundant nodes to be explored, fewer nodes to be explored compared to the number of nodes explored at 1 thread, or it does not see any change at all. Using no-goods from restarts have great influence on the number of nodes that will be explored by the solver, but it differs per problem if it has a positive (decrease) or negative (increase) influence.

Contents

List of Figures

List of Tables

1 Introduction

Constraint programming (CP) is an appealing technology for a variety of combinatorial problems which has grown steadily in the past few decades. The strengths of CP are the use of constraint propagation combined with efficient search algorithms [2]. Parallel search algorithms can be used for finding the optimal solution to a constraint satisfaction problem (CSP), since they can explore multiple solutions in parallel, thereby potentially reducing the overal search time. Parallel search algorithms may potentially perform some redundant search, because unlike sequential algorithms that can use the upper and lower bound on the cost to disregard subsequent solutions, a parallel algorithm may have already started to explore such solutions before the bound. It is also possible that we explore more of the search space since a solution has been found on one thread, but this result has not been synchronised yet with the other threads. It is not always possible to make use of the most effective pruning techniques to mitigate redundant search. Typically each processor gets assigned a part of the search space. When the optimal solution is located in the first partition, we will likely get some redundant exploration since other processors explore the other partitions in parallel and will already explore parts of the search space that would otherwise not have been considered.

The goal of the project is to find out how much redundant search takes place, how much this is a problem, and how much impact it has on the overall execution time of the algorithm. Needless exploration of the search space is not only redundant, but can also cost more electricity since the CPUs still do (redundant) work.

In this paper we analyse several CSPs and constraint satisfaction and optimisation problems (CSOPs). We use the Gecode solver [12] in sequential mode and for several numbers of threads. We then check to see if we explore more nodes in the search space or fewer when we use more than one thread. This should give us a reasonable overview whether using more threads does lead to more explored nodes (i.e. using parallel search leads to redundant exploration of the search space). We will also check the impact on using restarted search on the search. Using restarts avoids the problem of the heavy-tailed phenomenon [5], but it has the drawback that the same parts of the search tree can be explored several times. Using nogood recording at the end of each run can eliminate this drawback [6], therefore we will perform tests with nogoods from restarts, and tests without using restarts.

Finally, we will not only run the solver find the optimal solution, but we will also run the solver trying to find all solutions, and see how that impacts the amount of nodes we need to explore.

In Section 2 we look at the background information, where we explain what CSPs, parallel search algorithms, and nogoods are. In section 3 we look at the related work. Section 4 explains the methods we used, and all the problems that we looked at. Section 5 contains all the results. Similar results are discussed together. The conclusion and future work are found in Section 6 and Section 7, respectively.

2 Background Information

In this section we will give some definitions and background information for all the terms that will be used in the related works and in the methods.

2.1 Constraint Satisfaction Problems

The idea behind CP is to solve problems by stating which constraints must be satisfied by the solution(s) of the problem. A solver can then be used to fill in the values for the variables such that the constraint is satisfied. More formally, a CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is defined as a set of n *variables* $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, a set of *domains* $\mathcal{D} = \{D(X_1), \ldots, D(X_n)\}$, where $D(X_i)$ is a finite and discreet set of possible *values* for variable X_i, and $\mathcal{C}$ is a set of m *constraints* $\{C_1, C_2, \ldots, C_m\}$. A constraint on a set of variables is a restriction on the set of values that these variables can take simultaneously. A constraint C_{ij} on the ordered set of variables $\langle X_i, X_j \rangle$ is a subset of the Cartesian product $D(X_i) \times D(X_j)$ that specifies which combinations of values for the variables X_i and X_j are allowed such that the constraint is not violated. The whole set of combinations is referred to as the *search space*. An assignment on a subset of $\mathcal{X}$ is said to be *consistent* if it does not

violate any constraint [8]. A *solution* to the CSP is a consistent *total instantiation* of all the variables of the problem, i.e. no constraints are violated. Finding such a solution is known to be a NP-complete problem.

CP is often used for and deals well with combinatorial optmisation problems, such as the Traveling Salesman Problem and the Knapsack Problem. For optimisation problems one aims to finding the best (optimal) solution, maximising or minimising a given *objective function*. We can define a constraint optimisation problem (COP) as a quadruplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, obj \rangle$, where $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is a CSP, and $obj : Sol \mapsto \mathbb{R}$, where Sol is the set of all solutions of $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$.

It is usually not efficient or even feasible to solve a CSP by trying each possible value assignment and check if it satisfies all the constraints. Many works have been realised to make the solving more efficient in practice, by using for example optimised backtracking algorithms, heuristics, constraint propagation, non-chronological backtracking, or filtering techniques [10]. Most solvers commonly use several of these techniques for efficient solving. Constraint propagation is technique which prunes combinations of values from variable domains which cannot appear in any solution [2]. When combining constraint propagation with backtracking search, a (best) solution is always found or proven to not exist [7]. Backtracking search attempts to incrementally extend a partial assignment toward a complete soltuion, by repeatedly choosing a value for another variable and keeping the previous state of the variables so it can be restored if a failure (i.e. no value can be assigned to any remaining variable without violating a constraint) is encountered. The time complexity of backtracking algorithms naturally exponential, at least in $O(m \times d^n)$ where d is the largest domain. Solving a CSP is the traversal of a tree whose nodes correspond to the partial assignments and where the root of the tree is the initial problem with no assignment. Restart techniques generally allow to reduce the impact of bad choices performed thanks to heuristics, or of the occurrence of heavy-tailed phenomena, and typically use learning techniques like recording nogoods [10, 6]. A nogood is a partial instantiation that cannot be extended to a total consistent instantiation. When nogoods are recorded during backtracking search, they can then be used later to prevent the exploration of useless parts of the search tree.

2.2 Consistency Reinforcement Techniques

Consistency reinforcement techniques, also known as filtering techniques, provide the CSP with as much information as possible in order to simplify the resolution of the CSP before and/or during the search for the solution. They reduce the domains of the variables and the relations associated with the constraints by removing values that cannot appear in any solution. They also allow for the early detection of failures. A CSP is k-consistent if and only if, for any n-uplet of k variables $(X_1, \ldots, X_k)$ of $\mathcal{X}$, any consistent $k-1$ instantiation may be extended to a consistent instantiation with the k-th variable. CSP does not necesarrily have a solution even if it is k-consistent [3]. A CSP is said to be strongly k-consistent if and only if $\forall i, 1 \leq i \leq k, \mathcal{P}$ is i-consistent.

A CSP is called arc consistent if and only if, for any couple variables (X_i, X_j) of $\mathcal{X}$, each couple represents an arc in the associated constraint graph, and for any value v_i from the domain D_i that satisfies the unary constraints involving X_i, there is a value v_j in the domain D_j compatible with v_i.

2.3 Parallel Constraint Solving

Parallel computing is a form of computation in which many calculations are carried out simultaneously[1] operating on the principle that large problems can often be divided into smaller ones, which are then solved in parallel[2]. Constraint programming can take advantage of parallelism due to the declararitve nature of CP. Propagation can often be done in parallel, which can lead to good speedups when the propagation consumes the most computation time. Parallel propagation can limit the scalability, however, due to the overhead from synchronisation. Search can also benefit from parallelisation. If the search space is partioned, each partition can then try operate in parallel.

Parallelising search in CP can be done by splitting data between solvers, e.g. create a decision point for a selected variable X_i so that one computer handles $X_i < \frac{min(X_i)+max(X_i)}{2}$ and another handles $X_i \geq \frac{min(X_i)+max(X_i)}{2}$ [11]. Hashing constraints allow each worker to efficiently skip portions of the search space

not assigned to its current problem[2]. When a worker moves from one node to another, every move from the root to the target node need to be recomputated, which we call the *recomputation overhead*. Since each subproblem must be easier to solve than the original problem, we will always get some recomputation overhead. Generating trivial subproblems might be paid by some *exploration overhead*, because many of the trivial subproblems whould have been discarded by the propagation mechnism of the sequential algorithm. When a constraint solver tries to find the best solution, a sequential algorithm will keep track of the best solution that has been found thus far. When using multiple workers we need to decide if we want some *synchronisation overhead* to share the best solution found so far, or if each worker keeps track of its own best solution. This leads to less loose coupling, but this might be paid by some exploration overhead.

2.3.1 Amdahl's Law

Amdahl's law predicts the maximum speedup that can be expected from a system as we increase the number of processors. The law assumes that a program is composed of a parallel part P and a sequential part S, such that $P + S = 1$. The expected speed up is then $1/(S + P/N)$, where N is the number of processors. As N tends to infinity Amdahl's law predicts that maximum speedup will be $1/S$, as the original P/N term tends to zero. As an example if we had $P = 0.99$, so 99% of our problem can be parallelised, 64 processors would run our program 39 times faster. For 128 processors the speedup is 56 times, for 1024 processors it is 91. As the number of processors continues to increase the speedup tends to 100. If $P = 0.9$, the law predicts a maximum speedup of 10, and if $P = 0.5$, the maximum speedup is 2, regardless of the number of processors available.

In the late 1980's Gustafson (1988) argued that Amdahl's law is overly pessimistic, as it assumes that as we increase the available parallel processors we continue to keep the workload fixed and hope for reduced runtime. That is, it is a 'fixed-size speedup' model and assumes N and P are independent; multi-processing is only used to improve response time. Gustafson assumes that problem and size also scales with the number of processors, i.e. as we get more processors we increase the problem and that run time, not problem size, is a constant. Gustafson observed that the parallel or vector parts of a program scales with problem size and the serial part does not (it diminishes proportionally). Consequently as we get more processors the workload grows and P increases resulting in an increase in speedup. This is the 'fixed-time speedup' model and an example is weather forecasting, where we use multi-processors to increase the quality of our results (the weather prediction) in a fixed amount of time (before the evening news). Perhaps this model is more appropriate for parallel constraint solving where we are always striving to solve larger and harder instances.

3 Related Work

Malapart, Régin, Rezgui (2017) looked at embarrasingly parallel search in constraint programming[2]. They used multiple different constraint solvers on a multitude of problems, and looked at the solve time and the speedup achieved. They found that Embarrasingly Parallel Search (EPS) often reaches linear speedups in the number of cores, whereas this never happens with worker stealing. EPS is as least as efficient as a method as other state-of-the-art algorithms on a number of problems using various computing infrastructures. It relies less on communication and synchronisation and mostly relies on the underlying sequential solver, making it easier to implement and debug. Finally, EPS is easy to adapt to specific algorithms or computing infrastructures due to its simplicity. Three execution environments were used: a Multi-core Dell computer with 256 GB of RAM and 4 Intel E7-4870 2.40 GHz processors running on `Scientific Linux 6.0`, a Data Center "Centre de Calcul Interactif", hosted by the "Université Nice Sophia Antipolis" which provides a cluster composed of 72 nodes, for a total of 1152 cores, running on `CentOS 6.3`, each node with 64 GB of RAM and 2 Intel E5-2670 2.60 GHz processors with 8 cores, and the third environment is the cloud platform Microsoft Azure, each node containing 56 GB of RAM and Intel Xeon E5-2690E 2.60 GHz with 8 physical cores[4]. The problems were run on with up to 512 workers. Some of the problems were run with the number of workers w equal to the number of cores c, such that no core is unused. On the multi core (Dell) computer, two workers for every physical core where used, since hyperthreading is used. In the paper they acknowledge that there is exploration overhead in solving the problems, but it is not further discussed since the main focus lies on the speedup gained. For some of the problems the sequential algorithm was more

performant than the parallel version due to overhead. Most of this overhead is recomputation overhead, and a lot of time was spent on context switches due to shared memory. The paper concludes that Message Passing Interface (MPI) is more suited for implementing EPS than Thread Technology, because MPI does not use shared memory.

Christopher Lecoutre, Lakdhar Saïs, Sébastien Tabary, and Vincent Vidal have show that restarted search in conjunction with recorded negative last decision (nld) nogoods eliminates the drawback that we can explore the same part of the search tree several times using a (geometric) restart strategy, if we record a set of nogoods at the end of each run [6]. For the tests the average CPU time and the number of timeouts were recorded, and they concluded that nogood recording from restarts significantly improves the robustness of the solver, since both the number of unsolved problem instances and the average CPU time are reduced. The same part of the search space is never explored twice, therefore in theory we should see less redundant exploration of the search tree in our own experiments when we do use nogoods from restarts.

Md Masbaul Alam Polash, M.A. Hakim Newton, and Abdul Sattar presented a constraint-based approach to solve the all-interval series problem, exhibiting new properties of the reformulated all-interval series which prune the search space significantly [9]. Emperical analysis shows that search may go deep down a branch with no solution, thus pruning techniques that can cut a branch with no solution at the start of that branch will be very effective.

4 Methods

For our tests we use the Gecode solver, which is provided with the MiniZincIDE version 2.5.5[1]. We perform our tests on challenges from the MiniZinc Challenge 2021[13]. We looked at several challenges that Gecode was able to solve within 20 minutes with parallel search. In total we have twelve optimisation problems and eight satisfaction problems, as shown in Table 1.

Kind	Problem	Input File
Satisfaction	Monomatch	`data_n_5_percentage_0.5`
Satisfaction	Pentominoes-Zayenz	`size_5_tiles_10_seed_17_strategy_close`
Satisfaction	Pentominoes-Zayenz	`size_10_tiles_10_seed_17_strategy_target`
Satisfaction	Pentominoes-Zayenz	`size_15_tiles_15_seed_17_strategy_source`
Satisfaction	Pentominoes-Zayenz	`size_20_tiles_10_seed_42_strategy_far`
Satisfaction	Pentominoes-Zayenz	`size_20_tiles_15_seed_4711_strategy_source`
Satisfaction	Steiner Systems	`t3_k4_N8`
Satisfaction	Steiner Systems	`t6_k6_N7`
Optimisation	ATSP	`instance5_0p15`
Optimisation	ATSP	`instance10_0p25`
Optimisation	Carpet Cutting	`mzn_rnd_test.01`
Optimisation	Carpet Cutting	`mzn_rnd_test.12`
Optimisation	Community Detection	`rnd_n100_e5000_s500_d300_c4_p50`
Optimisation	Flowshop Workers	`5stat_ex3`
Optimisation	Flowshop Workers	`5stat_ex6`
Optimisation	Flowshop Workers	`ex4`
Optimisation	Java Routing	`trip_6_2`
Optimisaiton	Java Routing	`trip_7_5`
Optimisation	Neighbours	`neightbours-new-19`
Optimisation	Peaceable Queens	$n = 8$

Table 1: List of the problems we used for the tests.

[1] `https://www.minizinc.org`

Gecode uses flatzinc files as its input, and some additional configurable variables. With help from the MiniZinc IDE we compiled the problems for a specific input file containing these configurable variables to a flatzinc file so it can then be executed by Gecode. Gecode uses a search strategy based on active failure count with a decay of 0.99 and geometric restarts, with a scale factor of 250 and base 1.5. It implements a tree-search parallelisation technique, with a recomputation distance of 8, a recomputation adaption distance of 2, and a nogoods limit of 128. When Gecode is run, it will print the solution(s) to an output file if it can be found, and some statistics of the solver and the problem, being the initialisation time, the solve time, the number of solutions found, the number of variables of the problem, the number of propagators, the number of propagations, the number of explored nodes, the number of failures encountered, the number of restarts encountered, and the peak depth of the search tree. Since we want to know how much overhead we get when we try to solve the problem with more than 1 thread, we are mainly interested in the number of explored nodes. Since Gecode does not deterministicly partitions the search space, each time Gecode is run with more than 1 thread we will not necessarily get the same result as we get on a previous run. Hence, we run each problem with the same parameters 5 times. We can then take the median of the results produced by Gecode to get a more accurate overview of the results. The reason we take the median rather than the mean, is to better handle outliers in the results. The *overhead* is then defined as the median of the extra number of explored nodes over five runs compared to the number of explored nodes with 1 thread. Positive values for the overhead indicate that we have an increase in the number of explored nodes, and therefore do redundant work. Negative numbers for the overhead indicate that we do less work compared to sequential search. We compare the overhead for 2, 4, 8, 16, and 24 threads. We run the solver for each problem, searching either for one/the best solution, or searching all solutions. Then we run the solver twice, once without restarts enabled, and once with nogoods from restarts enabled, with a geometric research strategy. This gives us a total of $6 \times 2 \times 2 \times 20 \times 5 = 2400$ output files to perform analysis on. For all problems we use the same time limit that was used in the MiniZinc Challenge, which is 20 minutes.

5 Results

All our experiments were run on the Peregrine Cluster of the University of Groningen. Each node in the cluster consists of an Intel Xeon E5-2680 v3 processor with 24 cores, 128 GB memory, running CentOS 7. [2]. Of the statistics we get in the output files produced by Gecode, we are mainly interested in the number of explored nodes in order to compute the overhead. Other interesting statistics we will take into account are the solve time (so we can see if a problem could be solved within the specified 20 minutes), the number of propagations, and the number of failures. When we use no-goods from restarts, we are also interested in the number of restarts. When we want to find all the solutions, we are also interested in the number of solutions found. The results have been divided into groups which show roughly the same trend in solve time, solutions, propagations, explored nodes, failures, and restarts. In the results we plot the *median* values of five runs without restarts against the median values of five runs with geometric restarts and using no-goods from restarts.

5.1 Satisfaction Problems

The satisfaction problems can be split up in two groups where we see roughly the same trend. Most satisfaction problems show a trend in solveTime, propagations, nodes, failures, and restarts, resembling the one shown in Figure 1. Here, solveTime is the median of the solve time over 5 runs, solutions is the median of the number of solutions found (which can be either 0 or 1), propagations is the median of the number of propagations, nodes is the median of the number of explored nodes in the search space, failures is the median of the number of failures encountered, and restarts is the median of the number of restarts (only applicable when we actually enable restarts). We plot the results for restart-enabled runs and runs without them enabled. The shaded region represents the minima and maxima.

[2] https://www.rug.nl/society-business/centre-for-information-technology/research/services/hpc/facilities/peregrine-hpc-cluster

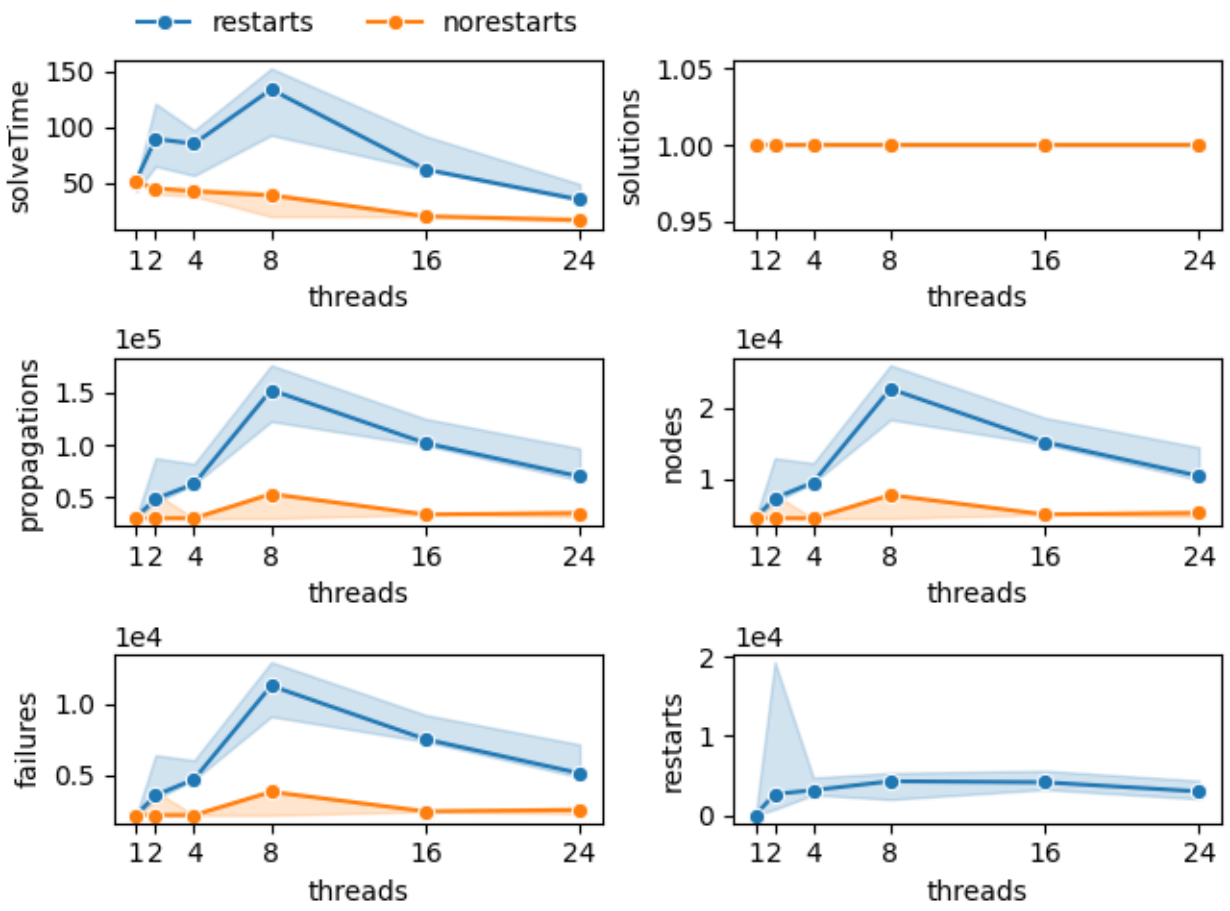

Figure 1: Trends for Pentominoes-Zayenz with `size_20_tiles_10_seed_42_strategy_far` as input. The general trend is that the amount of explored nodes is roughly the same at 1 thread for both the runs without restarts and the runs with restarts enabled. With more threads we explore more threads when we do use no-goods learning from restarts. We usually see a peak in the number of restarts at 2 threads.

When we use more than 1 thread, we explore more nodes when we use no-goods from restarts, compared to not using no-goods from restarts. The number of explored nodes increases up to 8 threads, after which it decreases again. Table 2 shows the percentage increase of the median of the explored nodes compared to the median of the runs at 1 thread, when we do not use restarts. Table 3 shows the percentage increase of the median of the explored nodes compared to the median at 1 thread, this time with no-goods from restarts enabled.

	Nodes	**% Increase in explored nodes**				
Threads	1	2	4	8	16	24
`size_15_tiles_15_seed_17_strategy_source`	181538	0.00	139.79	241.26	113.29	55.34
`size_20_tiles_10_seed_42_strategy_far`	4468	0.00	0.00	72.92	12.15	16.38
`size_20_tiles_15_seed_4711_strategy_source`	11925	0.00	0.00	0.00	0.00	−86.83
Steiner Systems (`t3_k4_N8`)	4149965	0.02	1.05	−2.98	−38.48	−1.70
Steiner Systems (`t6_k6_N7`)	127	0.00	0.00	0.00	0.00	0.00

Table 2: Percentage increase of explored nodes for satisfaction problems without using restarts.

For all the problems we see virtually no difference in number of explored nodes at 2 threads compared to

the number of explored nodes at 1 thread. The Steiner Systems problem with `t6_k6_N7` as input does not see an increase in explored nodes at all for any number of threads. The Pentonimioes-Zayenz problem with `size_20_tiles_15_seed_4711_strategy_source` as input also does not see an increase up to 16 threads. At 24 threads, we see a large decrease of 86.83%. At 4 threads, only the Pentominoes-Zayenz problem with `size_15_tiles_15_seed_17_strategy_source` as input sees a significant increase, where more than double the amount of nodes is explored compared to the runs with 1 thread. It sees another increase at 8 threads, as does the problem with `size_20_tiles_10_seed_42_strategy_far` as input. The Steiner Systems problem with `t3_k4_N8` as input on the other hand sees a small decrease at 8 threads. All three problems see a decrease in explored nodes compared to 8 threads, and either a further decrease or a small increase at 24 threads compared to 16 threads.

Figure 1 already showed that when no-goods from restarts is enabled we get a significant increase in number of nodes explored. A few of the problems also do not manage to get a solution for one or all of the runs before the time runs when we do not run with all threads, but all problems are solvable by Gecode when we only use 1 thread. The problems which could not be solved within 20 minutes are marked with a red star in Table 3. When we use no-goods from restarts we almost always explore more nodes than when we do not use restarts when we use more than one thread. The Steiner Systems problem with `t6_k6_N7` as input does not really seem to be affected that much by enabling restarts since we only see a small increase in explored nodes, and with 24 threads we even explore fewer nodes than when we run with 1 thread. The problem also explores very few nodes compared to the other problems at 1 thread, which can explain this difference. The Pentominoes-Zayenz problem with `size_20_tiles_15_seed_4711_strategy_source` as input also sees a large decrease in number of nodes explored at 24 threads, which is almost the same amount of decrease in number of explored nodes when we do not use restarts. The problem is not solvable with more than 1 or fewer than 16 threads however, so it is harder to draw conclusions from this problem.

For all the other problems we see a significant increase in explored nodes up to 8 threads. When using more than 16 threads we see a decrease in number of nodes explored relative to the number of explored nodes at 8 threads. The Pentominoes-Zayenz problems with input `size_15_tiles_15_seed_17_strategy_source` and `size_20_tiles_15_seed_4711_strategy_source` do not have a peak at 2 threads for amount of restarts, whereas the other problems do show a peak in the amount of restarts at 2 or 4 threads.

	Nodes	**% Increase in explored nodes**				
Threads	1	2	4	8	16	24
`size_15_tiles_15_seed_17_strategy_source`	181685	37.29*	212.31*	513.71*	389.84	316.83
`size_20_tiles_10_seed_42_strategy_far`	4503	61.71	110.82	402.27	236.95	130.98
`size_20_tiles_15_seed_4711_strategy_source`	11962	15.26*	45.45*	174.85*	69.40*	−86.38
Steiner Systems (`t3_k4_N8`)	4150165	130.81	205.61	257.13	62.33	30.05
Steiner Systems (`t6_k6_N7`)	136	1.47	6.62	2.94	7.35	−19.85
* One or more runs did not complete within 20 minutes						

Table 3: Percentage increase of explored nodes for satisfaction problems with no-goods from restarts enabled.

5.1.1 Monomatch problem and other outliers

Both the Monomatch problem and Pentominoes-Zayenz with `size_10_tiles_10_seed_17_strategy_target` as input follow the trend as seen in Figure 2, where the amount of propagations, explored nodes, and failures decreases when we use more threads. We also see a small peak in the amount of restarts at two threads, with some significant outliers at four threads.

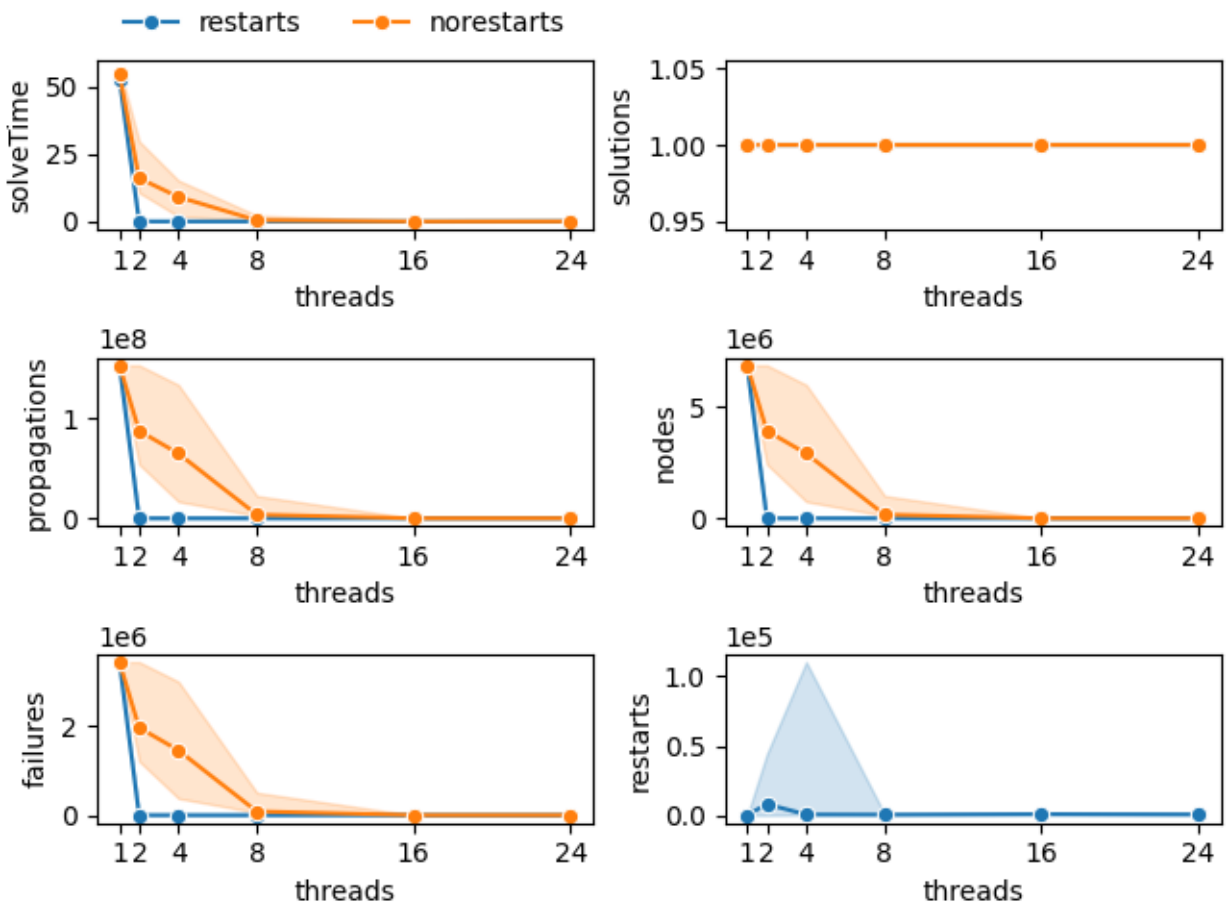

Figure 2: Trends for Monomatch. The Pentominoes-Zayenz problem with `size_10_tiles_10_seed_17_strategy_target` follows the same trend. We explore fewer nodes with more threads, and with restarts enabled we get fewer explored nodes than without. The solvetime, number of propagations, and failures follow the same trend. After 8 threads we can observe a minimal increase in number of explored nodes ($< 0.1\%$). At 2 threads we get a small peak in amount of restarts, and a large variation in amount of restarts at 4 threads.

This is roughly the trend one would expect; more threads means we find a suitable solution earlier so we need to explore fewer nodes. Table 4 and Table 5 show the percentage increase of the median of the explored nodes compared to the runs with a different amount of threads, without restarts and with no-goods from restarts, respectively. In Table 5 we can see the percentage increase when we do use no-goods from restarts.

	Nodes	**% Increase in explored nodes**				
Threads	1	2	4	8	16	24
Monomatch (`data_n_5_percentage_0.5`)	6887672	−42.85	−57.64	−97.61	−99.98	−99.97
`size_10_tiles_10_seed_17_strategy_target`	78116	−82.51	−99.21	−99.73	−99.55	−99.56

Table 4: Percentage increase of explored nodes for monomatch-like problems without using restarts.

We see some minimal increase in the amount of nodes explored at 24 threads for the monomatch problem and at 16 threads for the pentominoes problem. This suggests some slight overhead, Table 5 shows the percentage increase with restarts enabled.

	Nodes	% Increase in explored nodes				
Threads	1	2	4	8	16	24
Monomatch (`data_n_5_percentage_0.5`)	6888461	−99.995	−99.992	−99.988	−99.984	−99.978
`size_10_tiles_10_seed_17_strategy_target`	78191	−47.45	−94.19	−99.71	−99.48	−99.48

Table 5: Percentage increase of explored nodes for monomatch-like problems with no-goods from restarts enabed.

When we use no-goods from restarts, We also see a reduction in the amount of explored nodes at 2 threads, followed by an increase in explored nodes. Do note that after 2 threads we are only left with $\sim 0.005\%$ of the original amount of explored nodes. The Pentominoes-Zayenz problem keeps decreasing up until we use 8 threads. With 16 threads we see an increase, followed by a very small increase at 24 threads. Similar to the monomatch problem, we do not have many explored nodes relative to running with 1 thread, but we do see some overhead when using more threads as long as the amount of threads is large enough.

Pentominoes-Zayenz with `size_5_tiles_10_seed_17_strategy_close` as input does not see any difference between the results of the runs with restarts and the runs without restarts outside of some outliers. The problem is solved very quickly, within 0.05 seconds. The amount of explored nodes stays constant, whether we use 1 thread or more. On almost all runs runs the number of explored nodes count is 21. We only get 2 outliers at 2 threads and 4 threads for runs with restarts and without, respectively. The failure rate is 0 for all runs. This has the effect that we do not get any restarts. The outliers can be attributed to the non-deterministic nature Gecode's solver partitiones the search space when we use multithreading.

5.2 Optimisation Problems

The optimisation problems can be divided into a two groups. All the Flowshop Workers do not finish within the 20 minutes, and do not get any solutions. The other problems are solvable, but the amount of solutions found by Gecode differs per run, and per number of threads used. Figure 3 shows the trend for the Peaceable Queens problem. Most of the other optimisation problems follow the same general trend where we explore many more nodes of the search space when we use no-goods from restarts compared to when we do not use restarts. For some of the problems we see a slight upward trend in explored nodes when we use more threads, for some other problems we see a slight decreasing trend. When we use no-goods from restarts, we always see a peak at 2 threads in the amount of restarts. Another interesting observation is that the number of solutions Gecode reports is greater than 1. The number of solutions that gets reported likely includes intermediate results. The final solution reported is the same for all runs which finish within 20 minutes.

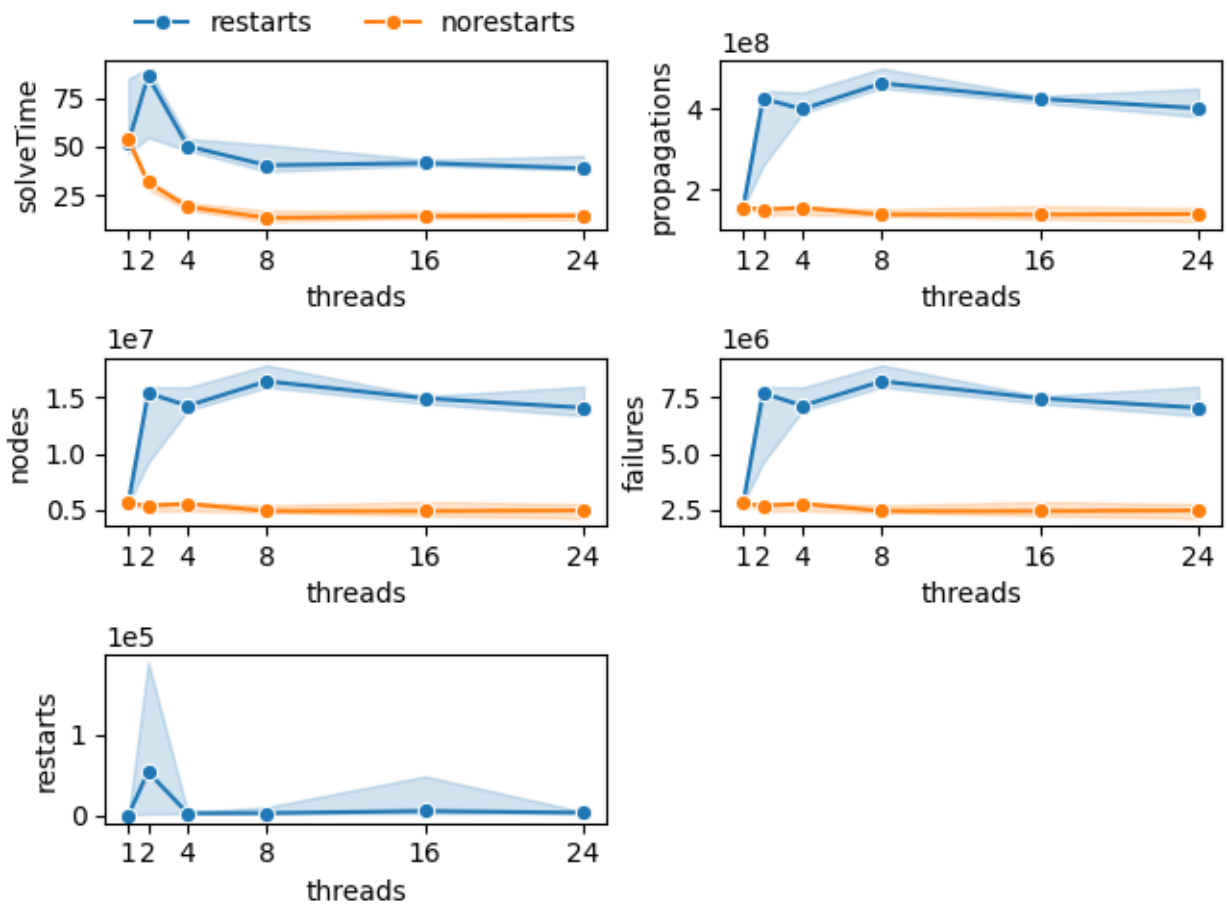

Figure 3: Trends for Peaceable Queens with $n = 8$ as input. Most of the satisfaction problems follow the same trend. The solve time goes down with more threads, but we do see a peak at 2 threads when we use restarts, which corresponds with a peak in the amount of restarts at 2 threads. Not all problems could be solved at 1 or 2 threads. The number of propagations, explored nodes, and failures follow roughly the same trend for all problems. We always explore a lot more nodes with restarts enabled.

As expected, the solve time goes down when we use more threads, although we see a peak at two threads when we do use restarts before the solve time goes down. This coincides with the peak at 2 threads when we use no-goods from restarts. The amount of solutions Gecode manages to find fluctuates between runs, but in general we get more solutions with more threads, however some problems see a tiny decrease in amount of solutions found on some of the runs. The reason we get a different amount of results for different runs can be attributed to the non-deterministic way Gecode partitions the search space. Since we are trying to optimise for a certain object value, some solutions are likely pruned in one iteration, and are not pruned in the next iteration, causing a different number of solutions found per iteration.

Table 7 shows the percentage increase of the median of the explored nodes compared to the median at the previous number of threads, when we do not use restarts. Some problems could not finish within the timelimit for a specific number of threads for a few or all runs. These are marked with a $*$.

	Nodes	% Increase in explored nodes				
Threads	1	2	4	8	16	24
ATSP (`instance5_0p15`)	69	0.00	0.00	0.00	0.00	0.00
Carpet Cutting (`mzn_rnd_test.01`)	4195586∗	−22.90	−64.21	−64.15	−64.04	−63.79
Carpet Cutting (`mzn_rnd_test.12`)	6766354∗	64.91∗	24.78	24.79	25.21	30.21
Java Routing (`trip_6_2`)	569369	11.16	21.60	83.85	110.83	104.24
Java Routing (`trip_7_5`)	11506991	−3.02	−2.98	−1.16	6.15	10.45
Neighbours (`neightbours-new-19`)	49613	0.00	0.00	0.02	0.05	0.05
Peaceable Queens ($n = 8$)	5586667	−3.77	−0.52	−12.21	−12.31	−11.36
∗ One or more runs did not complete within 20 minutes						

Table 6: Percentage increase of explored nodes for optimisation problems without using restarts.

Both the ATSP problem with `instance5_0p15` as input and the Neighbours problem with `neightbours-new-19` as input do not see a significant increase or decrease in explored nodes when we use more threads. The Peaceable Queens problem with $n = 8$ sees minor decreases and increases in explored nodes, and a significant decrease at 8 threads (close to 12% decrease). The Carpet Cutting problems do not finish with a single thread, and `test.12` also does not finish with 2 threads. After 8 threads we see a slight increase in amount of explored nodes when we use more threads. The Java Routing problem with `trip_6_2` sees a significant increase in amount of nodes explored when we use more threads. Especially at 8 threads we get more than a 50% increase. With `trip_7_5` we observe a slight upward trend in amount of nodes explored. Table 6 shows the percentage increase of the median of the explored nodes compared to the median at the previous number of threads, when we use no-goods from restarts. Some problems could not finish within the timelimit for a specific number of threads for a few or all runs. These are marked with a ∗.

	Nodes	% Increase in explored nodes				
Threads	1	2	4	8	16	24
ATSP (`instance5_0p15`)	68	2.94	45.59	55.88	64.71	72.06
Carpet Cutting (`mzn_rnd_test.01`)	4239060∗	32.28	18.84	8.67	22.47	18.24
Carpet Cutting (`mzn_rnd_test.12`)	6710359∗	76.72∗	279.11	358.01	94.75	265.77
Java Routing (`trip_6_2`)	569643	337.19	438.14	552.94	574.51	601.20
Java Routing (`trip_7_5`)	11507417	226.26∗	273.82	248.84	188.47	164.58
Neighbours (`neightbours-new-19`)	49660	222.26	205.25	245.95	199.85	275.99
Peaceable Queens ($n = 8$)	5587961	175.19	154.81	193.87	166.77	151.80
∗ One or more runs did not complete within 20 minutes						

Table 7: Percentage increase of explored nodes for optimisation problems with no-goods from restarts enabled.

The ATSP problem with `instance5_0p15` sees a mild increase in number of explored nodes at 2 threads, a significant increase at 4 threads, and a slight increasing trend for more threads. The Carpet Cutting problems do not finish for 1 thread, and `test.12` does not finish with 2 threads. The Java Routing problem with `trip_7_5` as input did finish at 1 thread, but did not finish at 2 threads. We see more than a 200% increase in explored nodes. After 4 threads Gecode is able to solve the problem again, but also explores more than double, and sometimes even tripple the amount of nodes that were explored at 1 thread.

5.2.1 Flowshop Workers Problems

None of the Flowshop Workers problems finish within 20 minutes, and no solutions are found. The amount of propagations, explored nodes, and failures keep increasing the more threads we use, as can be seen in Figure 4. This is not very surprising, since Gecode needs to explore more nodes in an attempt to find a solution. All three Flowshop Workers problems follow the same trend.

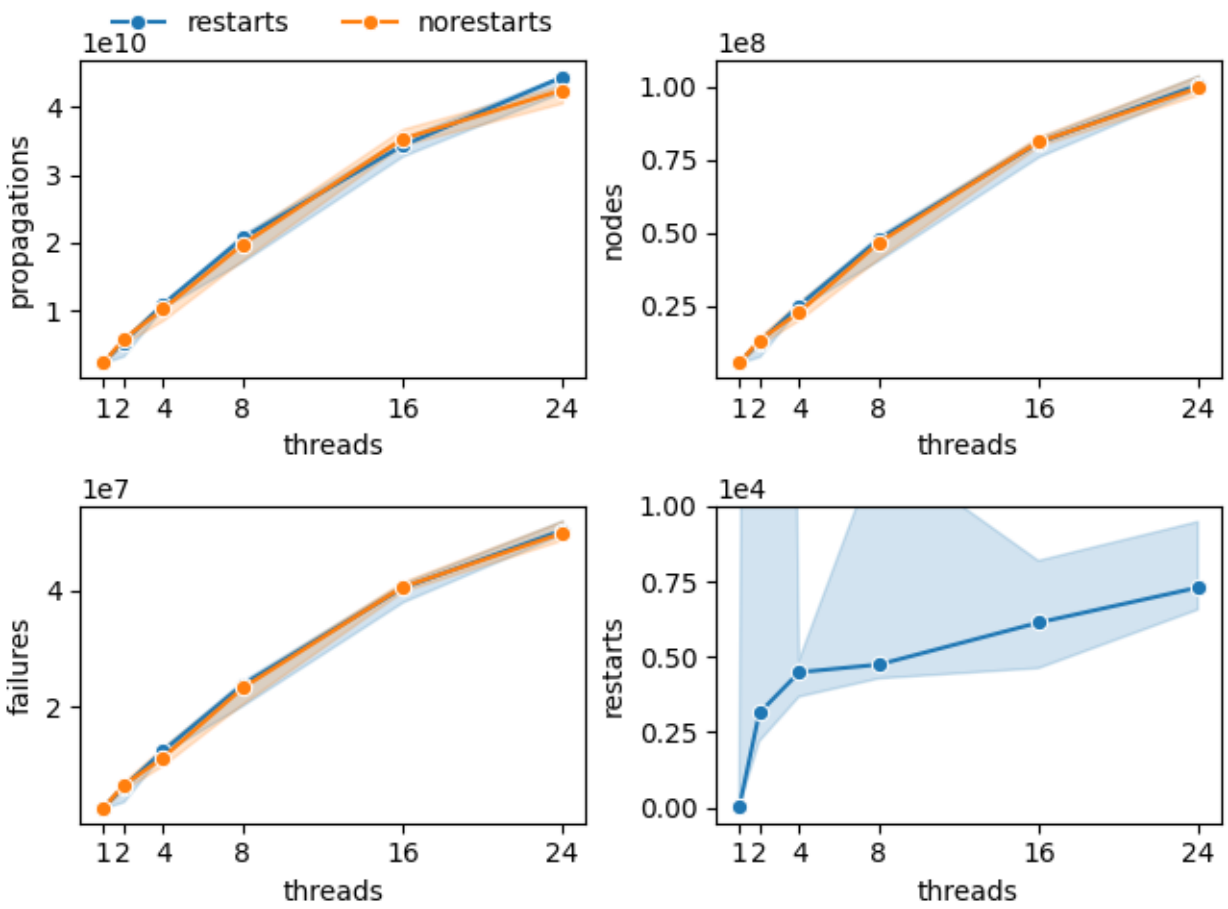

Figure 4: Trends for Flowshop Workers with `5stat_ex3` as input. These problems do not give a solution within the time limit. We see a continuously increasing amount of explored nodes. The other Flowshop Workers problems see the same trend, although the problem with `ex4` as input sees a slight decrease after 16 threads. There is not much of a difference between runs without restarts and runs with restarts enabled. The amount of restarts also keeps increasing the more threads we use, and we see some variation between runs, which does not appear to impact the amount of explored nodes.

Table 8 and Table 9 show the percentage increase in explored nodes. We see a fewer percentage increase in number of explored nodes when we do use no-goods from restarts. Overall there is not too great of a difference between the runs without restarts and those with no-goods from restarts.

	Nodes	**% Increase in explored nodes**				
Threads	1	2	4	8	16	24
Flowshop Workers (`5stat_ex3`)	5410395	143.73	321.32	762.81	1399.18	1740.46
Flowshop Workers (`5stat_ex6`)	4732170	108.59	374.73	736.03	1443.35	1631.15
Flowshop Workers (`ex4`)	1356641	106.36	197.45	469.95	759.75	651.05

Table 8: Percentage increase of explored nodes for Flowshop Workers problems without using restarts.

	Nodes	% Increase in explored nodes				
Threads	1	2	4	8	16	24
Flowshop Workers (`5stat_ex3`)	5612603	124.51	346.02	751.48	1339.80	1691.47
Flowshop Workers (`5stat_ex6`)	5773960	69.21	242.45	540.16	1022.23	1247.04
Flowshop Workers (`ex4`)	1213965	43.06	304.67	532.78	808.15	729.32

Table 9: Percentage increase of explored nodes for Flowshop Workers problems with no-goods from restarts enabled.

5.2.2 ATSP `instance10_0p25`

ATSP with `instance10_0p25` as input produces results which are a mix between the regular optimisation problems and the Flowshop Worker Problems. It does not finish within the 20 minutes when using fewer than 8 threads, as can be seen in Figure 5. The red line depicts the time limit of 20 minutes. Without restarts enabled we always hit the time limit. With using no-goods from restarts enabled, we finish with 8 threads for only 1 run. With 16 and 24 threads we always manage to complete within the time limit.

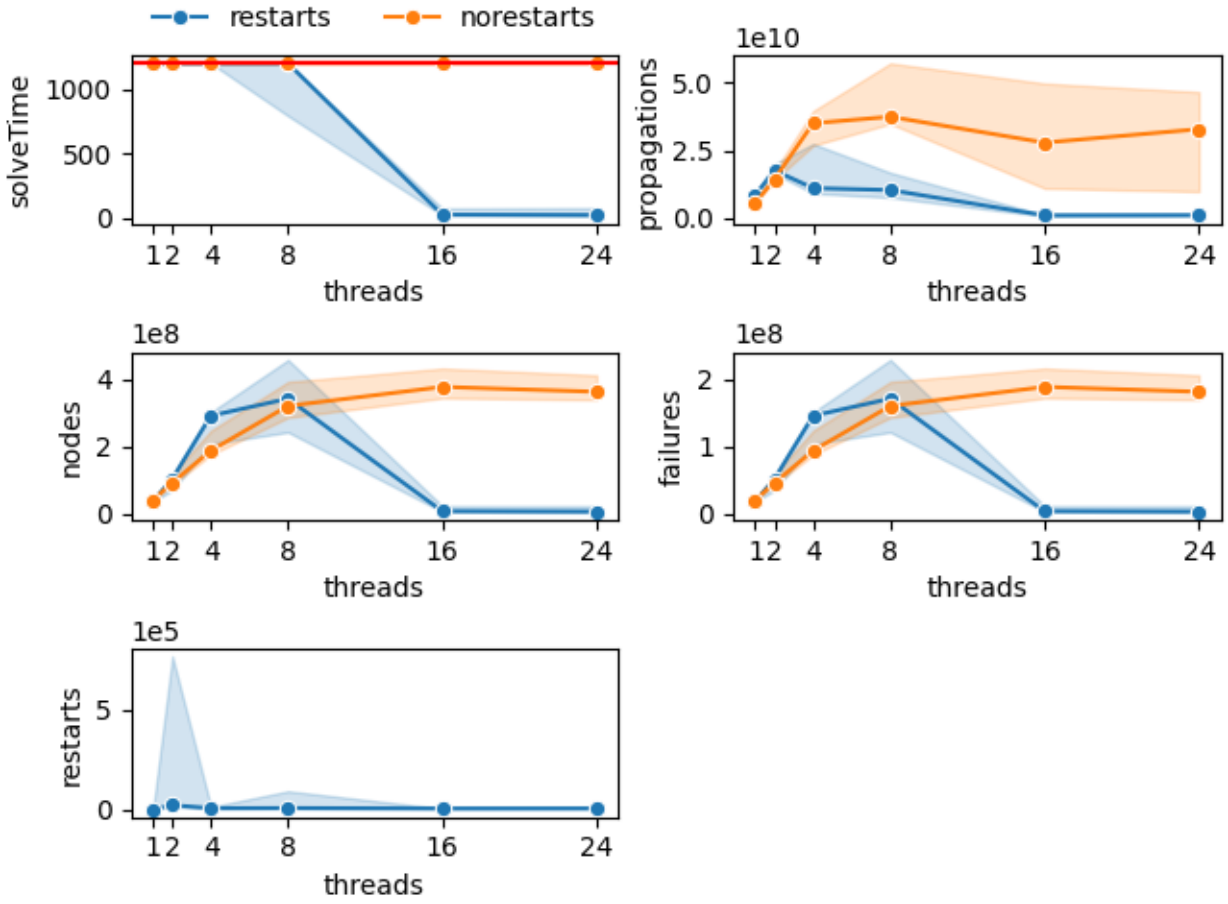

Figure 5: Trends for ATSP with `instance10_0p25` as input. The Problem does not finish within 20 minutes when not using no-goods from restarts. Without restarts, we see the same increase in nodes that can be seen for the Flowshop Workers problems. With restarts enabled, we see the same trend up to 8 threads. Then the problem finishes, and we see a decrease of roughly 97% going from 8 threads to 16 threads. On one of the runs we solve the problem within the timelimit with 8 threads, taking a little under 14 minutes. For the remaining number of threads we solve the problem within a minute.

The number of explored nodes when we do not use restarts follows roughly the same increasing trend as the trend from Figure 4. When we use no-goods from restarts we see a decreasing trend after 8 threads, similar to the one seen for the Java Routing problem with input `trip_7_5`. In Table 10 we see the percentage increase of the median of the explored nodes with each thread. Since we do not finish within 20 minutes with fewer than 8 threads, we cannot really draw conclusions from this particular instance. After 16 threads the decrease stabilises, with only a 2% further decrease instead of a 97% before. The one thing of note is that we see a spike in amount of restarts at 2 threads, just like with all other problems.

ATSP (`instance10_0p25`)	**Nodes**	**% Increase in explored nodes**				
Threads	1	2	4	8	16	24
Without restarts	41901972∗	121.53∗	349.07∗	664.45∗	798.83∗	764.52∗
With restarts	38312890∗	117.45∗	660.09∗	791.94∗	−77.07	−79.70
∗ One or more runs did not complete within 20 minutes						

Table 10: Percentage increase of explored nodes for the ATSP problem with `instance10_0p25` as input.

5.2.3 Community Detection problem

Community Detection with `rnd_n100_e5000_s500_d300_c4_p50` as input is a problem with some interesting results. It has zero propagators, and explores only one node. We always get 38048 propagations. For all parameters we change (one/all solutions, no-goods with restarts or no restarts, different threads), the only value that is not constant over the runs is the solve time. There is likely something wrong with either the challenge itself, or with the conversion to flatzinc.

5.3 Satisfaction Problems searching all solutions.

We also tried to run the problems searching all solutions instead of just one. For the problems that were classified as optimisation problems, we get almost the same results as when searching for one solution. We only see minor differences in amount of explored nodes. E.g. for the Peaceable Queens problem with restarts enabled, when we seek all solutions at 2 threads we get an 178.75% increase in explored nodes, whereas when we only seek one solution we get an 175.19% increase in explored nodes. The final solution found is not necessarily the same for all threads. When running the satisfaction problems trying to find all solutions we get substantially different results compared to searching only one solution. Gecode prints all solutions that could be found in addition to the statistics, unlike the optimisation problems where it only prints the final solution and the statistics. This unfortunately results in huge filesizes, averaging around 5GB per result file. Some result files exceed 10 GB, with the biggest being 13.8 GB. When we run the problems without restarts, we always get unique results. However, if we use no-goods from restarts we get duplicate (i.e. non-unique) results. This suggests something goes wrong with how Gecode partitiones the search space and how restarts are implemented.

Unfortunately, only four problems give somewhat meaningful results. Three of these have the same general trend. This are the Pentominoes-Zayenz problems with input `size_10_tiles_10_seed_17_strategy_target` and `size_20_tiles_10_seed_4711_strategy_source`, and the Steiner Systems problem with `t3_k4_N8` as input. The general trend cannot be easily put in a comparison plot like we did previously, since the results differ too much. The general trend without restarts can be seen in Figure 6, and the trend with restarts can be seen in Figure 7.

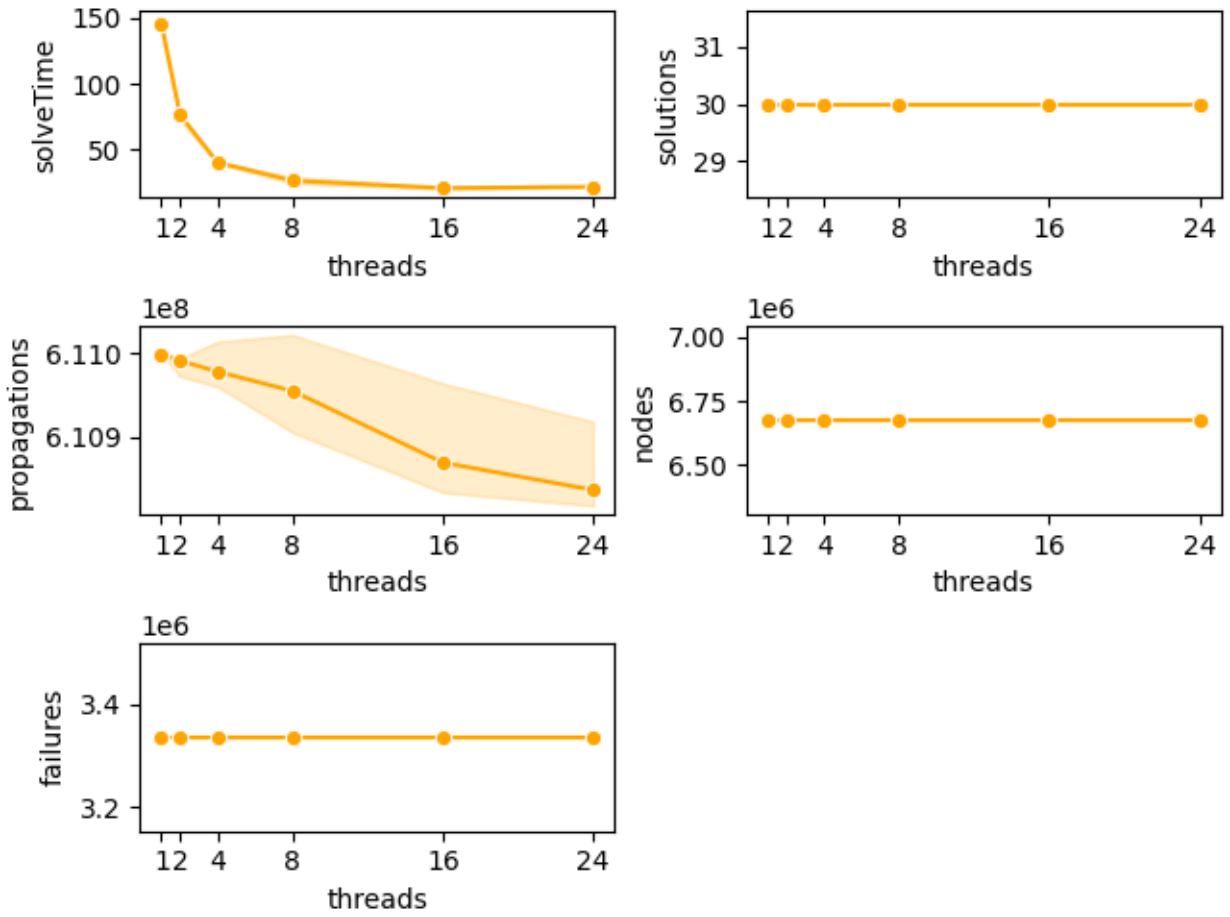

Figure 6: Trend for Steiner Systems with `t3_k4_N8` as input, without restarts.
The solve time and propagations decrease when we use more threads, whereas the solutions found, the explored nodes and the number of failures stay constant.

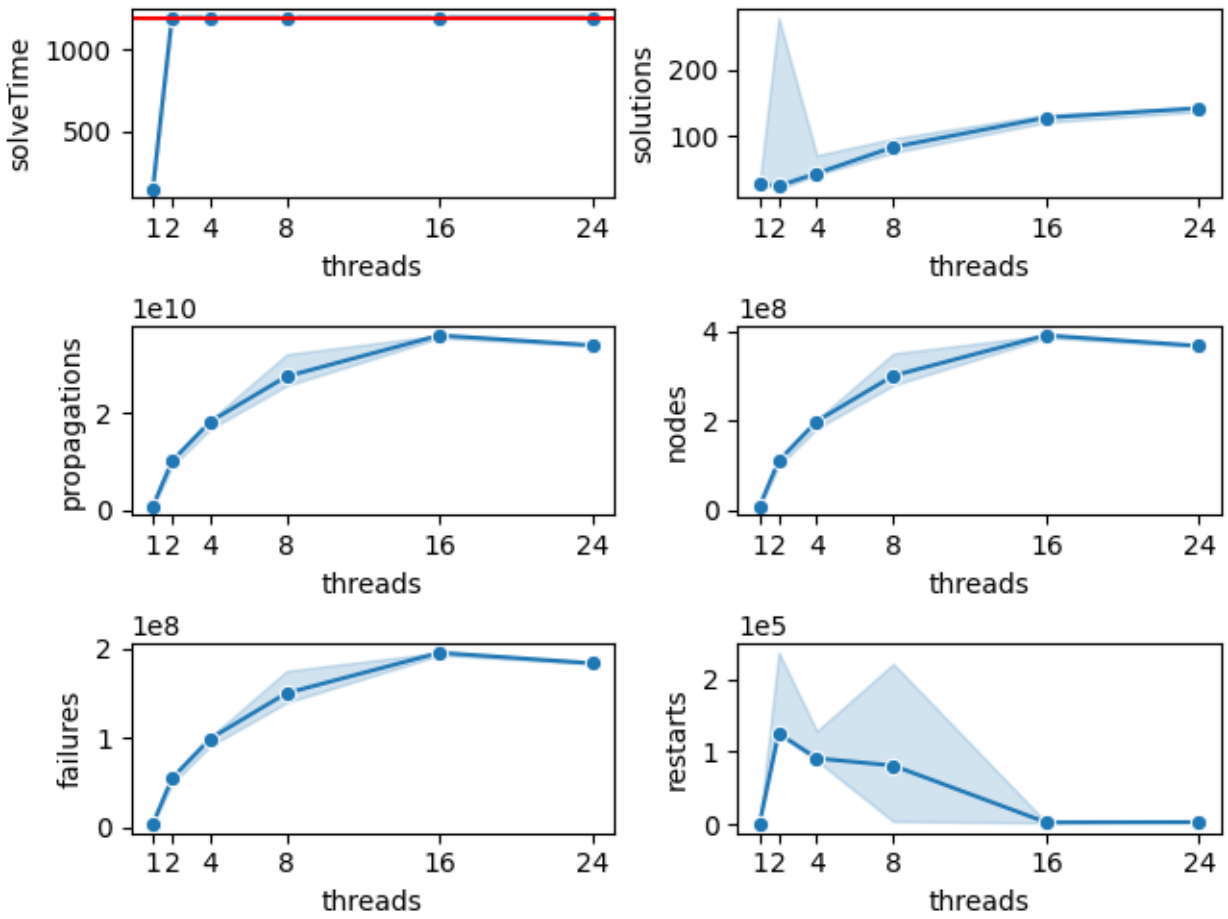

Figure 7: Trend for Steiner Systems with `t3_k4_N8` as input searching all solutions, with restarts. We do not finish within 20 minutes when we use more than 1 thread, and the amount of solutions, propagations, explored nodes, and failures keep increasing with more threads. The amount stabilise after 24 threads. At 2 threads we see a significant increase in restarts.

Without restarts enabled, we see a zero percent increase in the number of explored nodes for any number of threads. We also get a fixed number of (unique) solutions. We can therefore conclude that we do no not get overhead when using more threads if we search for all solutions. When no-goods from restarts are enabled, we always reach the time limit if we use more than 1 thread. The propagations, number of explored nodes, and number of failures keep increasing with more threads. The number of solutions found also continues to increase when we add more threads, and in the output files we can see that they contain duplicate solutions. Most problems have a few unique solutions but the majority consists of duplicate solutions. The amount of restarts shows a peak at 2 threads. The statistics for the number of explored nodes for the runs with no-goods from restarts are shown in Table 11.

	Nodes	**% Increase in explored nodes**				
Threads	1	2	4	8	16	24
`size_10_tiles_10_seed_17_strategy_target`	777526	401.14	82.25	997.80	809.89	616.97
`size_20_tiles_10_seed_42_strategy_far`	41271	322.88	267.25	485.96	629.08	834.39
Steiner Systems (`t3_k4_N8`)	6674940	1534.11	2856.86	4389.49	5744.01	5391.43

Table 11: Percentage increase of explored nodes for the satisfaction problems with no-goods from restarts enabled when searching for all solutions.

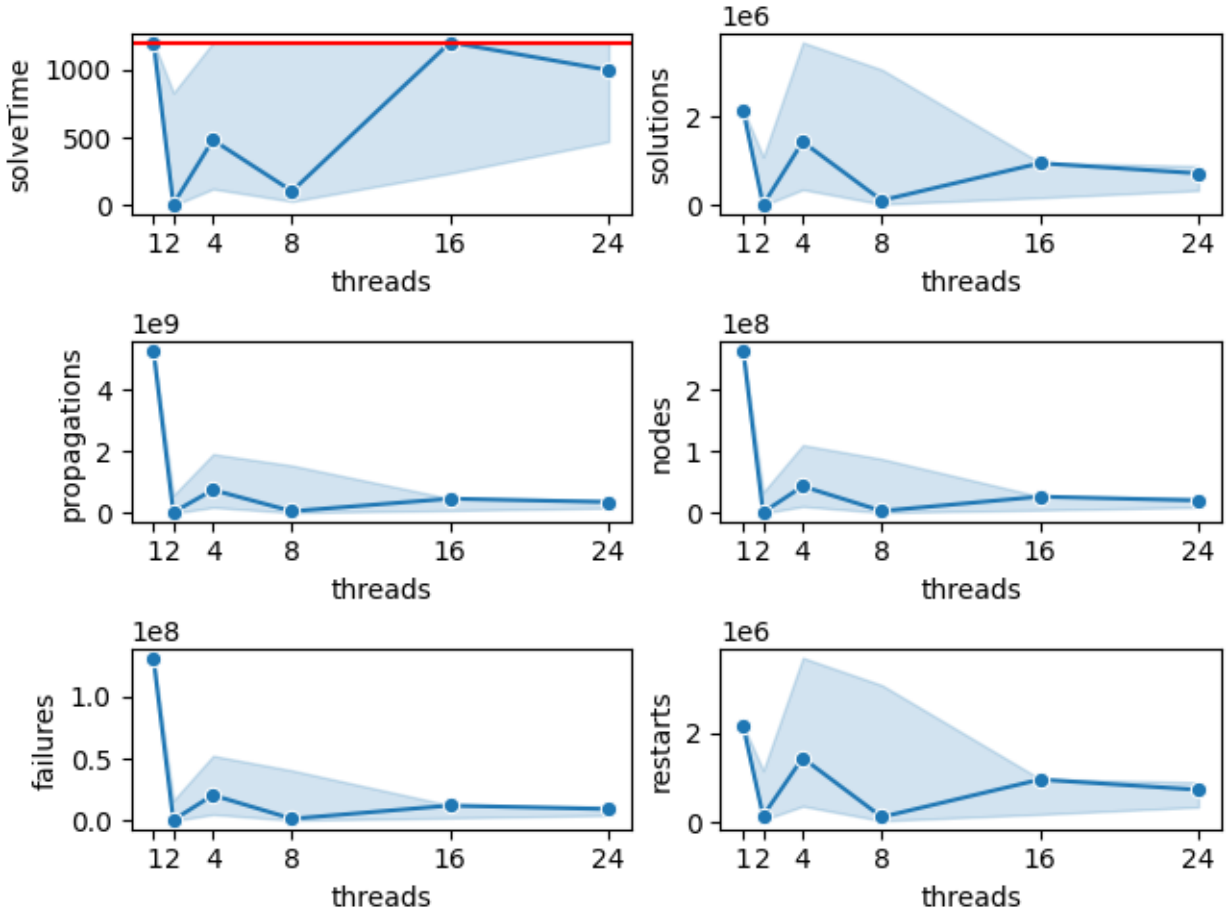

Figure 8: Trend for Steiner Systems with `t6_k6_N7` as input, searcing all solutions with no-goods learning from restarts. We always get a timeout except at 2 threads. We explore a massive amount of nodes at 1 thread, exceeding 260 million. With more threads we explore only a fraction of that number, for some runs we only explore 75. The number of solutions (which contain duplicates), the number of propagations, the number of explored nodes, the number of failures, and the number of restarts all follow the same general trend.

The general trend is that we explore more nodes when we use more threads, which is not surprising. More threads allow us to find more solutions before we hit the time limit. In general we explore the same number of nodes at 1 thread with and without restarts.

5.3.1 Steiner Systems `t6_k6_N7`

When the Steiner System problem with `t6_k6_N7` as input is run without restarts, the number of solutions, propagations, explored nodes, and failures stay constant for any number of threads, as with the previously discussed problems. The solve time however slightly increases with more threads used. This can be explained because the solve time is incredibly quick (less than 0.05 seconds), so the extra milliseconds can be attributed to the communication between the threads signalling that the solutions have been found. With restarts enabled we get a completely different trend for the statistics, as seen in Figure 8.

In Table 12 we find the percentage increase in explored nodes for the problem. The amount of explored nodes stays relatively stable after 2 threads, but we do see some increase and decrease. Just like the other satisfaction problems, we do not see an increase or decrease in explored nodes when we do not use restarts.

Steiner Systems (`t6_k6_N7`)	**Nodes**	**% Increase in explored nodes**				
Threads	1	2	4	8	16	24
Without restarts	127	0.00	0.00	0.00	0.00	0.00
With restarts	263364748	−99.63	−83.52	−98.77	−89.92	−92.28

Table 12: Percentage increase of explored nodes for the Steiner Systems problem with `t6_k6_N7` as input, when searching for all solutions.

5.3.2 Monomatch problem

When seeking all solutions for the mono-matching game, we get a timeout, regardless of the number of threads used, and regardless whether we are using restarts or not. The red line seen in Figure 9 marks the limit of 1200 seconds (20 minutes). Interestingly, we get the most solutions at 8 threads. All these solutions are unique. We see a decrease in solutions found with more than 16 threads, but this can be attributed to a higher failure rate, and a higher node exploration rate. The amount of explored nodes continues to increase with more threads when we do not use restarts, but decreases after 4 threads when we do use no-goods from restarts.

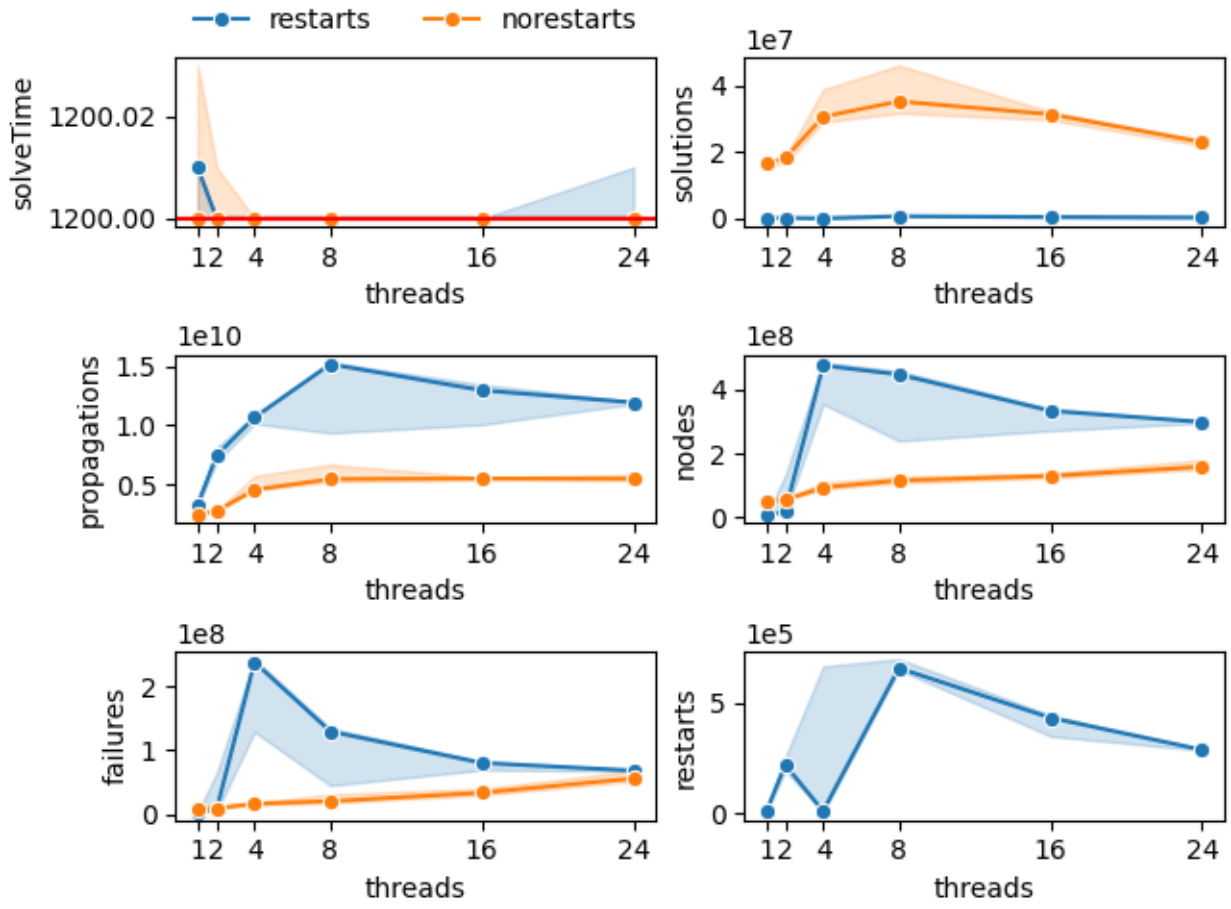

Figure 9: Trend for monomatch searching all solutions.
We always get a timeout for any number of threads used, regardless whether restarts are enabled or not. We always get more solutions without restarts, and explore more nodes with restarts. We see a small peak at 2 threads for the amount of restarts, and a larger increase at 8 threads.

If we enable no-goods from restarts, we can observe roughly the same trends for the solutions, propagations, and nodes as without using restart. Although the trends are the same, the values are a lot higher for

the amount of propagations, explored nodes, and failures. Fewer solutions are found, however, and not all of them are unique. For example, at 8 threads for a specific run we get $65,5049$ solutions. Not only is that an order of magnitude fewer than the average of roughly $36,000,000$ we found when we did not use restarts for the same number of threads, many of these solutions are duplicates. The failure rate is also substantially higher.

Table 13 shows the percentage increase of the median of the number of explored nodes.

Monomatch (`data_n_5_percentage_0.5`)	**Nodes**	**% Increase in explored nodes**				
Threads	1	2	4	8	16	24
Without restarts	50833760	9.29	85.49	128.37	156.10	212.78
With restarts	7263010	163.56	6451.30	6064.07	4497.38	4018.46

Table 13: Percentage increase of explored nodes for the Monomatch problem when searching for all solutions.

Like we saw earlier for problems that do not finish within the time limit, we get an increasing amount of explored nodes. When we do use restarts, we get a slight decrease again after 24 threads, but we still see an overall increase of more than 4000% explored nodes compared to 1 thread.

5.3.3 Other Pentominoes-Zayenz problems

A few of the Pentominoes problems all have a different trend, except that the problem does not finish within the time limit. When seeking all the solutions for the Pentominoes problem with `size_5_tiles_20_seed_17_strategy_close` as input, we get roughly the opposite trend as we get for the monomatch problem for the number of explored nodes, as can be seen in Figure 10. The amount of solutions we get follows the same trend as with the monomatch problem, where we get more solutions without restarts, and where we get fewer and duplicate results when we do enable restarts.

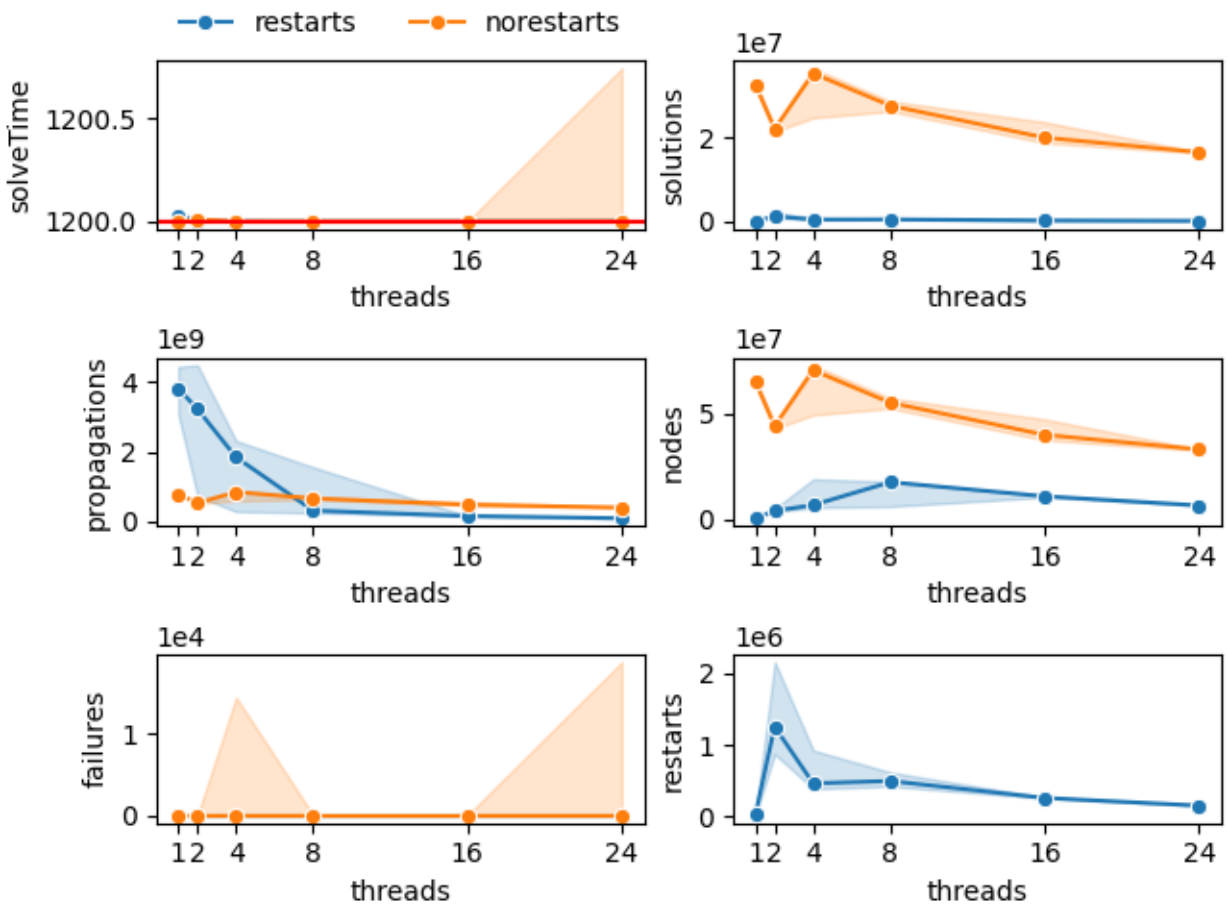

Figure 10: Trends for Pentominoes-Zayenz with `size_5_tiles_20_seed_17_strategy_close` as input. We get a time limit for all the runs, regardless of the amount of threads used. Similar to the monomatch problem, we find more solutions when we do not use restarts, but unlike the monomatch problem we also explore more nodes. We also get no failures, except for one run at 4 threads, and another run at 24 threads. The propagations do not follow the same trend as the number of explored nodes when we do use no-goods from restarts. At 2 threads we see a peak in the number of restarts.

The Pentominoes problem with `size_15_tiles_15_seed_17_strategy_source` as input sees a trend as depicted in Figure 11. We get an almost linear increase in the amount of explored nodes if we use more threads. We get between 2 and 8 solutions, whether we use restarts or not, except at 2 threads. When we do use no-goods from restarts we get up to 1635 solutions.

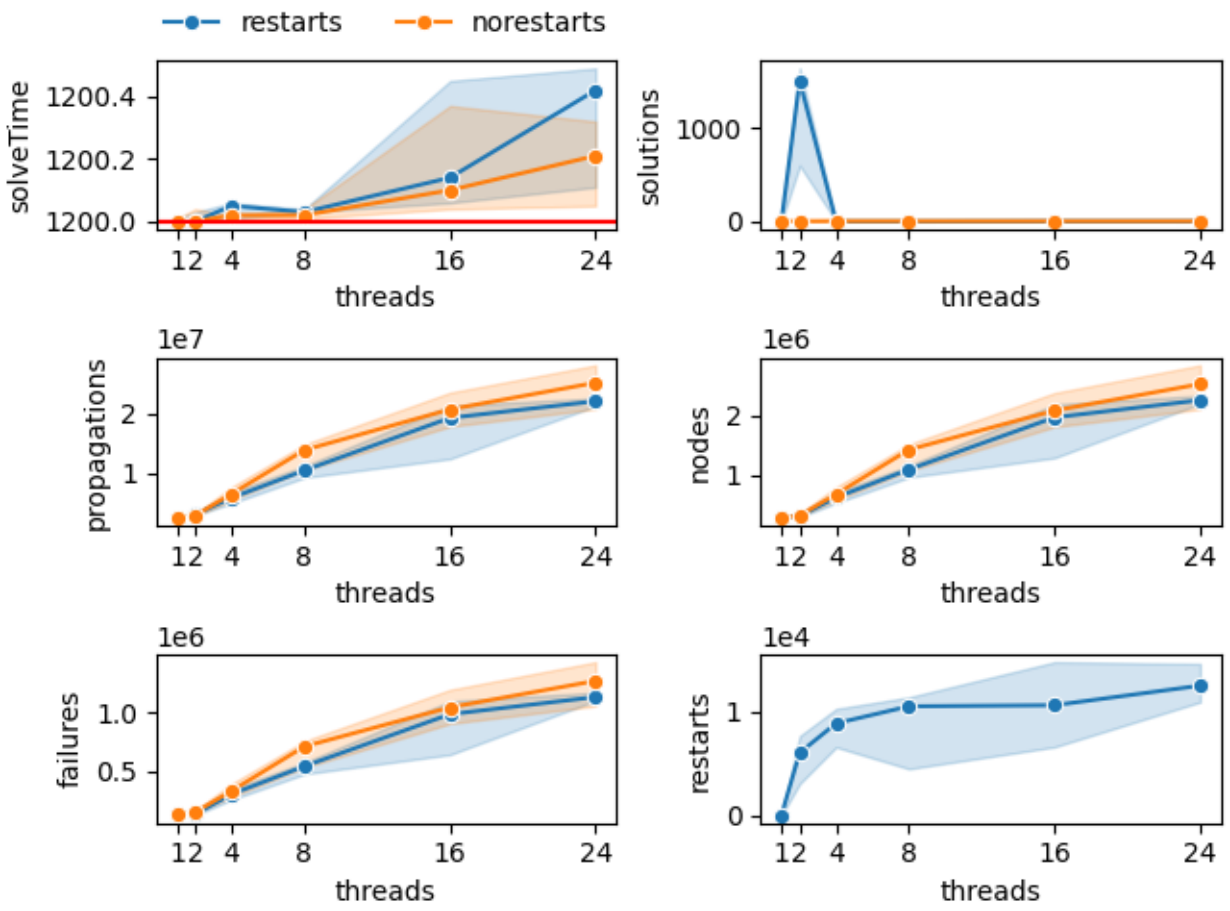

Figure 11: Trends for Pentominoes-Zayenz with `size_15_tiles_15_seed_17_strategy_source` as input. We always reach the time limit, for any number of threads. The number of solutions, propagations, nodes, failures, and restarts keep increasing the more threads we use. Without restarts we only get between 2 and 8 solutions, with restarts we get between 0 and 2 for most runs, except at 2 threads. Here we get more than 1500 solutions for most runs, although not all solutions are unique. We do not see a peak at 2 threads for the number of restarts.

The Pentominoes problem with input `size_20_tiles_15_seed_4711_strategy_source` only finishes within the time limit when restarts are not enabled, and we use more than 8 threads. The number of explored nodes goes up the more threads we use, but the increase is way smaller compared to the increase we see for the monomatch problem, for example. The number of propagations and failures follows the same trend as the number of explored nodes, as can be seen in Figure 12.

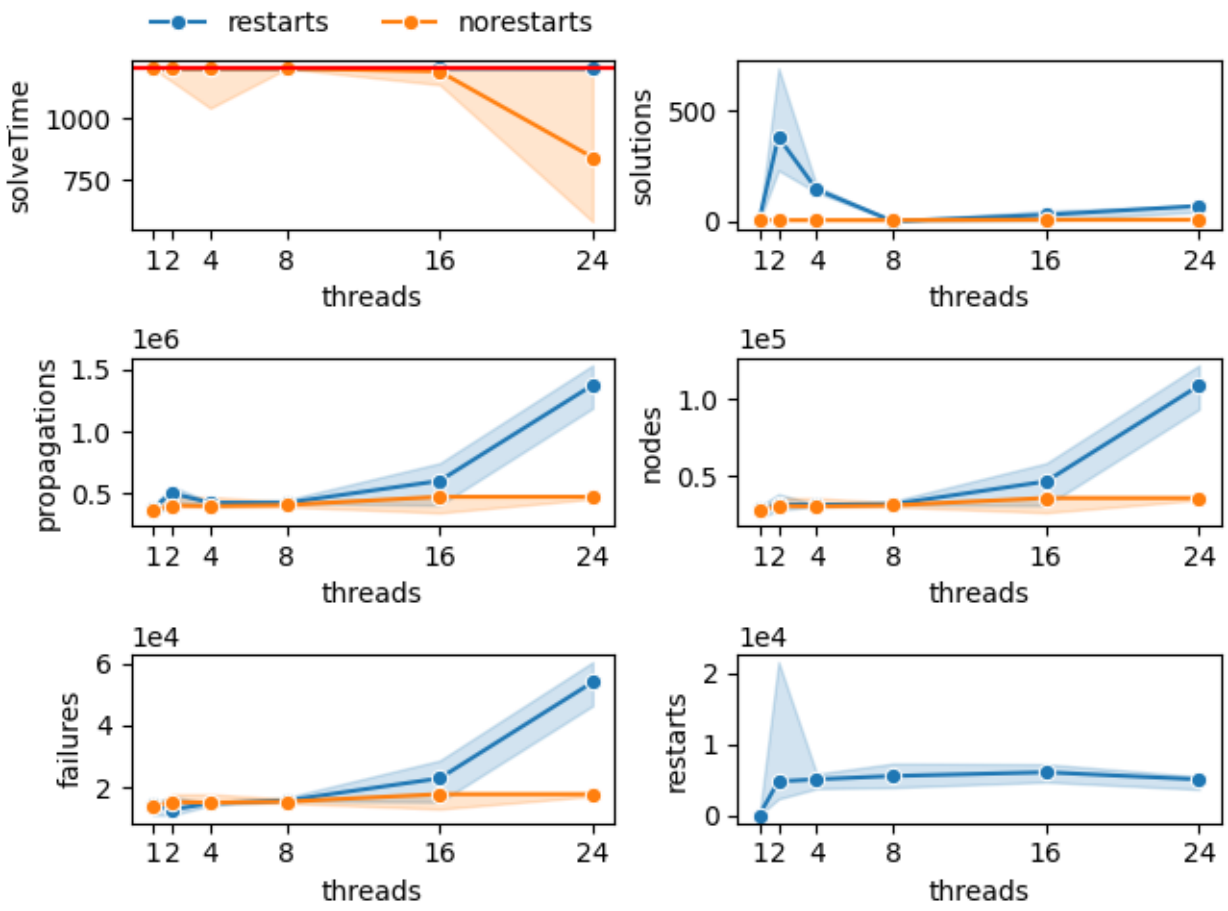

Figure 12: Trend for Pentominoes-Zayenz with `size_20_tiles_15_seed_4711_strategy_source` as input. The number of solutions, propagations, explored nodes, failures, and restarts increases with more threads used. We do not finish for most runs until 24 threads, but at 4 and 16 threads we get one run which does finish within the time limit. Only one run does not finish at 24 threads. With restarts enabled, we see a significant increase in explored nodes going from 16 to 24 threads, whereas without restarts we only see a minor increase going from 8 to 16 threads. We do see a peak at 2 threads for the amount of restarts, and a corresponding peak in the amount of solutions found (which do include duplicates).

In Table 14 and Table 15 we can see the percentage increase in number of nodes explored. With restarts we always explore more nodes than we do at 1 thread. Without restarts, we end up exploring fewer explored nodes for the problem with `size_5_tiles_20_seed_17_strategy_close` as input, although we explore more nodes at 4 threads. The other problems only see an increase in the number of nodes explored.

	Nodes	**% Increase in explored nodes**				
Threads	1	2	4	8	16	24
`size_5_tiles_20_seed_17_strategy_close`	65169971	−32.44	8.11	−15.46	−38.76	−49.34
`size_15_tiles_15_seed_17_strategy_source`	261794	16.15	154.98	444.32	698.95	871.34
`size_20_tiles_15_seed_4711_strategy_source`	27594	10.91	9.60	11.85	29.23	29.23

Table 14: Percentage increase of explored nodes for the Pentominoes-Zayenz problems without using restarts, when searching for all solutions.

	Nodes	% Increase in explored nodes				
Threads	1	2	4	8	16	24
size_5_tiles_20_seed_17_strategy_close	463214	744.57	1337.27	3681.96	2236.25	1304.38
size_15_tiles_15_seed_17_strategy_source	264032	11.85	131.69	312.19	647.97	759.00
size_20_tiles_15_seed_4711_strategy_source	28901	8.91	8.44	9.41	60.83	277.84

Table 15: Percentage increase of explored nodes for the Pentominoes-Zayenz problems with no-goods from restarts enabled, when searching for all solutions.

6 Conclusion

The goal of this thesis is to determine how much redundant work we do when using a parallel constraint solver like Gecode when solving a constraint problem. We looked at optimisation and satisfaction problems. To see how much overhead we get when we use more than 1 thread, we look at the number of explored nodes at 1 thread, and the percentage increase relative to this number at 2, 4, 8, 16, and 24 threads.

For two of the satisfaction problems we get expected results, where using more threads leads to fewer number of nodes explored. This are the Monomatch problem with `data_n_5_percentage_0.5` as input, and the Pentominoes-Zayenz problem with `size_10_tiles_10_seed_17_strategy_target` as input. After 8 threads we end up with fewer than 0.1% of the explored nodes for 1 thread. Using no-goods from restarts causes a more rapid decrease in amount of explored nodes.

The other satisfaction problems see almost the complete opposite result. For these problems we see an increase in number of nodes explored after 1 thread for most of the problems. Using no-goods from restarts leads to many more explored nodes, and causes an increase in time needed to solve the problems, sometimes not even being able to solve the problem within 20 minutes.

The optimisation problems are split up in three general groups. The first group consists of problems like the Carpet Cutting problem and the Peaceable Queens problem. When we do not enable restarted search, we see a general downward trend for the number of explored nodes, with a small increase at 24 threads. With no-goods from restarts we always explore many more nodes after 1 thread. The Carpet Cutting problem with `mzn_rnd_test.12` as input and the Java Routing problem with `trip_6_2` as input see an increase in amount of explored nodes with more threads used when we do not use restarts, but with restarts enabled we get the same trend where we explore more nodes with restarts enabled. We almost always see a peak at 2 threads in the solve time and number of restarts when we use no-goods from restarts, except for the problems which did not finish with 1 thread, and for the ATSP problem with `instance5_0p15` as input, which does not see a decrease in the solve time.

The second group of optimisation problems consists of the Flowshop Workers problems, which all do not finish within the time limit of 20 minutes and give no solutions. These results are not very surpising, since more nodes need to be explored to find the solution, and more threads allows Gecode to explore more nodes. The amount of restarts is not consistent for all the runs, but for some runs we do observe the peak at 2 threads. Other runs also had peaks at 8 or 16 threads.

The third group consists of two outliers, with completely different results. The first problem is the Community Detection problem, which does not see any increase in explored nodes because it only explores one. The other problem is the ATSP problem with `instance10_0p25`, which does not finish without restarts enabled. The amount of nodes therefore keeps increasing, as with the Flowshop Workers problems. With restarts enabled we finish after 8 threads. Up to 8 threads we follow the same increasing trend in explored nodes as without restarts, after 8 threads we see a massive decrease in explored nodes. At 2 threads we again see a peak in the number of restarts.

We also looked at finding all the solutions of the satisfaction problems, but we do not get meaningful results for most of the problems since we reach a time limit. Only four problems finish within the time limit of 20 minutes. All these problems see constant amount of explored nodes without restarts enabled. When no-goods from restarts are used, Gecode does not finish the problems when more than 1 thread is used.

The number of solutions continue to increase with more threads added. The output files contain duplicate solutions when we use restarts and when we use more than 1 thread, which suggests something goes wrong in the way Gecode's algorithm works when exploring all solutions. Even when subtracting all the duplicate results from the output files do we still end up with more than the number of solutions that we get when we do not use restarts.

We tried to also search all solutions for the optimisation problems, but the results are almost the same as for just one solution, with only minor differences between the results. All the problems show the same trend as for one solution.

We conclude that there is sometimes overhead when using parallel search to solve a constraint problems, but more research is necessary.

7 Future Work

Since the results are not very decisive, there are multiple ways we can get a more accurate overview of how much redundant exploration takes place when we use parallel search algorithms. The easiest way to get a more accurate representation is by running more tests. Currently there is a lot of fluctuation between the runs. By performing more runs it becomes clearer which values are common occurences and which values are clearly outliers.

Instead of using more runs per problem for a given amount of threads, we could also use more than 20 problems, or by using an increased time limit so that more problems can be solved. Currently we only tested with 10 different problems and a total of 20 input files. If we run the tests on more problems we can get a more accurate overview. We could also use more parameters for the solver. We currently only change the amount of threads and the amount of solutions we search. We either run without restarts, or with no-goods from restarts with a geometric restart strategy. Tests could be performed with restarts enabled, but without no-goods learning. Another option is to use a different restart strategy, like Luby or a sequential restart sequence. Parameters like the recomputation adaption distance might influence the amount of nodes there needs to be explored.

Further investigation is also necessary why Gecode produces duplicate values for satisfaction problems when using no-goods from restarts when we are searching for all solutions.

Finally, one could also use a different solver than Gecode, and see how different solvers compare for the same problem with the same parameters. This way we can get a more accurate overview about how much overhead can be attributed to the way Gecode works, and how much the results actually show a real trend of redundant exploration.

References

[1] Gottlieb Allan J. Almasi, George S. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Redwood City, CA, USA, 1989.

[2] Mohammed Rezgui Arnoud Malapert, Jean-Charles Régin. Embarrasingly parallel search in constraint programming. *Journal of Artificial Intelligence Research*, 57:421–464, 2016.

[3] Christian Bessiere and Jean-Charles Régin. Refining the basic constraint propagation algorithm. pages 13–26, 01 2001.

[4] Microsoft Corporation. Microsoft hpc pack 2012 r2 and hpc pack 2012. `http://technet.microsoft.com/en-us/library/jj899572.aspx`.

[5] Bart Selman Nuno Crato Gomes, Carla P and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1-2):67–100, 2000.

[6] Sais Lakhdar Tabary Sebastien Vincent Vidal Youssef Hamadi Lecoutre, Christophe and Lucas Bordeaux. Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(3-4):147–67, 2007.

[7] Rui Machado, Vasco Pedro, and Salvador Abreu. On the scalability of constraint programming on hierarchical multiprocessor systems. In *2013 42nd International Conference on Parallel Processing*, pages 530–535, 2013.

[8] Jegou Philippe and Cyril Terriou. Combining restarts, nogoods and bag-connected decompositions for solving csps. *Constraints : An International Journal*, 22(2):191–229, 2017.

[9] M. A. Hakim Newton Polash, Md Masbaul Alam and Abdul Sattar. Constraint-directed search for all-interval series. *Constraints : An International Journal*, 22(3):403–31, 2017.

[10] Jean-Charles Régin and Arnaud Malapert. *Parallel Constraint Programming*, pages 337–379. Springer International Publishing, Cham, 2018.

[11] Carl Christian Rolf and Krzysztof Kuchcinski. Combining parallel search and parallel consistency in constraint programming. 2010. TRICS workshop at the International Conference on Principles and Practice of Constraint Programming ; Conference date: 06-09-2010 Through 10-09-2010.

[12] C. Schulte. Gecode: Generic constraint development environment. `https://www.gecode.org`.

[13] Monash University and CSIRO Data61. Minizinc challenge 2021 results. `https://www.minizinc.org/challenge2021/results2021.html`, 2021.