Parallelism in Constraint Programming

Rolf, Carl Christian

2011

Citation for published version (APA):
Rolf, C. C. (2011). *Parallelism in Constraint Programming.*

Total number of authors:
1

Parallelism in Constraint Programming

Carl Christian Rolf

ABSTRACT

Writing efficient parallel programs is the biggest challenge of the software industry for the foreseeable future. We are currently in a time when parallel computers are the norm, not the exception. Soon, parallel processors will be standard even in cell phones. Without drastic changes in hardware development, all software must be parallelized to its fullest extent.

Parallelism can increase performance and reduce power consumption at the same time. Many programs will execute faster on a dual-core processor than a single core processor running at twice the speed. Halving the speed of a processor can reduce the power consumption up to four times. Hence, parallelism gives more performance per unit of power to efficient programs.

In order to make use of parallel hardware, we need to overcome the difficulties of parallel programming. To many programmers, it is easier to learn a handful of small domain-specific programming languages than to learn efficient parallel programming. The frameworks for these languages can then automatically parallelize the program. Automatically parallelizing traditional programs is usually much more difficult.

In this thesis, we study and present parallelism in constraint programming (CP). We have developed the first constraint framework that automatically parallelizes both the consistency and the search of the solving process. This allows programmers to avoid the difficult issues of parallel programming. We also study distributed CP with independent agents and propose solutions to this problem.

Our results show that automatic parallelism in CP can provide very good performance. Our parallel consistency scales very well for problems with many large constraints. We also manage to combine parallel consistency and parallel search with a performance increase. The communication and load-balancing schemes we developed increase the scalability of parallel search. Our model for distributed CP is orders of magnitude faster than traditional approaches. As far as we know, it is the first to solve standard benchmark scheduling problems.

Acknowledgements

PAPERS

The following papers are included in this thesis.

- Parallel Consistency in Constraint Programming, *The International Conference on Parallel and Distributed Processing Techniques and Applications: SDMAS workshop*, 2009.

- Combining Parallel Search and Parallel Consistency in Constraint Programming, *International Conference on Principles and Practice of Constraint Programming: TRICS workshop*, 2010.

- Load-Balancing Methods for Parallel and Distributed Constraint Solving, *IEEE International Conference on Cluster Computing*, 2008.

- State-Copying and Recomputation in Parallel Constraint Programming with Global Constraints, *Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2008.

- Distributed Constraint Programming with Agents, *International Conference on Adaptive and Intelligent Systems*, 2011.

Other papers, not included in this thesis.

- Load-Balancing Methods for Parallel Constraint Solving, *International Conference on Principles and Practice of Constraint Programming: Doctoral Program*, 2008.

- Task Parallelism in Constraint Programming: Parallel Consistency, *CORS-INFORMS International Meeting*, 2009.

- Parallel Consistency in Constraint Programming, *Second Swedish Workshop on Multi-Core Computing*, 2009.

- Parallel Solving in Constraint Programming, *Third Swedish Workshop on Multi-Core Computing*, 2010.

POPULAR SCIENCE SUMMARY

At our workplace, at home, and on the road, we rely on software. It has become one of the central technologies on which we base our society. Yet, software is perhaps the major technology that is understood the least by the general population.

Most software today is written in languages that requires the programmer to specify not only *what* is to be achieved, but also *how*. This is a major limitation, since the best way to do things depends on the hardware.

The older the software code is, the more the computer architectures are likely to have changed. Today, many companies still run software written in the seventies. While this code is likely to be all but bug free, the performance of these systems is lacking.

In the past decade, the hardware development has moved more and more towards parallelism. The reason is that processors can no longer become faster while still remaining reasonably easy to cool. This situation will continue unless some drastically different materials or technologies emerge.

Every day, we become more reliant on constant access to high performance software and software based services, such as online banks. This increases the performance requirements of almost all software. Previously this was not a major problem, as faster processors would automatically run all programs faster. However, today, the increase in hardware capability comes in the form of more potential parallelism.

Thus, the performance needs of tomorrow's users can only be satisfied by embracing parallelism.

Writing efficient parallel programs is the biggest challenge of the software industry for the foreseeable future. We are currently in a time when parallel computers are the norm, not the exception. Soon, parallel processors will be standard in cell phones. Without drastic changes in hardware development, all software must be parallelized to its fullest extent.

Parallelism can increase performance and reduce power consumption at the same time. Many programs will execute faster on a dual-core processor than a single core processor running at twice the speed. Halving the speed of a processor can reduce the power consumption up to four times. Hence, parallelism can give more performance per unit of power.

In the high-performance computing industry, energy efficiency is a primary concern. The majority of the total cost of a supercomputer today, during its lifetime, often comes from cooling and power consumption. More parallelism can both reduce the cost and the environmental impact of the software services we rely on.

In order to make use of parallel hardware, we need to overcome the difficulties of parallel programming. To many programmers, it is easier to learn a handful of small domain-specific programming languages than to learn efficient parallel programming. The frameworks for these languages can then automatically parallelize the program. Automatically parallelizing traditional programs is usually much more difficult.

There are many programming paradigms in use today for solving difficult problems, such as scheduling. For instance, constraint programming (CP), integer programming (IP), and satisfiability programming (SAT). Of all these paradigms, CP is usually considered to be closest to the holy grail of programming: *the user states the problem, and the computer solves it*.

In this thesis, we study and present parallelism in constraint programming. We have developed the first constraint framework that automatically parallelizes both the consistency and the search of the solving process. This allows programmers to avoid the difficult issues of parallel programming. We also study distributed CP with independent agents and propose solutions to this problem.

Our results show that automatic parallelism in CP can provide very good performance. Our parallel consistency scales very well for problems with many large constraints. We also manage to combine parallel consistency and parallel search with a performance increase. The communication and load-balancing schemes we developed increase the scalability of parallel search. Our model for distributed CP is orders of magnitude faster than traditional approaches. As far as we know, it is the first to solve standard benchmark scheduling problems.

One of the main uses of CP today is air-crew scheduling, that is, creating work schedules for pilots and air hostesses. Providing even slightly better schedules can lead to savings in the order of millions of dollars annually.

By using parallel solving, better schedules can be found for many problems. Depending on the industry, this can lead to major reductions in fuel use, increased production speed, or less over-production, all of which are important for competitiveness and the environment.

CONTENTS

Included Papers 57

Appendix 183

INTRODUCTION

Efficient parallel programming is necessitated by current hardware development. Unless a drastic change in hardware design occurs, all programs need to be parallelized. Software must become parallel either because of performance requirements, or to offer more functionality with retained performance.

Constraint programming (CP) is one of the paradigms closest to the holy grail of computing: *the user states the problem and the computer solves it* [17]. Similar approaches such as satisfiability programming and integer programming, rarely offer as intuitive modeling. The declarative nature of CP creates opportunities for parallelization of CP programs.

In this thesis, we study and present parallelism in constraint programming. We have developed the first constraint framework that automatically parallelizes both the consistency and the search of the solving process. This allows programmers to avoid the difficult issues of parallel programming. Solving these issues is the greatest software challenge of the foreseeable future. We also study distributed CP with independent agents and propose solutions to this problem.

In the following chapters, we present constraint programming and parallelism. We describe ways to parallelize constraint programs and methods to increase performance. We then summarize the contributions of this thesis. Finally, we present our conclusions and the future.

1   Constraint Programming

Constraint programming first started to appear in the late 1960's [31]. In the beginning it mostly consisted of logic programming extensions to traditional programming languages that allowed the programmer to use constraint relations. These 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.

A quick and simple example of constraint programming is the send more money-problem, where the digits 0 to 9 are assigned to the letters in a way that solves the equation below.

$$
\begin{array}{cccccc}
  &   & S & E & N & D \\
+ &   & M & O & R & E \\
= & M & O & N & E & Y \\
\end{array}
$$

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

Figure 1.1: Send more money-problem modeled in Eclipse Prolog.

The program in Figure 1.1 models the send more money-problem in Eclipse Prolog. The capital letters are variables that can assume any numeric value. The global constraint on line 3 constrains the variables to only hold a value between 0 and 9. On line 4 it is declared that all the

variables must have different values. On line 5 and 6 the variables representing the first digits in the problem numbers are set to not be equal to zero. On the last three lines the relationships between the values are defined. The actual search for a valid solution is started on line 10. During the labeling process all the constraints are checked for consistency and the variables are assigned values.

Large industry relevant problems can be modeled quite easily in CP. Today, one of the main uses of CP is air crew rostering, in which the schedules of pilots and stewards are created for the coming weeks. This is a very difficult problem, as there are thousands of employees, each requiring a variable in the problem. Integer programming is popular for air crew rostering, but, e.g., modeling that two employees cannot work with one another is quite inefficient and complex. In CP, this can be modeled by a single inequality constraint.

When we try to solve a constraint problem, we usually have to search through combinations of assignments. For instance, we can start by assigning value 1 to S in the problem in Figure 1.1. Then, we can remove this value from all other variables. This removal is performed by *consistency algorithms* in the constraints. Then, if all constraints hold, we continue searching by assigning a value to the next variable and so on. This search progresses until all variables have been assigned a value, giving us a solution to the problem. If a constraint is violated, we have to undo the last assignments and try a new value for that variable.

Solving constraint problems is very difficult. It falls into the category called *NP-complete* problems, which are very likely to require massive computing power to solve. Even a small instance of an NP-complete problem can in some cases take years of computing power to solve.

Constraint programming allows many heuristics to be used in order to minimize the time needed for solving. These heuristics carry no guarantee of improving performance. In practice, however, using the CP heuristics often reduce the solving time by several orders of magnitude compared to not using any heuristic.

2 Parallelism

For many decades *Moore's law* has held. It dictates that the number of transistors on a given surface will double every 2 years [38]. Previously, this density increase has been accompanied by roughly a doubling in processor speed. However, in the past few years this connection has fallen apart.

It is no longer feasible to increase the processor speed as we have done in the past. The main reason is the heat produced by the processor. Unless drastically different materials emerge, a doubling in processor speed means the heat output increases fourfold. The limit at which it is feasible to cool away the heat has already been reached [3].

Moore's law still holds. Today, however, the doubling comes in the form of putting several processor cores on the same chip. This happens while the processor speed is held constant. Instead of doubling the clock frequency, as before, we double the number of cores. This means that we will be able to run twice as many programs in parallel.

Making programs parallel is necessary in order to harness the power of the current hardware architectures. As long as putting more processor cores on the same chip is the cheapest way to improve performance, more parallelism is needed in all high-performance software.

The main challenge of modern hardware is writing parallel programs. In imperative languages, the programmer has to specify not only *what* is to be achieved, but also *how*. Parallel programming in imperative languages, such as Java and C, is entirely different from sequential programming. In parallel programs, several software modules can modify the exact same data at the exact same time. This might create non-deterministic behavior.

In order to write parallel programs in imperative languages, we sometimes need to give software modules exclusive access to data. In other words, we need to prevent all other program modules from reading or writing the locked data. However, the larger the parts to which we grant exclusive access, the more parallelism we lose.

Reducing the need for exclusive access is necessary to maximize performance. Sometimes, allowing one tenth of your program remain non-parallel severely limits the performance [2]. For example, assume the fol-

lowing: we have a house with ten rooms, only one person is allowed in a room at any time, and each room takes the same amount of time to paint. Then, if one person can paint the house in ten hours, ten people can paint the house in one hour. But if you have ten people and eleven rooms, it will take twice as long.

Imperative languages place a too heavy burden on the programmer. Minimizing exclusive access is difficult and bug prone. Declarative languages, such as CP, can in principle automatically parallelize difficult problems. For many programmers it is probably easier to learn a handful of domain specific declarative languages than to learn high performance parallel programming in imperative languages.

3   Parallelism in Constraint Programming

When we automatically parallelize a constraint program, we divide the work needed to solve the problem between several constraint solvers. These solvers are either run on their own processor cores or on separate computers. Depending on the hardware used, communication either takes place in main memory or over a network.

One way of parallelizing the program in Figure 1.1 is to evaluate the constraints in parallel. This is called *parallel consistency*. It means that we run the consistency algorithm of each constraint in parallel. This can be very useful if the consistency is time consuming.

Another way to parallelize constraint programs is to split the possible assignments of the variable between computers. This is called *parallel search*, since the search takes place on several solvers at the same time. An example of such a split is depicted in Figure 1.2. After the split, both solvers can work independently of each other. The theoretical maximum performance gain usually scales super-linearly with the number of solvers used. However, due to hardware and software limitations, the real world scalability is typically much lower.

The kind of splitting depicted in Figure 1.2 is the most common way of parallelizing constraint programs. In this thesis we will also study parallel consistency, and how to combine it with parallel search. As far as we know, this is the first time this combination has ever been studied.
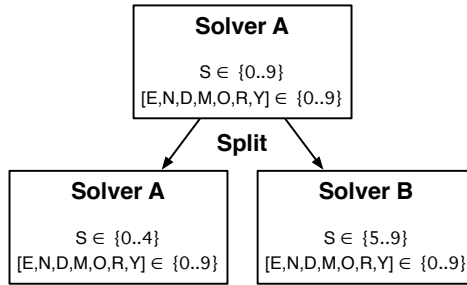
Figure 1.2: Splitting in the send more money problem.

4 The Future

It is always hard to predict the future, but some trends are clear. If the current hardware development continues towards more parallelism, all programs will have to be written to make the most of the hardware. This is especially important for high performance programs.

Another major trend in software development is a move towards more abstraction and the use of domain specific languages. More abstract representation makes the software more portable between different computer architectures and reduces maintenance costs. Domain specific languages allow complex software problems to be described in a more human readable way, this shortens development time and reduces costs.

Constraint programming offers a way to write programs with a high level of abstraction. CP is also suitable for complex problems that require a lot of processing power to solve. Furthermore, the solving of these problems can be automatically parallelized to make use of the latest hardware. Once, the problem model is written, it can easily be moved to other constraint solvers in order to maximize performance.

In industry, integer programming (IP) is much more prevalent than CP. IP has both benefits and drawbacks. Satisfiability programming (SAT) is overtaking CP with regard to performance for specific classes of problems, but the modeling is more complex. In the future, CP, IP, and SAT will move closer so that problem models can be split between these different solving methods to make the most of each paradigm. In fact, this is already happening to some extent [62]. Parallelism is likely to be a key component in the solving whichever way CP/IP/SAT are mixed.

BACKGROUND

This chapter introduces the basics of constraint programming and parallelism.

1 Constraint Programming

Constraint programming has been used with great success to tackle many NP-complete problems such as graph coloring and scheduling [31, 55]. Problems modeled in CP are called constraint satisfaction problems (CSP).

Formally, we define a CSP as a 3-tuple $P = (X, D, C)$, called a *constraint store*, with the following properties.

- X is a set of variables;

- D is a set of finite domains;

- C is a set of binary constraints; and

- c_{ij} is a relation between x_i and x_j

Solving a CSP means finding assignments to all variables in X, such that the value of x_i belongs to d_i, while all the constraints are satisfied. Solving a CSP is NP-complete [55].

There are two main parts that make up constraint programming, modeling and solving. The modeling typically uses a declarative language, such as MiniZinc [39]. The solving is performed by a constraint framework that performs *consistency* and *search*.

Figure 2.1 depicts a simple example of a CSP. We have three variables, X, Y, and Z. They all have to assume different values. This is modeled using $\neq$ constraints.

All the variables in the example can assume values between 0 and 9. Since the domains are not singletons, finding a solution requires search. We do this by picking a variable and assigning it one of its possible values. In this case, we pick X and assign it the value 0. By running the consistency algorithm of $X \neq Y$ and $X \neq Z$, we can remove the value 0 from the domains of Y and Z. Since some domains are still not singletons, we have to continue our search. We pick Y, and assign it the value 1. Now the consistency can remove 1 from the domain of Z. In the last search step, we pick Z and assign it the value 2. Now all variable domains in our problem are singletons and all constraints are satisfied. Hence, we are finished and have found a solution.
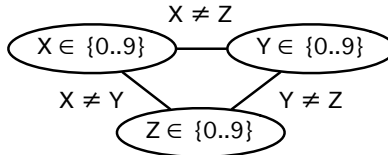
Figure 2.1: Model of a simple CSP.

1.1 Modeling

Modeling a problem in CP is quite simple. Students in the courses we teach usually pick up the basics within two or three weeks. For example, Figure 2.2 shows how to model sudoku as a constraint problem. This is the entire code needed to solve a 9x9 sudoku using MiniZinc [39].

Thanks to its declarative nature, constraint programming does not require the programmer to deal with how the actual solving takes place. Furthermore, given a programmer-friendly constraint framework, no significant changes need to be made to the problem declaration in order to solve the problem using parallelism. This means that synchronization and other difficult aspects of parallel programming can be left entirely to the creator of the constraint framework.

```
1 % includes the AllDifferent constraint in the program
2 include "all_different.mzn";
3 % creates 81 variables with values between 1 and 9
4 array [1..9, 1..9] of var 1..9: sudoku;

6 % creates one AllDifferent for each row
7 constraint
8   forall (row in 1..9)
9     (all_different (col in 1..9) (sudoku[row, col]));
10 % creates one AllDifferent for each column
11 constraint
12   forall (col in 1..9)
13     (all_different (row in 1..9) (sudoku[row, col]));
14 % creates one AllDifferent for each box
15 constraint
16   forall (row, col in {0, 3, 6})
17     (all_different (i, j in 1..3) (sudoku[row+i, col+j]));

19 solve satisfy;      % starts the search for one solution
20 output [ show(sudoku) ];   % prints the solution
```

Figure 2.2: Sudoku modeled in constraint programming.

In the model depicted in Figure 2.2 we use global constraints. These encompass several of the variables in the store. All such constraints can be translated into a purely binary form [55]. However, this would reduce the efficiency of the consistency.

Modifying our earlier definition to allow global constraints, we get the 3-tuple $P = (X, D, C)$ with the following properties.

- X is a set of n variables;

- D is a set of n finite domains;

- C is a set of n-ary constraints; and

- c_K is a relation over the set of variables $\{x_i, i \in \{0..n\} \mid x_i \in X\}$.

1.2 Consistency

Consistency is used to remove values that cannot be part of a solution. Whenever the domain of a variable changes, all the constraints containing that variable will be evaluated for consistency. Due to time complexity issues, consistency is rarely complete [13]. Hence, the domains may contain values that are *locally consistent*, but cannot be part of a solution. The process of removing inconsistent values is called *pruning*.

More formally, consistency can be described as follows. We start with the store $P = (X, D, C)$. Assume $D^0 = D$, then D^1 is the set of finite domains representing the values for X that remained after the pruning of the consistency methods of the constraints in C. We apply the consistency methods of the constraints in C iteratively until $D^N = D^{N-1}$, i.e., we have reached a fixpoint. Then, if $\exists d'_i = \emptyset$ the store is inconsistent and we have no solutions starting from D. In this case we have to undo assignments in the search.

One of the most efficient ways of improving the performance of the solving process is to use constraints that maximize the pruning. The most powerful constraints, referred to as *global constraints*, include several or all of the variables in the problem. If a global constraint containing all variables is satisfied, finding a solution is often quite simple. The downside of global constraints is that they are computationally more expensive in the average case, since they often implement algorithms of rather high computational complexity [13].

Representing a global constraint with a set of binary constraints reduces the pruning power, but only the constraints whose variables change need to be recomputed, reducing the average time-complexity.

The model of sudoku in Figure 2.2 uses the AllDifferent constraint. This is a global constraint taking a list of variables and ensuring that they all have different values. The pruning algorithm for this constraint relies either on finding Hall intervals and has a time complexity of $O(nlog(n))$ [43] or bipartite graph matching [46].

All problem models can be reduced to one using only binary constraints. But this usually comes at the cost of greatly reduced pruning, and therefore, much longer execution times.

Typically, consistency methods of constraints are monotonic [61]. Hence, most constraints never increase the size of the domains of the variables they contain. In order to achieve consistency, we run all constraints whose variables have changed until there is no change from one iteration to the next. When no changes occur, we have reached a fixpoint.

If all the consistency algorithms used are monotonic, the order in which they are executed is irrelevant for correctness. Hence, such consistency can be automatically parallelized. Depending on the hardware architecture, there may be data dependencies that need to be handled if two constraints want to prune the same domain. In this case, all prunings can be collected and enforced at the same time. No changes need to be made to the problem model in order to use parallel consistency.

```
1  // set of constraints containing changed variables C_changed
2  // the constraint store Store
3
4  consistency() {
5      while C_changed ≠ ∅
6          c ← selectConstraint(C_changed)
7          C_changed ← C_changed \ {c}
8          c_i.enforceConsistency(Store)
9          if not c.consistent
10             return false
11         else
12             for each Variable v ∈ c
13                 if v.changed
14                     C_changed ← C_changed ∪ v.constraints
15                     v.changed = false
16     return true
17 }
```

Figure 2.3: The basic algorithm for consistency.

Figure 2.3 depicts the basic algorithm for consistency. Every time a variable changes, we have to re-evaluate all the constraints that it participates in. If all consistency algorithms are monotonic, we will eventually reach a fixpoint where no more changes will be performed between iterations. Then we are ready to return to the search.

1.3 Search

In order to find solutions to a CSP we perform search, usually depth-first search (DFS) [31]. Typically, in each search node, we pick a variable and then assign a value to it. The order in which the variables are chosen is decided by a heuristic that guides the search. Typically *first-fail* gives good performance. This heuristic always picks the variable with the smallest domain. This often leads to a smaller search space [55].

More formally, if we have a store $P = (X, D, C)$ as defined in Section 1.1. We take a variable x_i from X so that $X' = X \setminus \{x_i\}$, assign it a value k from d_i by setting $d_i' = \{k\}$. If applying consistency does not result in inconsistency, we progress recursively by picking another variable. When $X' = \emptyset$, we have found a solution. If an inconsistency was found, we backtrack by setting $d_i = d_i \setminus \{d_i'\}$. If this leads to $d_i = \emptyset$, we backtrack to the previously chosen variable and reset d_i to contain all the values we had when we chose x_i.

The depth-first search algorithm in CP is depicted in Figure 2.4. Once a variable has been selected, the solver will pick a value to assign. How this value is chosen is controlled by another heuristic. When searching for a minimal solution, it is typically good to select the smallest value in the domain of the chosen variable.

There are no guarantees that the order in which the variables are assigned is optimal [55]. The order of values assigned carries no guarantee either. Minimizing the execution time for solving a particular problem often comes down to finding especially good ways to order the variables and values that the search will explore.

```
1 // variables to be labeled X, with FDV xᵢ ∈ X
2 // domain of xᵢ is dᵢ
3
4 search() {
5     if X ≠ ∅
6         if not consistency()
7             return false
8         else
9             xᵢ ← selectVariable(X)
10            X ← X \ {xᵢ}
11            k ← selectValue(dᵢ)
12            xᵢ ← k
13            if search()
14                return true
15            else
16                dᵢ ← dᵢ \ {k}
17                X ← X ∪ xᵢ
18                if search()
19                    return true
20                else    // backtrack
21                    dᵢ ← dᵢ ∪ k
22                    return false
23    else
24        if consistency()
25            printSolution
26            return true
27        else
28            return false
29 }
```

Figure 2.4: The search algorithm for a single solution.

In constraint satisfaction problems, each branch of the search tree is independent of the other branches. If we are trying to find the optimal solution, the only relation between branches is in how the cost of the solutions are communicated.

Since the branches are independent, they are easy to parallelize. Automatically parallelizing the search process is simply a matter of sending part of the domain of the last selected variable to another instance of the constraint solver. The two instances can then work independently of each other. The problem model need not be changed to use parallel search.

1.4   Distributed Constraint Problems

Distributed constraint satisfaction problems (DisCSP) are quite similar to CSP. DisCSP can be defined with the 4-tuple $P = (A, X, D, C)$, where A is a set of agents and X is a set of variables so that x_i is controlled by a_i [55]. D is a set of finite domains, and C is a set of sets of *binary* constraints. Each variable x_i has a finite domain d_i, and each agent a_i has a set of binary constraints c_i. The constraints act as connections between the agents. Furthermore, each variable is controlled by exactly one agent. Lastly, the set of constraints between agents a_i and a_j are given by the intersection $c_{ij} = c_i \cap c_j$.

Figure 2.5 depicts a simple DisCSP. Each agent holds only one variable, and we have binary constraints between the agents. Any changes to a variable that is in a constraint between two agents are communicated. Typically the communication is done by sending one value at a time during search and the receiving agent answers with either a *Good* or *NoGood* message [55].
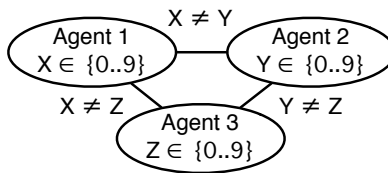
Figure 2.5: Model of a DisCSP, where each agent holds one variable.

DisCSP have a number of motivations [55]. The two main motivations are robustness and privacy. Robustness is improved since there are several independent agents running. This will guarantee that at least part of the problem will be solved even if some agents stop working. With regard to privacy, not all data needs to be shared between computers during solving. This is useful during, for instance, negotiations about schedules, where the parties do not want to say why they are not available at certain times.

Almost all work on DisCSP focuses on the scenario of each agent holding a single variable and only binary constraints exist between the agents [59]. These problems are typically solved with an asynchronous search, where consistency is enforced between the connected agents [69]. The asynchronous search is parallel in nature, and could potentially give an increase in performance.

The main difference between CSP and traditional DisCSP, as defined above, is to eliminate the possibility of global constraints. As mentioned previously, global constraints are central to gain good performance. Hence, DisCSP has yet to mature to the point of being used in industry. Our work on distributed constraint programming (DCP) largely remedies these problems, allowing DCP to run large problems with acceptable performance.

2 Parallelism

2.1 SMP and Clusters

In this thesis, we deal with two types of parallel architectures, symmetric multiprocessing (SMP) and cluster computing. We will often refer to the former as *shared-memory* and the latter as *distribution*. The reason is that the SMP-machine we used had two four core processors, all communicating to the same memory. In contrast, the cluster we used had a separate memory for every processor core, i.e., all communication between cores took place over a network.

Figure 2.6 depicts the two architectures we used in this thesis. There are other architectures which bridge the gap between shared-memory and distribution. However, we did not have access to such machines during our work, hence we leave the curious reader with the following reference [25].
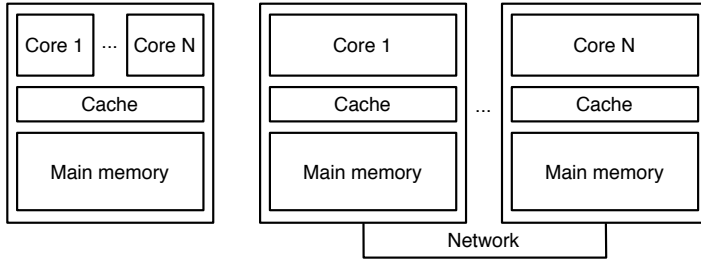
Figure 2.6: The two types of parallel architectures we used in this thesis. SMP on the left, cluster on the right.

Depending on the middleware, there may be no difference to the programmer what the underlying architecture is. In this thesis we developed our own middleware in order to facilitate detailed testing and optimization.

One main difference between SMP and clusters is the processor cache. Caching is central to good sequential performance. However, running several processes on the same cache, as in the SMP-machine we used, will make the cache *dirty*. This means that one thread overwrites the cache which that thread wants. Hence, there will be more cache misses, reducing performance. Depending on the programming language, there may be ways to avoid this. However, Java offers little in the ways of avoiding the invalidation of cache lines used by other processes.

Performance of parallel programs is best measured by *absolute speed-up* [37]. This means that the fastest sequential execution time is divided by the fastest parallel execution time. This contrasts to *relative speed-up*, where the fastest parallel execution time, when run on one processor core, is divided by the fastest parallel execution time. Often, the speed-up is limited to sub-linear [2]. However, super-linear speed-up is possible for depth-first search algorithms [45].

Although the sequential performance of some programs may be better, it is not necessarily relevant to compare parallel performance to them. For example, say that program A is faster than program B because A has more efficient memory handling, rather than better pruning. Then, comparing the execution time of consistency between A and B makes little sense.

2.2 Communication

The main difference between distribution and shared-memory lies in the communication. In distribution, we have to use a network in order to communicate between solvers. In shared-memory, on the other hand, we can simply send a reference to an object allocated in memory. The latter is much faster, although the number of processor cores of an SMP-machine is usually more limited than the number of processors in a cluster.

Communication is a major bottleneck for the performance of parallel programs. When we have a shared-memory, communication is faster since the bandwidth is much higher and with lower latency. On the other hand, the memory bus can quickly become saturated. Reducing the communication is one of the simplest ways of improving the performance of parallel programs.

The communication on an SMP-machine is simply a matter of handing over reference to memory allocated by the sending process. Reducing communication in this case is therefore largely a question of minimizing the memory usage of the program. On the architecture we used, the more memory a program uses, the more it dirties the shared cache. This makes the benefit of reducing the memory usage even greater.

Communication on a cluster is usually done over a fast ethernet network. The communication often uses the TCP/IP protocol, which is what we have used in our work. Hence, there is little to be done on the protocol level to reduce the size of communication. Instead, working on a higher level to improve encoding and compression can increase performance.

Reducing the size of communication through compression increases the execution time on both the sending and the receiving end. Hence, we want to have an adaptable way of communication. Compressing the communication makes no sense if there is lots of free bandwidth. It may take more time to compress, send, and uncompress, than to skip the compression altogether.

2.3 Load-Balancing

Load-balancing is perhaps *the* most important way to improve the performance of parallel programs. Ideally, all processes should have the exact same amount of work to perform so that they all finish at the same time. Otherwise, some processes will have to wait for others to finish. If the wait time becomes long, we lose lots of performance.

Unless we can calculate exactly how much work we have, we cannot expect to split the work perfectly between processes. Hence, we need to be able to send more work to the processes that become idle. In this way, load-balancing is related to communication. Ideally, the work we send will require much more processing time than the time it takes to send it to another process. If the communication is costly, we will lose performance by sending many small pieces of work.

For NP-complete problems, such as those we have studied in this thesis, one cannot efficiently calculate the exact size of the work. We can construct elaborate measurements, such as [10], but they remain rough estimates. Where it is infeasible to calculate the absolute size of work, it makes sense to use relative measures. Simple relative measures from many machines can increase the accuracy of the estimates.

It should be noted that the ultimate goal of load-balancing is not to maximize the processor utilization. If the processor usage increases communication, we may get reduced performance with very high loads. This will be the case for all architectures where the communication is a bottleneck, which is quite common.

Several methods of load-balancing have been developed for parallel DFS [20]. Two of the most well-known methods are random polling and round-robin. Random polling is often more efficient than round-robin for parallel DFS [28, 29], but neither method was designed with CSP solving in mind.

There are two main communication principles for load-balancing, *work-stealing* and *work-sharing* [20]. In the former, idle processes try to steal work from busy processes. In the latter, busy computers try to find idle ones to send work to. On many architectures work-stealing is more efficient [4], since the idle processes are not doing any useful work. However, on a cluster there is little difference between the two principles since both machines have to perform work for communication [36].

When using either work-stealing or work-sharing, the order in which work is shared may be important. For instance, if all processes try to send work to others in the same order, lots of time will be wasted. Using a unique ordering for each process will avoid this scenario. However, as shown in [47], random polling works well for all but the largest architectures.

PARALLELISM IN CONSTRAINT PROGRAMMING

There is increasing interest in work on parallelism in declarative languages. In this thesis we look at parallelism in constraint programming. Parallelism in logic programming is somewhat related, but falls outside the scope of this thesis. Instead, we refer the curious reader to the excellent summary in [22].

Today, parallelism is offered by several constraint solvers. Our solver, JaCoP [27], offers both parallel search and parallel consistency on SMP and clusters. Gecode [18] has parallel search for SMP architectures. Comet [12] offers parallel solving on both SMP and clusters.

1 Modeling for Parallelism and Distribution

One great advantage of declarative programming, when compared to imperative programming, is its potential for automatic parallelism. Since the solving method details are left unspecified, the CP framework can choose freely how to search for solutions. This is a much simpler approach than trying to automatically rewrite imperative code, e.g., by parallelizing loops [66].

Automatically parallelizing the search process does not require any change to the problem model. Only the call to the solver needs to be modified. Figure 3.1 depicts the code for solving Sudoku. The code is written in Java for the JaCoP solver [27], version 1.5. The only change needed to run with parallelism is to change the search method `new SearchOne()` on line 29 to `new SMPSearchOne()`.

```
1   class Sudoku {
2       public static void main(String args[]) {
3           boolean res = false;
4           int[] step = {0, 3, 6, 27, 30, 33, 54, 57, 60};
5           ArrayList<FDV> fdvs = new ArrayList<FDV>(81);
6           FDstore store = new FDstore();
7           for (int i = 0; i < 81; i++)
8               fdvs.add(new FDV(store, "var" + i, 1, 9));
9           for (int i = 0; i < 9; i++) {
10              FDV[] row = new FDV[9];
11              FDV[] col = new FDV[9];
12              FDV[] block = new FDV[9];
13              for (int j = 0; j < 9; j++) {
14                  fdvs.add(new FDV(store, "v" + (i + j),
15                                  1, 9));
16                  row[j] = fdvs.get(i * 9 + j);
17                  col[j] = fdvs.get(i + j * 9);
18                  block[j] = fdvs.get((j % 3) + (j / 3)
19                                        * 9 + step[i]);
20              }
21              store.impose(new Alldiff(row));
22              store.impose(new Alldiff(col));
23              store.impose(new Alldiff(block));
24          }
25          res = JaCoP.Solver.searchOne(store, fdvs,
26              new SearchOne(), new IndomainRandom(),
27              new Delete(new FirstFail()));
28
29           System.out.println("Solution found = " + res);
30      }
31  }
```

Figure 3.1: The Sudoku problem written for JaCoP 1.5.

Most parallel CP solvers such as [10, 34] use a modeling approach similar to ours. The main design trade-off is between ease of use and performance. By providing the programmer with a more configurable search, higher performance can sometimes be achieved. Unfortunately, this may make the constraint framework harder for novices to use. This issue is discussed further in Appendix A.

If the parallel solving takes place on a cluster, complications arise for the creator of the constraint framework. In this scenario, the solvers have to be set up before the search can begin. This means that an instance of the constraint framework must be started on all the computers that will take part in the solving. This initialization phase can be automated, as depicted in Figure 3.6.

Even though a lot of work takes place behind the scenes, the difference between modeling for a cluster, compared to modeling for SMP or sequential execution, is negligible. For cluster computing in our solver, the programmer simply changes the search method on line 29 of Figure 3.1 from `new SearchOne()` to `new ClusterSearchOne()`.

2 Parallel Consistency

In this thesis, we will often refer to parallel search as data parallelism, and parallel consistency as task parallelism. While this may not be absolutely correct terminology in the strictest sense, it is useful to illustrate the inherently different nature of these two types of parallelism. Also, this terminology is used later in the included papers.

Figure 3.2 presents the model of parallel consistency in constraint programming used in this thesis. In the example, the search process is sequential but the enforcement of consistency is performed in parallel. Constraints C1, C2, and C3 can be evaluated independently of each other on different processor cores. Depending on the architecture and constraint framework, the prunings that the constraints perform may need to be synchronized.
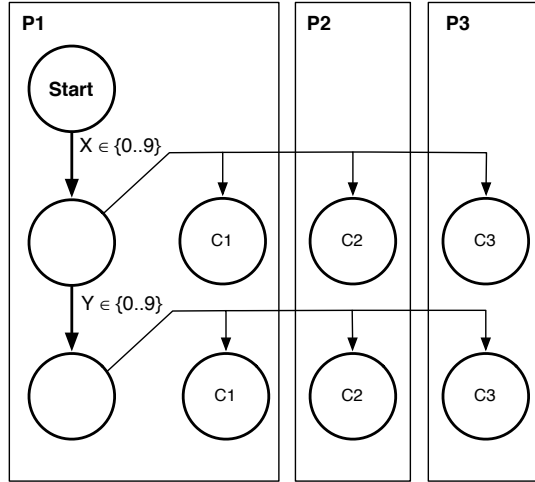
Figure 3.2: Parallel consistency in constraint programming.

Task parallelism is the most realistic type of parallelism for problems where the time needed for search is insignificant compared to that for enforcing consistency. This can happen when the consistency algorithms prunes almost all the inconsistent values. Such strong pruning is particularly expensive and demands more parallelism. The advantage of these large constraints over a massively parallel search is that the execution time will be more predictable.

Parallel consistency in CP means that several constraints will be evaluated in parallel. On most architectures, constraints that contain the same variables have data dependencies, and therefore their pruning must be synchronized. However, since the pruning is monotonic, the order in which the data is modified does not affect the correctness. This is because well-behaved constraint propagators must be both decreasing and monotonic [61]. In our solver this is guaranteed by the consistency method implemented in our solver. The solver takes the intersection of the old domain and that given by the consistency algorithm. The result is written back as a new domain. Hence, the domain size will never increase.
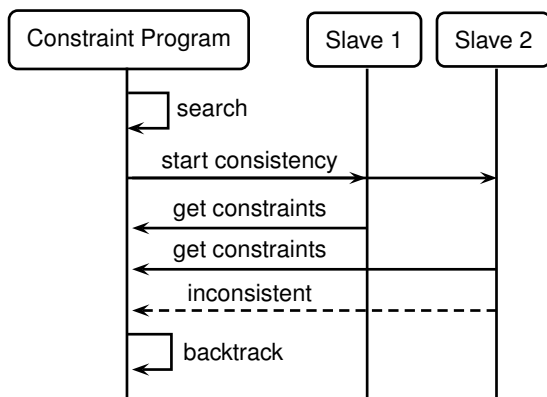
Figure 3.3: Model of parallel consistency with two consistency threads. The dashed line indicates the final return to the constraint program. In this example it leads to a backtrack of the search procedure.

Consistency enforcement is iterative. When the solvers are ready, the constraint queue is divided between them. Then one iteration of consistency can begin. This procedure will be repeated until the constraints no longer change the domain of any variable. The constraints containing variables that have changes will be added to the constraint queue after the updates have been performed.

The order in which constraints are processed is very important for performance [65]. Hence, the best way to divide the constraint queue is to give each consistency thread a set of high-priority constraints to handle. We then distribute the low priority constraints evenly. The variables we use to control the parallel consistency are explained in Appendix A.

2.1 Related Work

Parallel consistency in CP has received very little attention compared to parallel search. Parallel consistency bears some similarity to the AND-parallelism of logic programming studied in, e.g., [8,22,26,42]. However, logic programming is based on rather different assumptions than CP.

Parallel consistency in CP has previously been studied in [40, 57, 58]. However, these papers only investigate arc-consistency, which usually precludes the use of global constraints. Without global constraints, it is very difficult for CP solvers to handle large industry-relevant problems.

As far as we know, parallel consistency with global constraints has only been studied in our work [52,53]. Earlier work uses models that allow global constraints, but these constraints are not used and no experimental results are given for them [57, 58]. In fact, [57] explicitly mentions the need for global constraints in order to make parallel consistency scale well.

Parallel consistency algorithms for specific global constraints are emerging [5]. However, this is a different approach from parallel consistency, since only one constraint is processed at a time. Nevertheless, such algorithms can be used to extend the scalability of parallel consistency.

We are not aware of any work but our own that combines parallel consistency with parallel search [53].

3   Parallel Search

Parallel search is the most studied form of parallelism in CP [10]. It is intuitive to let a solver split the domain of each variable and let another solver handle those possible assignments. A good speed-up can be achieved with a rather small modification to the original solver. For a pure SMP version, only about 1 000 lines of code were required to extend our solver. This constitutes small part of the roughly 100 000 lines of the latest version of JaCoP.

Figure 3.4 depicts a small search tree with parallel search. Parallelizing search in CP can be done by splitting data between solvers. For example, one could 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 $X_i \geq \frac{min(X_i)+max(X_i)}{2}$. The different possible assignments are explored by the solvers running on processors P1, P2, and P3. Clearly, we are not fully utilizing all three processors in this example. At the first level of the search tree, only two out of three processors are active. The procedure for how we control the splitting is explained in Appendix A.
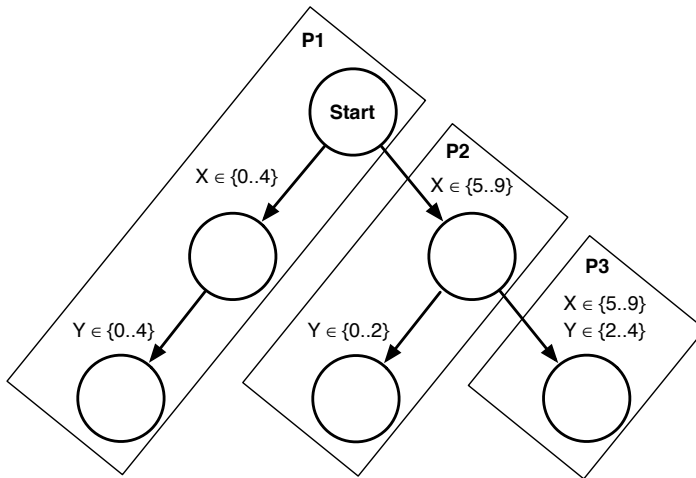
Figure 3.4: Parallel search in constraint programming.

The downside of parallel search is that the search tree below the split-point may be small. The smaller this search space is, the less work is performed in the receiving and sending solvers. Eventually we will reach the point where communicating the work takes more time than exploring both search trees on a single processor.

The algorithm for parallel search is depicted in Figure 3.5. In this algorithm we use work-sharing. If an idle solver is found, we send half of the current domain to that processor. There are several optimizations that can make the splitting more efficient, for instance, sending all but one value for the current variable.

We use work-sharing, although work-stealing is often more efficient on an SMP [4]. However, the difference is rather small when only using eight cores. When running on a cluster, the difference between work-stealing and work-sharing is almost non existent [36]. The busy computer still has to handle all the communication and serialization involved in transmitting work to other computers.

```
1 // variables to be labeled X, with FDV xᵢ ∈ X
2 // domain of xᵢ is dᵢ; set of constraint solvers S
3
4 search() {
5     if X ≠ ∅
6         if not consistency()
7             return false
8         else
9             xᵢ ← selectVariable(X)
10            X ← X \ {xᵢ}
11            for each Solver s in S
12                if s.notBusy
13                    d_remote ← getHalfDomain(dᵢ)
14                    sendWork(s, d_remote)
15                    dᵢ ← dᵢ \ d_remote
16                    dᵢ.splitDomain ← d_remote
17                    break
18            k ← selectValue(dᵢ)
19            xᵢ ← k
20            if search()
21                return true
22            else
23                dᵢ ← dᵢ \ {k}
24                X ← X ∪ xᵢ
25                if search()
26                    return true
27                else    // backtrack
28                    dᵢ ← dᵢ ∪ k ∪ dᵢ.splitDomain
29                    dᵢ.splitDomain ← ∅
30                    return false
31    else
32        if consistency()
33            printSolution
34            return true
35        else
36            return false
37 }
```

Figure 3.5: The parallel search algorithm.

Figure 3.6 illustrates the execution of solving on a cluster. First the servers are initialized on the cluster. Once all the slave solvers are running, the search phase can begin. The work is split between slaves in order to achieve an even processor load. Finally, when the master runs out of work, the termination phase starts. This phase lets all the slaves finish, using a termination detection similar to the one described in [9]. Lastly, the master returns the result to the model program. To utilize the cluster more efficiently, a slave solver can be started on the machine that the master was using before it ran out of work.
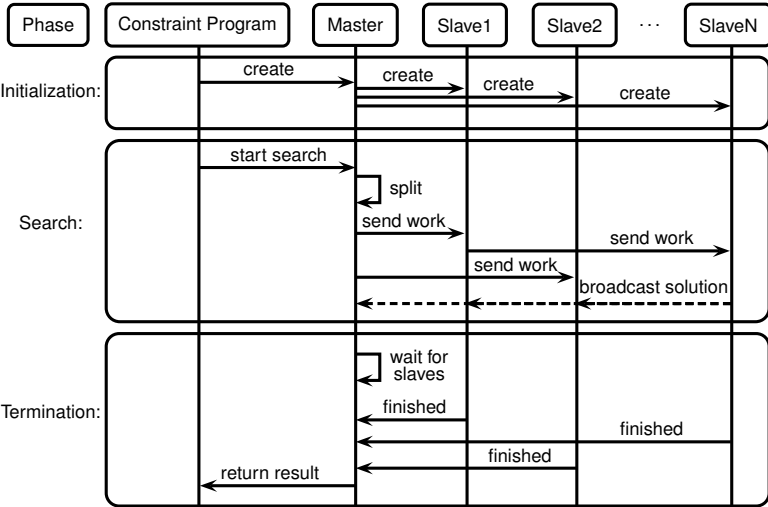
Figure 3.6: The parallel search on a cluster.

3.1   Related Work

Most research on parallelism in CP has focused on parallel search [60]. Parallel search is similar to OR-parallelism in logic programming [22], which has been studied in, among others, [11, 15, 64]. The systems developed in these papers are interesting, but logic programming solvers are based on quite different assumptions than CP solvers.

In [33], parallel local search was studied. This search is very different from the one we have used in all papers presented in this thesis. Hence, the only features of the paper which are relevant to our work are the possibility of using a heterogenous cluster, and the somewhat similar model of parallelism. However, our first study of distribution, [49], was finished well before [33] was published. Furthermore, in [49] and subsequently in [50, 51], we evaluated the performance on eight times as many processors, thereby testing the scalability of our solver much further.

In [68] communication for parallel search in CP was studied. Our work in [51] is somewhat similar in that we also moved beyond the copying used in most previous work, such as [23,47,56]. However, unlike [68], we made a study of both binary problem models and problems that use global constraints, thereby investigating how pruning affects communication needs.

Another major novelty of [51] was our algorithm for switching between communication modes during search in order to increase performance. This was not studied at all in [68]. As far as we know, [51] is the first and only example of dynamically adaptable communication for parallel search in CP.

In [34], transparent parallelization of constraint programs was discussed. The automatic parallelism in that paper is somewhat similar to our earlier work in [49]. However, the study in [34] used a rather limited number of computers, more specifically, a four-machine cluster. In [49–51], we went much further and tested our solver on a 32-machine cluster. Scalability problems barely arise when only using four machines. Hence, the need for better communication and load-balancing, as introduced in [50, 51], did not arise in [34].

The work in [10] studies load-balancing for parallel search on SMP. They achieve a better performance than when using in-order variable selection heuristic, a heuristic which is usually rather weak. Unlike in [10], in [50] we study the performance against random ordering and without using a timeout. Random ordering, unlike other ordering heuristics, does not risk getting caught in unproductive parts of the search tree every time an experiment is run.

In [10] an advanced estimate of work-size is used. However, in our previous work, we showed that simple measurements are often equivalent or even preferable to more advanced ones [50]. Moreover, we tested our scalability further in [50], as there are rarely any major differences between load-balancing methods when only using eight processors.

4 Parallelism in Distributed Constraint Programming

Parallelism is a natural component of DCP. Running each agent on a separate core allows parallelism of consistency. If using asynchronous search [69], the search will also be parallel. This makes DCP well suited to illustrate the parallelism in the program, while still letting the constraint framework handle all the difficult parallel programming aspects.

The limited models and the number of messages sent between solvers in traditional DisCSP makes it prohibitive for large problems [59]. In our version of DCP, we use more efficient communication and allow global constraints.

Figure 3.7 depicts a simplified view of the distributed constraint evaluation process and the search we currently use. All time consuming steps in our solving are parallel. As depicted in Figure 3.7, we evaluate consistency and vote on the next master in parallel. But in order to guarantee synchronicity, we must wait for all prunings to be finished before we can move on to selecting the next master. Hence, we move from synchronous to asynchronous execution of the agents, and back again, with every assignment.

4.1 Related Work

Most work on DCP deals with the scenario where each agent holds a single variable and only binary constraints exist in the problem model [59]. In [54], we use global constraints. As far as we know, this is the only paper to do so in DCP. Soon, Frodo [30] will be supporting global constraints through our solver [63].
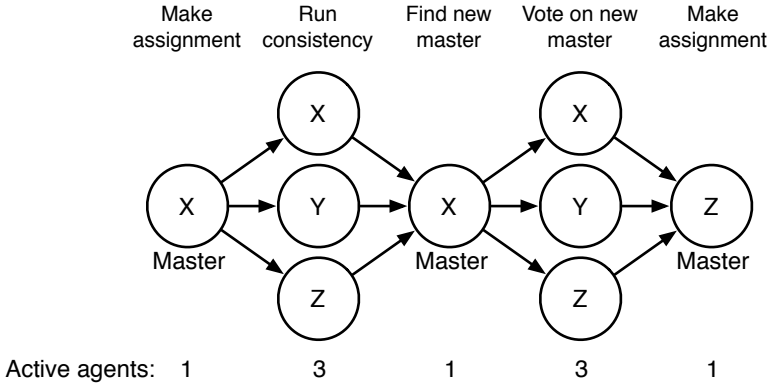
Figure 3.7: The progress from one assignment to the next. X, Y, and Z are agents.

Many DCP systems such as [30], model problems using table constraints that enumerate all possible assignments. This enumeration is infeasible for large problems. In [54] we use a full constraint solver that evaluates constraints without any need for enumeration of possible assignments. In [16], models could be written using standard constraints. However, these constraints are then expanded into table form before the solving begins.

One of the few related works on DCP is [32], in which a similar definition of the DCP model is used. However, the similarities end there, as [32] makes no mention of global constraints. Without global constraints, such as those we use in [54], complex problems modeled in CP can rarely be solved efficiently.

Perhaps the most related work is [7]. It brings up the advantages of collecting complex subproblems onto a few agents. These subproblems can then be solved more efficiently. However, [7] specifically mentions that global constraints would be very good, but this is never implemented, tested, or expanded upon.

Another major difference between our model and previous work on DCP, such as [6, 16, 19, 32, 55, 69, 70], is that we communicate entire domains. This is much more efficient than sending one value from a domain at a time and getting a *Good* or *NoGood* message back.

Another contribution of [54] is to introduce advanced search to DCP. We can add new constraints during search to create an ordering of variables before we start making assignments. These constraints can be communicated to the other solvers. This type of ordering is necessary to solve complex scheduling problems efficiently using DCP [54]. Ordering like this is often not possible in other DisCSP solvers.

5   The Future

Lately, a good deal of research has been performed on parallelism in SAT [21] and parallelism has strong foundations in integer programming [44]. Work on, e.g., exploration strategies [24], are sure to be developed for CP as well. We refer the reader to [14] for more information.

In the future, mixes of CP, IP, and SAT are likely to appear. Some solvers already mix these methods to some extent [62]. Running three different solving methods in parallel might lead to major performance increases.

Most current work on parallelism in CP is still focused on parallel search [1, 35, 41, 67]. Many of these papers show a somewhat limited performance, or have not studied the performance on larger architectures. This is unfortunate, since it is rarely possible to test scalability when using few processors.

The increased publication of papers on parallelism in constraint programming over the past few years is encouraging. Taking such work further will benefit the community. However, as far as we know, few papers about parallel CP with global constraints have been published in the parallelism community apart from ours.

CONTRIBUTIONS

In this thesis, we present the contributions below. In all the included papers, my supervisor Krzysztof Kuchcinski has helped with the writing and through discussions. Otherwise, all work has been performed by me. This includes choice of topic, design and development of software components, and experimental evaluations.

- *Parallel consistency* is introduced, implemented and evaluated. We have designed and developed the first, and so far only, consistency algorithm that processes global constraints in parallel. Our results show that very good absolute speed-up is possible for problems with models that use many similar-sized large global constraints.

- *The combination of parallel consistency and parallel search* is presented and studied. This is the first and thus far only time that the combination of these two types of parallelism has been studied in constraint programming. Our results indicate that successfully combining them is a difficult problem. We show that a performance increase is possible, compared to when only having one type of parallelism.

- *Relative-measured load-balancing for parallel search* is developed and evaluated. We have created a load-balancing where constraint solvers compete for the right to send work. This competition can use any per-solver estimate or measure of work size that can be partially ordered. Our results show that, for simple measures, higher performance can be achieved than with random polling, and that this benefit increases with the number of solvers.

- *Dynamic trade-off to balance the costs of communication and computation* has been implemented and studied. We have created an algorithm that allows the model of communication to be changed dynamically depending on an estimate of network load. This allows us to reduce the bandwidth demands of parallel solving at the expense of increased computation. It also allows us to reduce the computational needs at the expense of higher bandwidth requirements. Our results show that our algorithm can improve the performance and scalability of both purely binary problem models and problems that also use global constraints.

- *Distributed constraint programming with agents* is introduced, implemented and evaluated. We have designed a completely new architecture for modeling and solving of distributed constraint programming problems. Our architecture allows more memory-efficient modeling, a greater degree of consistency and faster search. We have solved scheduling problems that have never been dealt with in previous work on DisCSP. Our results show that we can efficiently solve difficult scheduling problems using our model of distributed constraint programming.

1   Parallel Consistency

Consistency which processes several global constraints in parallel was first introduced in [52], included in this thesis as Paper I. This paper illustrates the potential of parallel consistency, and how it behaves for consistent and inconsistent problems. We achieved an almost linear speed-up for problems that make heavy use of global constraints.

Our experiments indicate that parallel consistency can give a major speed-up for problems with regular models. One example is sudoku, which uses large global constraints that are all of the same size. Other problems such as n-Queens show a rather lackluster performance increase. The reason is that n-Queens is limited by having only two large constraints which need to be processed several times.

See *Paper I* for the full study and evaluation of parallel consistency.

2 Combining Parallel Consistency and Parallel Search

The second contribution of this thesis is to introduce and study the combination of parallel consistency and parallel search. This is a natural development of parallel consistency. The idea is simple: run parallel search with each solver having a set of threads available for processing consistency. This work was first presented in [53], included in this thesis as Paper II.

Our experiments show that it is difficult to find problems that benefit from having both parallel consistency and parallel search. The reason is that parallel consistency requires large constraints to pay for the cost of synchronization, and it is rarely feasible to solve such large problems completely.

For the complete details and evaluation of the combination of parallel consistency and parallel search, see *Paper II*.

3 Relative-Measured Load-Balancing for Parallel Search

The third contribution of this thesis is relative-measured load-balancing for parallel search. We are not aware of any previous work on parallel search in CP that uses competition-based load-balancing. Usually in CP, we cannot know the exact size of the work a solver can share with other solvers. However, since processors with work to send often compete for free recipient solvers, we can say which computer is *likely* to have the most work to send.

The motivation for using a relative measure is that solvers always compete for resources. If we had an infinite amount of solvers receiving work, we would only care about the accuracy of our estimates, and not send work that is too small. However, we never have an infinite number of machines.

Regardless of the accuracy of an estimate, it may not improve the load-balancing. For example, as depicted in Figure 4.1, assume we have three

solvers that are working and one that is idle. If solver A has a lot less work than solvers B and C, A will probably need less time to estimate the size of its work than B and C need. Hence, since only solver D is free to receive work, A will be able to make it busy before the other solvers have finished estimating their work sizes. This is likely to reduce the overall performance, as we may decrease the average size of work that is shared.
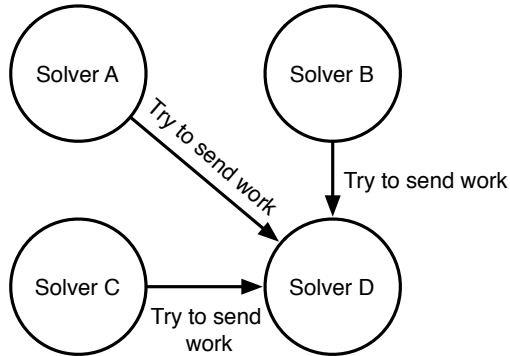
Figure 4.1: An example of competition to send work to a free solver.

During parallel search, competition for sending work to idle solvers often occurs. By using a relative measure of work size, this competition works to our benefit. All estimates of work size have a variance from the correct value. By running a competition between several solvers, we can eliminate the effects that any systematic error in our estimate may cause. Thereby, we increase the accuracy of our estimate, relative to not having a competition.

An example of the procedure for running the competition is shown in Figure 4.2. First, solvers A, B, and C are busy searching. The solvers will ask the idle solver D if it is free to receive work. Once D receives the first $busy?$-message, it waits for n milliseconds before replying, to let more work-sending requests arrive. Finally, D tells the solver with the largest estimate of work size that D is not busy. D replies to all other solvers that it is busy.
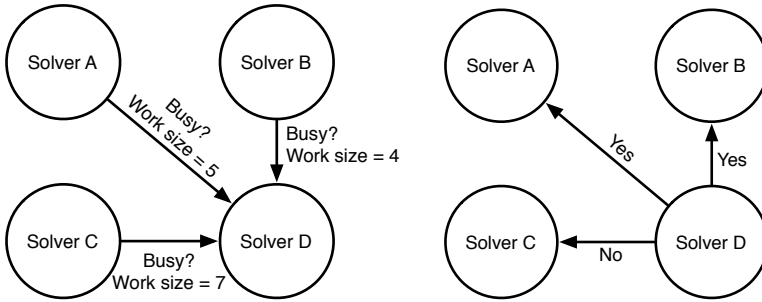
(a) Solvers ask D if it is free, telling D what work size they have estimated. D waits for n milliseconds from the first request, to let more work sharing requests arrive.

(b) D answers Yes to the solver with the most work, No to all other requesting solvers

Figure 4.2: The operating sequence for the competition in relative-measured load-balancing.

Our relative-measured load-balancing works on a higher level than all load-balancing methods that only use absolute measures to determine whether the work is large enough to send or not. Our model of the competition can be run on any estimate of work size. This holds, regardless of if it is simple measurements, as in [50], or complex ones, as in [10]. Our arbiter can decide that no solver has enough work to be allowed to send, and tell everyone it is busy.

Our experiments show that an increased performance can be achieved by using relative-measured load-balancing. Furthermore, in our experiments, the benefit increases with the number of solvers. The more solvers are used, the more competition there will be. Even though we used simple measures, we still managed to increase the performance by up to 20 % compared to random polling.

For the full experimental evaluation of relative-measured load-balancing, see *Paper III*.

4 Dynamic Trade-off to Balance the Costs of Communication and Computation

The fourth contribution of this thesis is the development and evaluation of an algorithm that can dynamically balance computation and communication during parallel search. Parallel search in CP has two main modes of communication, copying and recomputation. Our algorithm switches between these two in order to avoid congestion in the communication network. This is somewhat similar to what has been done for sequential solving with copying and recomputation [48].

We studied both problems with global constraints and problems that only use binary constraints. Unlike previous work, we also studied optimization problems. For all problems, and almost all different numbers of machines, our dynamic choice of communication mode outperformed pure copying or recomputation.

The algorithm we have developed allows a higher performance to be achieved for parallel search in CP on clusters. For many problem instances, we shorten the execution time while lowering the processor load compared to the best performing alternative. Hence, the cluster can be used more efficiently to solve several problems at the same time.

Our full study of the trade-off between communication and computation are presented in *Paper IV*.

5 Distributed Constraint Programming with Agents

The fifth contribution of this thesis is the introduction of distributed constraint programming (DCP). We contrast this to DisCSP, since our model equips each agent with a full constraint solver.

Unlike any work we are aware of, we can solve complex standard scheduling benchmark problems in reasonable time. Previous research has usually focused on one variable per agent and only binary constraints. We show that a more complex model, with more powerful agents, is necessary to solve complex problems.

In our comparison, our model of DCP outperforms traditional DisCSP by orders of magnitude. We have achieved this by empowering our agents with global constraints and advanced search, something that we have never seen before in work on distribution with agents in CP.

The full study of DCP with experimental results is presented in *Paper V* of this thesis.

CONCLUSIONS

The main conclusion of this thesis is that parallelism in constraint programming has great potential. The extensions we have implemented in our solver provide several kinds of parallelism. These are available automatically, with hardly any modification of the problem specification. In particular, no knowledge of parallel programming is needed to achieve good parallel performance.

We can also conclude that parallel consistency can provide excellent performance. This is especially important for problems that do not benefit much from parallel search. Our work is the first to provide parallel consistency with global constraints, this makes the task granularity more manageable.

In this thesis we also present the first combination of parallel consistency and parallel search. Our conclusion is that some problems benefit from using both these types of parallelism. We also conclude that many problems benefit much more from one type of parallelism than the other. Letting computing resources switch dynamically between these two types may provide more robust parallel performance for many problems.

From our work on relative-measured load-balancing, we draw the conclusion that the competition for computing resources can be used to increase performance. By letting solvers compete with each other, we can partly overcome the problem of not knowing the exact size of the work that solvers send. In our load-balancing, we can use any estimate of work size that can be partially ordered.

We have developed an algorithm for dynamically choosing the method of communication depending on the network load. We conclude that this type of dynamic adaptation can increase performance of parallel solving.

Even with simple measures, approximations of bottlenecks can allow the computing resources to be used more efficiently.

We have presented the first model of distributed constraint programming that uses global constraints and solves complex scheduling problems. Our model enables performance for these problems that has never been achieved through traditional approaches to DisCSP. This is partly because we can run ordering before search, and partly because global constraints provide much more efficient pruning. We conclude that our model of DCP solves many of the inherent issues of DisCSP. This makes our solution much more viable for solving complex real-world problems.

Future Work

There are several future approaches we would like to see. The first one is parallel consistency algorithms for global constraints. This is an approach that may provide more robust speed-up than our parallel consistency. However, these algorithms have to be constraint specific. This makes such solutions problem specific, something which may reduce the automaticity of the parallelism.

Another interesting development is the use of different exploration heuristics. Exploring the same search space with, for example, different variable ordering sometimes provides a speed-up. This approach has been quite successful in SAT. Similar use of heuristics is likely to be beneficial to parallel solving in CP as well. Using SAT to facilitate back-jumping in CP is already appearing [62].

One major area of future research is dynamic solving that adapts to the search space during solving. There is much work to be done on, e.g, dynamically selecting consistency algorithms, changing the type of search, allocating computing resources to different types of parallelism etc. This type of adaptation may also include such modifications as switching to SAT solving for a part of the problem. This would further the integration of CP, integer programming, and SAT.

BIBLIOGRAPHY

[1] D. Allouche, S. de Givry, and T. Schiex. Towards parallel non serial dynamic programming for solving hard weighted CSP. In D. Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 53–60. Springer Berlin / Heidelberg, 2010.

[2] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 30:483–485, 1967.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46:720–748, Sept 1999.

[5] S. Boivin, G. Pessant, and B. Gendron. Parallelizing global constraints. *CORS-INFORMS International Meeting*, June 2009.

[6] I. Brito. Synchronous, asynchronous and hybrid algorithms for DisCSP. In M. Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 791–791. Springer Berlin / Heidelberg, 2004.

[7] D. A. Burke and K. N. Brown. A comparison of approaches to handling complex local problems in DCOP. In *Distributed Constraint Satisfaction Workshop*, pages 27–33, 2006.

[8] A. Casas, M. Carro, and M. V. Hermenegildo. A high-level implementation of non-deterministic, unrestricted, independent AND-parallelism. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 651–666, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63–75, Feb 1985.

[10] G. Chu, C. Schulte, and P. Stuckey. Confidence-based work stealing in parallel constraint programming. In I. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241. Springer Berlin / Heidelberg, 2009.

[11] K. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8:1–49, January 1986.

[12] Comet. http://www.comet-online.org, 2011.

[13] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[14] G. Dequen, P. Vander-Swalmen, and M. Krajecki. Toward easy parallel SAT solving. In *Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '09, pages 425–432, Washington, DC, USA, 2009. IEEE Computer Society.

[15] G. Efthivoulidis, N. Vlassis, P. Tsanakas, and G. Papakonstantinou. An experiment for truly parallel logic programming. *Journal of Intelligent & Robotic Systems*, 16:169–184, 1996.

[16] R. Ezzahir, C. Bessiere, M. Belaissaoui, and E. Bouyakhf. DisChoco: A platform for distributed constraint programming. In *Proceedings of IJCAI-07 Workshop on Distributed Constraint Reasoning*, pages 16–27, 2007.

[17] E. C. Freuder. In pursuit of the holy grail. *Constraints*, 2:57–61, 1997.

[18] Gecode. http://www.gecode.org, 2011.

[19] A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward bounding for distributed COPs. *Journal of Artificial Intelligence Research*, 34:61–88, Feb 2009.

[20] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):28–35, Jan/Feb 1999.

[21] L. Guo, Y. Hamadi, S. Jabbour, and L. Sais. Diversification and intensification in parallel SAT solving. In D. Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 252–265. Springer Berlin / Heidelberg, 2010.

[22] G. Gupta, E. Pontelli, K. A. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel execution of Prolog programs: A survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[23] Z. Habbas, F. Herrmann, P.-P. Merel, and D. Singer. Load balancing strategies for parallel forward search algorithm with conflict based backjumping. In *ICPADS '97: Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pages 376–381, Washington, DC, USA, 1997. IEEE Computer Society.

[24] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: A parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009.

[25] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[26] M. V. Hermenegildo. *An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel*. PhD thesis, The University of Texas at Austin, 1986.

[27] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, July 2003.

[28] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: Design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[29] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22:60–79, July 1994.

[30] T. Léauté, B. Ottens, and R. Szymanek. FRODO 2.0: An open-source framework for distributed constraint optimization. In *Proceedings of IJCAI-09 Workshop on Distributed Constraint Reasoning*, pages 160–164, Pasadena, California, USA, July 2009.

[31] K. Marriott and P. J. Stuckey. *Introduction to Constraint Logic Programming*. MIT Press, Cambridge, MA, USA, 1998.

[32] A. Meisels and R. Zivan. Asynchronous forward-checking for DisCSPs. *Constraints*, 12:131–150, 2007.

[33] L. Michel, A. See, and P. Van Hentenryck. Distributed constraint-based local search. In F. Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, volume 4204 of *Lecture Notes in Computer Science*, pages 344–358. Springer Berlin / Heidelberg, 2006.

[34] L. Michel, A. See, and P. Van Hentenryck. Parallelizing constraint programs transparently. In C. Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 514–528. Springer Berlin / Heidelberg, 2007.

[35] L. Michel, A. See, and P. Van Hentenryck. Parallel and distributed local search in COMET. *Computers & Operations Research*, 36:2357–2375, Aug 2009.

[36] R. Mirchandaney, D. Towsley, and J. A. Stankovic. Analysis of the effects of delays on load sharing. *IEEE Transactions on Computers*, 38:1513–1525, Nov 1989.

[37] G. Mitra, I. Hai, and M. T. Hajian. A distributed processing algorithm for solving integer programs using a cluster of workstations. *Parallel Computing*, 23(6):733–753, 1997.

[38] G. Moore. Progress in digital integrated electronics. In *1975 International Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[39] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer Berlin / Heidelberg, 2007.

[40] T. Nguyen and Y. Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1-2):227–250, 1998.

[41] L. Otten and R. Dechter. Finding most likely haplotypes in general pedigrees through parallel search with dynamic load balancing. In *Pacific Symposium on Biocomputing*, pages 26–37, 2011.

[42] E. Pontelli, G. Gupta, and M. V. Hermenegildo. &ACE: A high-performance parallel Prolog system. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 564–571, Washington, DC, USA, 1995. IEEE Computer Society.

[43] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 359–366, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[44] T. K. Ralphs. Parallel branch and cut. In E.-G. Talbi, editor, *Parallel Combinatorial Optimization*, chapter 3, pages 53–101. John Wiley & Sons, Inc., 2006.

[45] V. Rao and V. Kumar. Superlinear speedup in parallel state-space search. In K. Nori and S. Kumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 338 of *Lecture Notes in Computer Science*, pages 161–174. Springer Berlin / Heidelberg, 1988.

[46] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, AAAI '94, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[47] A. Reinefeld. Parallel search in discrete optimization problems. *Simulation Practice and Theory*, 4(2-3):169–188, 1996.

[48] R. Reischuk, C. Schulte, P. Stuckey, and G. Tack. Maintaining state in propagation solvers. In I. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 692–706. Springer Berlin / Heidelberg, 2009.

[49] C. C. Rolf. Parallel and distributed search in constraint programming. Master Thesis, Department of Computer Science, Lund University, June 2006.

[50] C. C. Rolf and K. Kuchcinski. Load-balancing methods for parallel and distributed constraint solving. In *IEEE International Conference on Cluster Computing*, pages 304–309, Sep/Oct 2008.

[51] C. C. Rolf and K. Kuchcinski. State-copying and recomputation in parallel constraint programming with global constraints. In *Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 311–317, Feb 2008.

[52] C. C. Rolf and K. Kuchcinski. Parallel consistency in constraint programming. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 638–644. CSREA Press, July 2009.

[53] C. C. Rolf and K. Kuchcinski. Combining parallel search and parallel consistency in constraint programming. In *International Conference on Principles and Practice of Constraint Programming: TRICS workshop*, Sept 2010.

[54] C. C. Rolf and K. Kuchcinski. Distributed constraint programming with agents. In A. Bouchachia, editor, *International Conference on Adaptive and Intelligent Systems - ICAIS '11*, volume 6943 of *Lecture Notes in Computer Science*, pages 320–331. Springer Berlin / Heidelberg, 2011.

[55] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier Science Inc., New York, NY, USA, 2006.

[56] C. Roucairol. Parallel processing for difficult combinatorial optimization problems. *European Journal of Operational Research*, 92(3):573–590, 1996.

[57] A. Ruiz-Andino, L. Araujo, F. Sáenz, and J. Ruz. Parallel execution models for constraint programming over finite domains. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 134–151. Springer Berlin / Heidelberg, 1999.

[58] A. Ruiz-Andino, L. Araujo, F. Sáenz, and J. J. Ruz. Parallel arc-consistency for functional constraints. In *Implementation Technology for Programming Languages based on Logic*, pages 86–100, 1998.

[59] M. Salido. Distributed CSPs: Why it is assumed a variable per agent? In I. Miguel and W. Ruml, editors, *Abstraction, Reformulation, and Approximation*, volume 4612 of *Lecture Notes in Computer Science*, pages 407–408. Springer Berlin / Heidelberg, 2007.

[60] C. Schulte. Parallel search made simple. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, editors, *Proceedings of TRICS: Techniques for Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, Singapore, Sept 2000.

[61] C. Schulte and M. Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[62] P. Stuckey. Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. In A. Lodi, M. Milano, and P. Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *Lecture Notes in Computer Science*, pages 5–9. Springer Berlin / Heidelberg, 2010.

[63] R. Szymanek. Private communication, 2011.

[64] S. Taylor. *Parallel logic programming techniques*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[65] R. J. Wallace and E. C. Freuder. Ordering heuristics for arc consistency algorithms. In *Canadian Conference on Artificial Intelligence*, 1992.

[66] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[67] F. Xie and A. Davenport. Massively parallel constraint programming for supercomputers: Challenges and initial results. In A. Lodi, M. Milano, and P. Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *Lecture Notes in Computer Science*, pages 334–338. Springer Berlin / Heidelberg, 2010.

[68] J. Yang and S. D. Goodwin. High performance constraint satisfaction problem solving: State-recomputation versus state-copying. In *Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 117–123, Washington, DC, USA, 2005. IEEE Computer Society.

[69] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3:185–207, 2000.

[70] R. Zivan and A. Meisels. Concurrent search for distributed CSPs. *Artificial Intelligence*, 170(4-5):440–461, 2006.

INCLUDED PAPERS

PARALLEL CONSISTENCY IN CONSTRAINT PROGRAMMING

This paper is a reformatted version of *Parallel Consistency in Constraint Programming*, The International Conference on Parallel and Distributed Processing Techniques and Applications: SDMAS workshop, 2009.

Parallel Consistency in Constraint Programming

Carl Christian Rolf and Krzysztof Kuchcinski

Department of Computer Science, Lund University

Carl_Christian.Rolf@cs.lth.se, Krzysztof.Kuchcinski@cs.lth.se

Abstract

Program parallelization becomes increasingly important when new multi-core architectures provide ways to improve performance. One of the greatest challenges of this development lies in programming parallel applications. Using declarative languages, such as constraint programming, can make the transition to parallelism easier by hiding the parallelization details in a framework.

Automatic parallelization in constraint programming has previously focused on data parallelism. In this paper, we look at task parallelism, specifically the case of parallel consistency. We have developed two models of parallel consistency, one that shares intermediate results and one that does not. We evaluate which model is better in our experiments. Our results show that parallelizing consistency can provide the programmer with a robust scalability for regular problems with global constraints.

1   Introduction

In this paper, we discuss parallel consistency in constraint programming (CP) as a means of achieving task parallelism. CP has the advantage of being declarative. Hence, the programmer does not have to make any significant changes to the program in order to solve it using parallelism. This means that the difficult aspects of parallel programming can be left entirely to the creator of the constraint framework.

Constraint programming has been used with great success to tackle different instances of NP-complete problems such as graph coloring, satisfiability (SAT), and scheduling [5]. A constraint satisfaction problem (CSP) can be defined as a 3-tuple $P = (X, D, C)$, where X is a set of variables, D is a set of finite domains where D_i is the domain of X_i, and C is a set of primitive or global constraints containing between one and all variables in X. Solving a CSP means finding assignments to X such that the value of X_i is in D_i, while all the constraints are satisfied. The tuple P is referred to as a constraint store.

Finding a valid assignment to a constraint satisfaction problem is usually accomplished by combining backtracking search with consistency checking that prunes inconsistent values. To do this, a variable is assigned one of the values from its domain in every node of the search tree. Due to time-complexity issues, the consistency methods are rarely complete [2]. Hence, the domains of the variables will contain values that are locally consistent, but cannot be part of a solution.

In this paper, we refer to parallel search as data parallelism, and parallel consistency as task parallelism. When parallelizing search in CP, the data is split between solvers. As depicted in Figure 1, data parallelism in CP can cause major problems. In the figure, we send the rightmost nodes to another constraint solver running on a different processor core. However, since there are no solutions in those search nodes, the parallelism will inevitably lead to a slowdown because of communication overhead. This problem cannot be avoided, since the consistency algorithms are not complete. Hence, we cannot predict the amount of work sent to other processors.
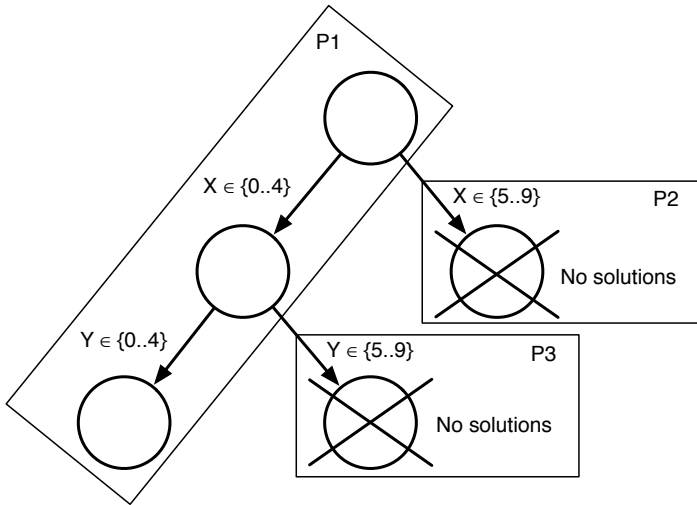
Figure 1: Parallel search in constraint programming.

Figure 2 presents the model of parallel consistency in constraint programming discussed in this paper. In the example, the search process is sequential, but the enforcement of consistency is performed in parallel. Constraints C1, C2, and C3 can be evaluated independently of each other on different processor cores, as long as the changes they try to perform are synchronized. This type of parallelism does not involve splitting data, and will never lead to any unnecessary search. We may, however, have to perform extra iterations of consistency, since the updates to domains are based on the store from the beginning of each consistency phase.

The problem of performing unnecessary work in parallel constraint solving is pervasive. Most problems do not scale well when using many processors. In our previous work [11, 12] we have tried to reduce the cost distributing work, and reduce the probability of performing unnecessary work. However, some problems cannot be data-parallelized at all without causing a severe slowdown, this is true in particular when searching for a single solution.

Data parallelism can be problematic, or even unsuitable, for other reasons. Many problems modeled in CP spend a magnitude more time enforcing consistency than searching. Trying to use data parallelism for these problems often reduces performance. In these cases, task parallelism is the only way to take advantage of modern multicore processors.

The rest of this paper is organized as follows. In Section 2 the background issues are explained, in Section 3 the parallel consistency is described in detail. Section 4 introduces the experiments and the results, Section 5 gathers the conclusions, and Section 6 presents our future work.
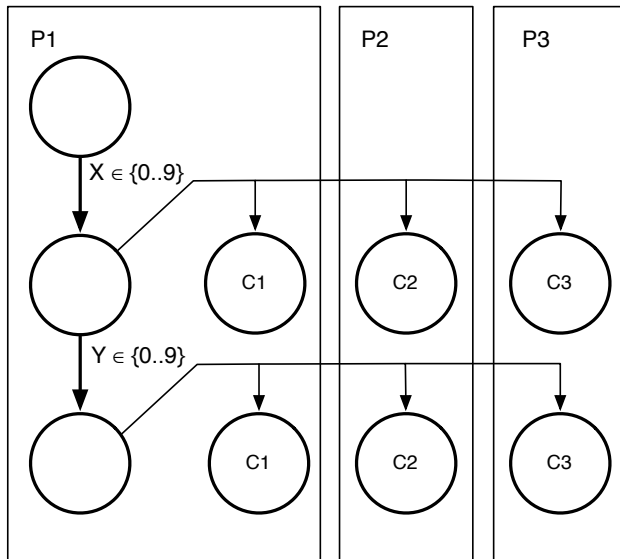
Figure 2: Parallel consistency in constraint programming.

2 Background

Most work on parallelism in CP has dealt with data parallelism [14]. While this offers the greatest theoretical scalability, it is often limited by a number of issues. Today, the main one is that processing disjoint data will saturate the memory bus faster than when processing the same data. In theory, a super-linear performance should be possible for depth-first search algorithms [10]. This, however, has only rarely been reported, and only for small numbers of processors [6]. The performance-limits placed on data-parallel constraint solving are especially apparent on modern multi-core architectures.

Another issue with data parallelism in CP arises for problems modeled using intervals. This category includes scheduling problems, which are the most industry-relevant applications of constraint programming. Splitting an interval in a scheduling problem will reshape the search tree of both the computer sending work and the one receiving it. Such a change in shape can lead to bounding solutions not being found in reasonable time. In the worst case, this can lead to a very large slowdown. Previous work on data parallelism for scheduling problems has either relied on specialized splitting [14], or only reported results for limited discrepancy search and not for depth-first search [8].

Task parallelism is the most realistic type of parallelism for problems where the time needed for search is insignificant compared to that of enforcing consistency. This can happen when the consistency algorithms prunes almost all the inconsistent values. Such strong pruning is particularly expensive and in a greater need of parallelism. The advantage of these large constraints over a massively parallel search is that the execution time will be more predictable.

Previous work on parallel enforcement of consistency has focused on parallel arc-consistency algorithms [7, 13]. The downside of such an approach is that processing one constraint at a time may not allow inconsistencies to be discovered as quickly. If one constraint holds and another does not, the enforcement of the first one could be cancelled as soon as the inconsistency of the second constraint is discovered.

Perhaps the greatest downside of parallel arc-consistency is that it is not applicable to global constraints. These constraints encompass several, or all, of the variables in a problem. This allows them to achieve a much better pruning than primitive constraints that can only establish simple relations between variables, such as $X + Y \leq Z$.

3   Parallel Consistency

Parallel consistency in CP means that several constraints will be evaluated in parallel. Constraints that contain the same variables have data dependencies, and therefore their pruning must be synchronized. However, since the pruning is monotonic, the order in which the data is modified does not affect the correctness. This follows from that well-behaved constraint propagators must be both decreasing and monotonic [15]. In our solver this is guaranteed by the consistency method implemented in our solver. It makes the intersection of the old domain and the one given by the consistency algorithm. The result is written back as a new domain. Hence, the domain size will never increase.

Our model of parallel consistency is depicted in Figure 3. The pseudo-code for our model is presented in Figure 4 and Figure 5. At each level of the search, consistency is enforced. This is done by waking the consistency threads available to the constraint program. These threads will then retrieve constraints from the queue of constraints whose variables have changed. In order to reduce synchronization, each thread will take several constraints out of the queue at the same time. When all the constraints that were in the queue at the beginning of the consistency phase have been processed, all prunings are committed to the constraint store. If there were no changes to any variable, the consistency has reached a fix-point and the constraint program resumes the search. If an inconsistency is discovered, the other consistency threads are notified and they all enter the waiting state after informing the constraint program that it needs to backtrack.

As depicted in Figure 6, we have to stop all thread in order to enforce updates. The reason is that most constraints cannot operate on a partially updated store. However, speculative execution of the constraints already in the queue could reduce the idle time for some threads.
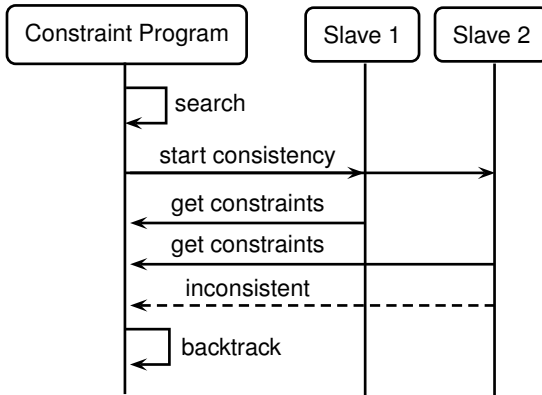
Figure 3: Model of parallel consistency with two consistency threads. The dashed line indicates the final return to the constraint program. In this example it leads to a backtrack of the search procedure.

Consistency enforcement is iterative. When the threads are ready, the constraint queue is split between them. Then one iteration of consistency can begin. This procedure will be repeated until the constraints no longer change the domain of any variable. The constraints containing variables that have changes will be added to the constraint queue after the updates have been performed.

The greatest challenge in parallel consistency lies in distributing the work evenly between the threads. This load-balancing requires a tradeoff between synchronization overhead and an uneven load. The best balance is when each thread has its own local constraint queue, that receives a number of the constraints from the global queue. If a thread runs out of work, it can perform work stealing from another thread without having to lock the global constraint queue.

```
// variables to be labeled V, with FDV x_i ∈ V
// domain of x_i is d_i, list of slave computers S

while V ≠ ∅
  V ← V \ x_i
  select value a from d_i
  x_i ← a
  for each slave s in S
    s.enforceConsistency
  wait  //wait for all slaves to stop
  if Inconsistent
    d_i ← d_i \ a
    V ← V ∪ x_i
return solution
```

Figure 4: The constraint program of parallel depth-first search algorithm.

One of the main implementation issues of parallel consistency is the overhead for synchronization. If this overhead is too high, compared to the time needed to enforce consistency, there will be no speed-up. Furthermore, when starting the consistency, there is additional synchronization needed for waking the threads in the thread pool from the waiting state.

The issue of load-balancing is related to the model in the constraint program. Global constraints usually have consistency algorithms with a time complexity of at least $O(n \log n)$. Primitive constraints, however, typically have a constant running time with regard to the number of variables. While a good load-balancing can alleviate this problem, some problems may simply have too few global constraints to motivate the cost of synchronization in parallel consistency.

There are two variations of the model that we have presented. The difference lies in how the intermediate domains used for updates are handled. The two variations are:

- Shared intermediate domains, which requires synchronization of changes to variables. This variant is described in section 3.1.

- Thread local intermediate domains, which does not require changes to be synchronized, described in section 3.2.

```
// set of constraints to be processed PC
// set of constraints processed in this slave SC
// returns result to the constraint program

while PC ≠ ∅
  PC ← PC \ SC
  while SC ≠ ∅
    SC ← SC \ c
    c.consistency
    if c.inconsistent
      for each slave s in S
        s.stop
      return Inconsistent
    if all other slaves waiting
      perform updates
      for each changed constraint cd
        PC ← PC ∪ cd
      for each slave s in S
        s.wake
    else
      wait  //wait for updates
return Consistent
```

Figure 5: The slave program of parallel depth-first search algorithm.

The domain that is used during update at the barrier in Figure 6 is the intersection of all intermediate domains given by the constraints. If a shared intermediate domain is used, the intersection will be calculated each time a constraint changes a variable. If the intermediate domains are thread local, the intersection will be calculated at the barrier.

3.1 Shared Intermediate Domains

Using shared intermediate domains requires changes to be synchronized. This inevitably reduces the scalability of the parallel consistency. The entire calculation of the domain intersection has to be synchronized, oth-

erwise intervals in the domain could be modified concurrently. Hence, the shared domains cannot be made lock-free, unless the entire domain fits into an architecture-atomic data type.

The advantage of shared intermediate domains, is that inconsistencies will be discovered earlier. The domain used at the update barrier is the intersection of the intermediate domains given by the constraints. Hence, an empty intermediate domain means that the constraint store is inconsistent. If any constraint leads to an empty intermediate domain, we can cancel the enforcement of all other constraints, as the pruning is monotonic.

3.2   Thread Local Intermediate Domains

The principle behind thread local intermediate domains is shown in Figure 7. The downside of using separate intermediate domains is that we will not be able to detect all inconsistencies before the update barrier. Often, inconsistency is reached when the combined changes of two constraints lead to an empty intermediate domain. If we are using thread local intermediate domains, we will only detect such inconsistencies if the two incompatible constraints are enforced by the same thread.
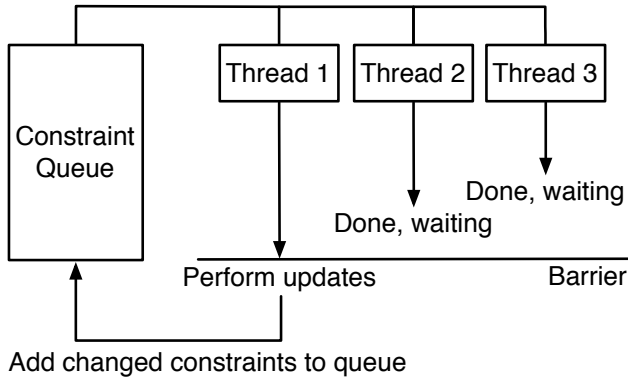
Figure 6: The execution model for parallel consistency.
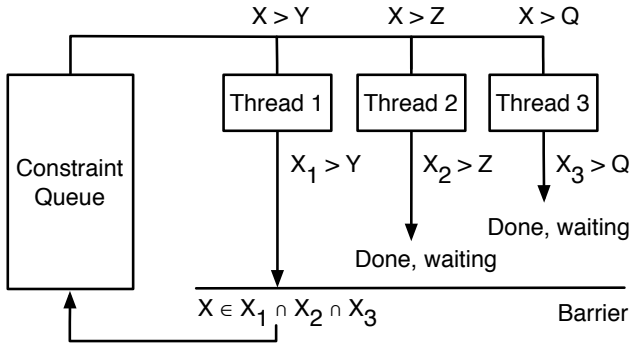
Figure 7: The model of thread local updates.

Thread local variables do not require synchronization, this increases the scalability. In the case of thread local intermediate domains, the only concern is ensuring visibility at the update barrier. This may add extra cost of synchronization depending on which thread performs the actual updates.

If the constraint store is consistent, thread local intermediate domains are preferable. Since there is less synchronization, the scalability will be better, especially when using many consistency threads. However, if the store is inconsistent, we may have to enforce many more constraints since we cannot see the changes caused by the other threads. Inconsistency will therefore be detected later, possibly not until the update barrier is reached.

4   Experimental Results

We used the JaCoP solver [3] in our experiments. The experiments were run on a Mac Pro with two 3.2 GHz quad-core Intel Xeon processors running Mac OS X 10.5. The parallel version of our solver is described in detail in [11].

4.1 Problem Set

We used three problems in our experiments: n-Sudoku, which gives an $n \times n$ Sudoku if the square root of n is an integer, LA31 which is a well-known 30×10 jobshop scheduling problem [4], and n-Queens which consists in finding a placement of n queens on a chessboard so that no queen can strike another. The presented results are the absolute speed-ups of enforcing consistency of all constraints before the search. For Sudoku we used $n = 1024$ and for Queens we used $n = 40\,000$.

The characteristics of the problems are shown in Table 1. n-Sudoku is very regular when modeled in CP, it uses $3 \times n$ alldiff constraints. Our implementation of alldiff uses the $O(n^2)$ algorithm for bounds consistency [9]. LA31 was formulated using ten cumulative constraints, which also have a time complexity of $O(n^2)$ [1]. However, this problem also contains a number of primitive constraints for task precedence. Queens was formulated using three alldiff constraints, combined with a large number of primitive constraints to calculate the diagonals of each queen.

Table 1: Characteristics of the problems.

Problem	Variables	Primitive Constraints	Global Constraints
Sudoku	1048576	0	3072
LA31	632	301	10
Queens	119998	79998	3

4.2 Results for a Consistent Store

We performed experiments on both variations of parallel consistency. The results for shared intermediate domains are presented in Table 2 and Figure 8. The results for thread local intermediate domains are presented in Table 3 and Figure 9. From the tables we can see that the scheduling problem of LA31 is quite small compared to Queens and Sudoku. However, we wanted to use a standardized test for this industry-relevant problem instead of generating a new one.

Table 2: Execution times in milliseconds for shared intermediate domains.

Problem	Threads			
	1	2	4	8
Sudoku	10991	5524	3108	1843
LA31	84.66	50.92	32.44	27.22
Queens	33428	19419	14928	14420

Table 3: Execution times in milliseconds for thread local intermediate domains.

Problem	Threads			
	1	2	4	8
Sudoku	10991	5541	3161	1897
LA31	84.66	47.05	33.22	26.98
Queens	33428	18413	14729	14477

Figure 8 and Figure 9 show that Sudoku is the problem that scales the best by far. This is because it is very regular. The constraints used in this problem are all of the same size, which makes it easy to achieve a good load-balancing. Moreover, all constraints contain 1024 variables, making them very expensive to compute. In contrast, the other problems use combinations of large and small constraints, which makes it difficult to distribute the load evenly.

The scheduling problem of LA31 does not scale as well as Sudoku. The two main reasons are problem size and a lack of large constraints. The short execution time of enforcing consistency increases the relative cost of synchronization. Furthermore, the global constraints in LA31 contain much fewer variables than the ones in Sudoku.

Clearly Queens does not scale well at all, but as we can see in Tables 2 and 3, this is not because of problem size. The low scalability is instead caused by the constraints. There are only three global constraints used in this problem. The rest are small, primitive constraints that finish quickly.
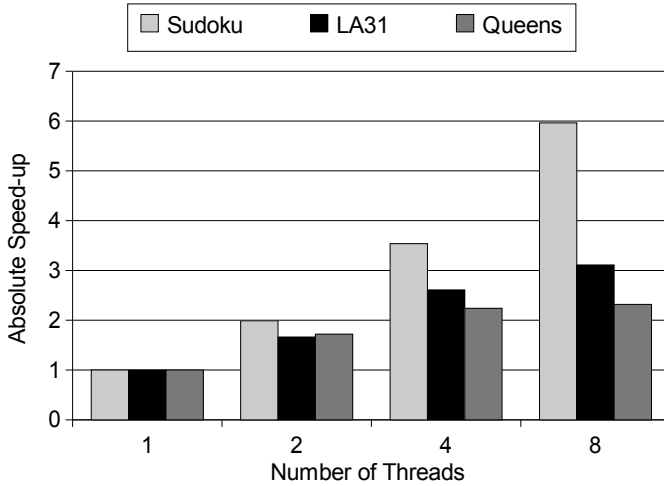
Figure 8: Absolute speed-up when using shared intermediate domains.

Hence, we will have at most three threads running heavy consistency algorithms. LA31 scales better than Queens, despite its short running time, since it contains more global constraints.

The lack of speed-up for Queens compared to Sudoku relates not only to the load-balancing during consistency, but also the consistency iterations. Since the three global constraints in Queens are much larger than the ones in Sudoku, it would not be unreasonable to expect a speed-up of about three for Queens. However, the updates caused by primitive constraints, require the global constraints to be enforced a second time. Hence, the pruning pattern of a problem can have a large negative impact on performance.

The small difference between using shared and thread local intermediate domains is noteworthy. The minimal differences suggests that most of the locks are uncontended. The cases where shared domains are faster are probably caused by the operating system scheduler.
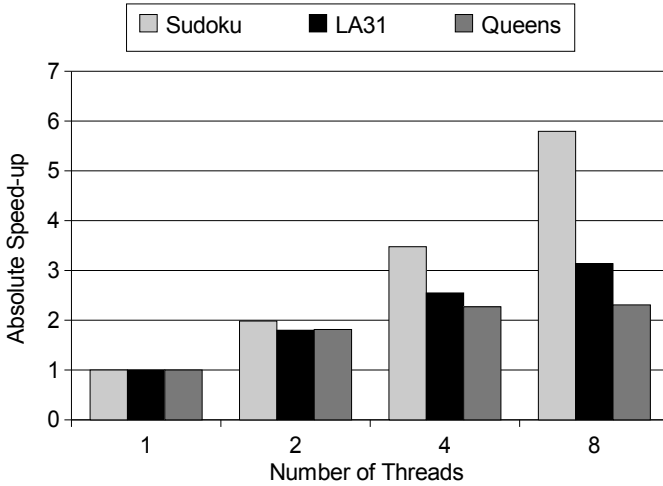
Figure 9: Absolute speed-up when using thread local intermediate domains.

Clearly it does not matter which model of parallel consistency is chosen when the store is consistent. The closer the store is to global consistency, the less pruning there will be. The less pruning, the fewer the dependencies are caused by the update of intermediate domains, reducing lock contention.

4.3 Results for an Inconsistent Store

During search, the store is likely to become inconsistent more often than consistent. Hence, we also performed experiments on an inconsistent store. In order to make the store inconsistent, we made two incompatible assignments and then enforced consistency.

The execution times in milliseconds of the two models are presented in Table 4 and Table 5. The absolute speed-ups are depicted in Figure 10 and Figure 11. Clearly, the scalability of parallel consistency is not as good if the store is inconsistent.

Table 4: Execution times in milliseconds for shared intermediate domains.

Problem	Threads			
	1	**2**	**4**	**8**
Sudoku	7342	5503	3018	1703
LA31	69	58	38	36
Queens	9709	5442	5127	5312

Table 5: Execution times in milliseconds for thread local intermediate domains.

Problem	Threads			
	1	**2**	**4**	**8**
Sudoku	7342	5599	2994	1875
LA31	69	59	41	37
Queens	9709	5750	5370	5547

Table 6 and Table 7 present the behavior of the two model variations. As expected, many more constraints are evaluated when using parallel consistency. Furthermore, the performance is completely determined by the order in which the constraints are evaluated. Ideally the constraints should be ordered by the probability of causing an inconsistency.

Table 6: Constraints evaluated by the shared intermediate domains.

Problem	Threads			
	1	**2**	**4**	**8**
Sudoku	2049	3073	3073	3073
LA31	1749	2941	2941	2941
Queens	3	80002	16610	14873

Table 7: Constraints evaluated by the thread local intermediate domains.

Problem	Threads			
	1	2	4	8
Sudoku	2049	3072	3072	3072
LA31	1749	2981	2981	2981
Queens	3	80001	12310	14041
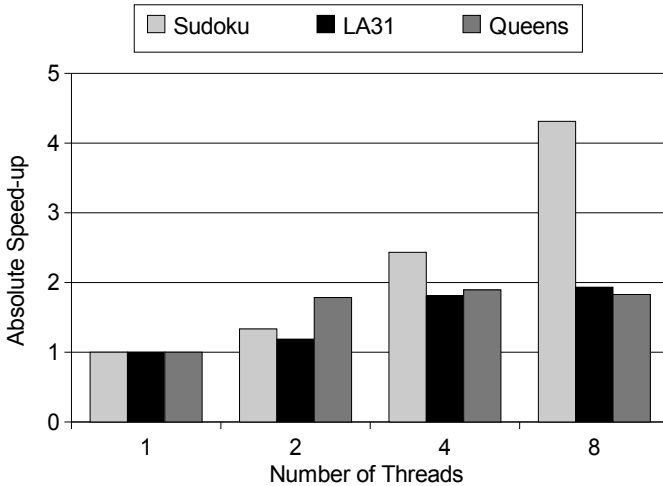
Figure 10: Absolute speed-up when using shared intermediate domains.

The reason why the scalability is lower when the store is inconsistent is that we base our computations on the store at the beginning of the consistency phase. Hence, even with shared intermediate domains, the pruning will not be as strong per consistency iteration as when using sequential consistency.
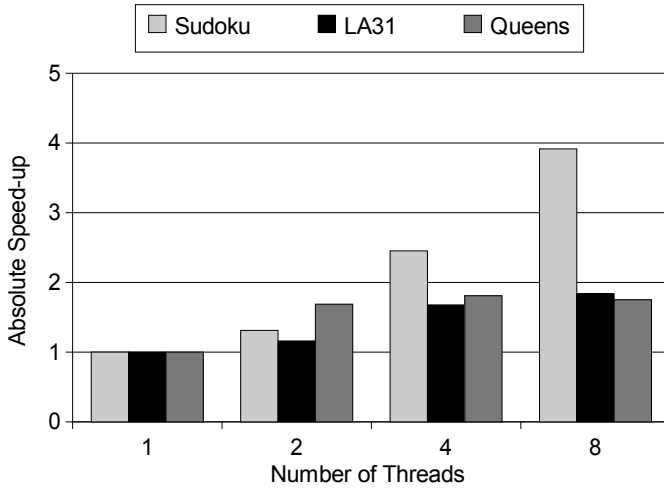
Figure 11: Absolute speed-up when using thread local intermediate domains.

4.4   Processor Load

As depicted in Figures 12 to 14, the processor loads for the three problems are quite different. The biggest difference is that the problems need a different amount of consistency iterations. Sudoku performs no pruning and needs only one iteration of consistency. LA31 needs 12 iterations, hence the heavily varying curve in Figure 13. Queens needs two iterations, which is the cause of the spike in Figure 14.

The average load of the problems is presented in Table 8. The load of LA31 is quite low despite the fact that there are more global constraints than available threads. The main cause is the large number of consistency iterations. In order to enforce the updates, we have to perform twelve barrier synchronizations, at which no consistency threads are active.
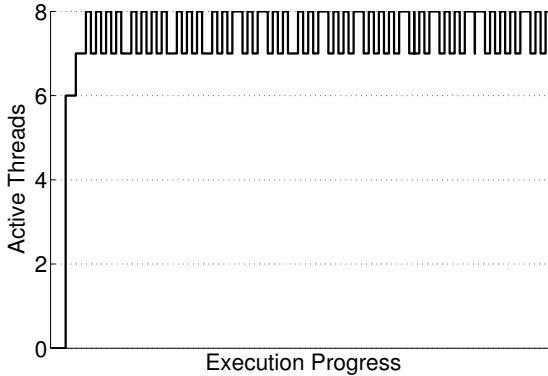
Figure 12: The processor load of Sudoku using eight threads.

The reason why Queens has such a low load is that there are few global constraints. Given the time complexity, the three alldiff constraints will take several orders of magnitude longer to compute than the combined time for the primitive constraints. In the second iteration of consistency, the load comes from the two alldiff constraints used to calculate the diagonals.

Table 8: Average load when using eight threads.

Problem	Average Load	Percentage of Maximum
Sudoku	6.77	0.85
LA31	2.13	0.27
Queens	1.71	0.21

From the average load it is clear that the performance of parallel consistency depends heavily on achieving a good load distribution. Unfortunately, the problem structure may not allow for the load to be shared using only task parallelism. In the case of Queens, a parallel consistency algorithm for alldiff would be necessary to improve the scalability.
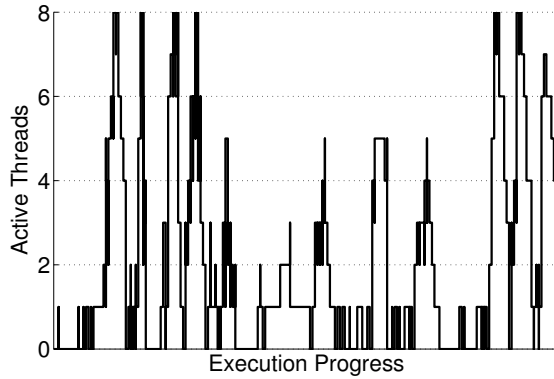
Figure 13: The processor load of LA31 using eight threads.
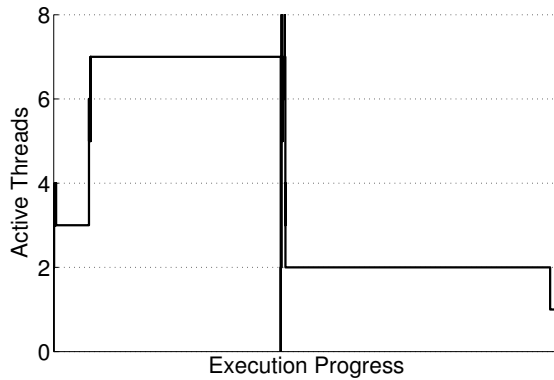
Figure 14: The processor load of Queens using eight threads.

5   Conclusions

The main conclusion of this paper is that task parallelism, in the form of parallel consistency, can offer great improvements in performance. The prerequisite is that the problem is formulated using many global constraints. For problems that consist mainly of primitive constraints, that are easily enforced, the scalability can be severely limited.

Depending on the load-balancing used in the consistency threads, the regularity of the problem has a large impact on the scalability. The more regular the problem, the less of an issue load-balancing becomes. Sudoku is an example of a problem that is both regular and consists only of global constraints. Hence, this problem illustrates the upper bound of the scalability of parallel consistency.

The synchronization cost limits which problems can benefit from parallel consistency. Problems that mostly consist of small constraints will not scale well since even the locking in a thread pool is to costly compared to the performance benefits.

Clearly there is little difference between the two variations of our model of parallel consistency. Reducing synchronization by using thread local intermediate domains will most likely give a better scalability when using many threads. However, which model is better depends on the problem, and how often the constraint store becomes inconsistent.

6   Future Work

In our future work we hope to investigate the possibility of speculative execution. The last iteration of consistency will not make changes to any domain. Hence, speculative execution of the last iteration will always be successful.

We also hope to improve the load-balancing by implementing work stealing. This will alleviate some of the issues that occur for problems with irregular constraints. However, this may not prevent the extra updates caused by the primitive constraints.

The problems that show poor scalability in our experiments are those that often need a greater amount of search. Such problems would benefit primarily from data parallelism. However, parallel consistency could be used to increase the scalability when the memory bus starts to get congested.

References

[1] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[2] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[3] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, July 2003.

[4] S. R. Lawrence. Resource-constrained project scheduling: An experimental investigation of heuristic scheduling techniques. Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh PA, 1984.

[5] K. Marriott and P. J. Stuckey. *Introduction to Constraint Logic Programming*. MIT Press, Cambridge, MA, USA, 1998.

[6] L. Michel, A. See, and P. Van Hentenryck. Parallelizing constraint programs transparently. In C. Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 514–528. Springer Berlin / Heidelberg, 2007.

[7] T. Nguyen and Y. Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1-2):227–250, 1998.

[8] L. Perron. Search procedures and parallelism in constraint programming. In J. Jaffar, editor, *Principles and Practice of Constraint Programming - CP 1999*, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360. Springer Berlin / Heidelberg, 1999.

[9] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 359–366, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[10] V. Rao and V. Kumar. Superlinear speedup in parallel state-space search. In K. Nori and S. Kumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 338 of *Lecture Notes in Computer Science*, pages 161–174. Springer Berlin / Heidelberg, 1988.

[11] C. C. Rolf and K. Kuchcinski. Load-balancing methods for parallel and distributed constraint solving. In *IEEE International Conference on Cluster Computing*, pages 304–309, Sep/Oct 2008.

[12] C. C. Rolf and K. Kuchcinski. State-copying and recomputation in parallel constraint programming with global constraints. In *Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 311–317, Feb 2008.

[13] A. Ruiz-Andino, L. Araujo, F. Sáenz, and J. J. Ruz. Parallel arc-consistency for functional constraints. In *Implementation Technology for Programming Languages based on Logic*, pages 86–100, 1998.

[14] C. Schulte. Parallel search made simple. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, editors, *Proceedings of TRICS: Techniques for Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, Singapore, Sept 2000.

[15] C. Schulte and M. Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook*

of Constraint Programming, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

Combining Parallel Search and Parallel Consistency in Constraint Programming

This paper is a reformatted version of *Combining Parallel Search and Parallel Consistency in Constraint Programming*, International Conference on Principles and Practice of Constraint Programming: TRICS workshop, 2010.

Combining Parallel Search and Parallel Consistency in Constraint Programming

Carl Christian Rolf and Krzysztof Kuchcinski

Department of Computer Science, Lund University

Carl_Christian.Rolf@cs.lth.se, Krzysztof.Kuchcinski@cs.lth.se

Abstract

Program parallelization becomes increasingly important when new multi-core architectures provide ways to improve performance. One of the greatest challenges of this development lies in programming parallel applications. Declarative languages, such as constraint programming, can make the transition to parallelism easier by hiding the parallelization details in a framework.

Automatic parallelization in constraint programming has mostly focused on parallel search. While search and consistency are intrinsically linked, the consistency part of the solving process is often more time-consuming. We have previously looked at parallel consistency and found it to be quite promising. In this paper we investigate how to combine parallel search with parallel consistency. We evaluate which problems are suitable and which are not. Our results show that parallelizing the entire solving process in constraint programming is a major challenge as parallel search and parallel consistency typically suit different types of problems.

1 Introduction

In this paper, we discuss the combination of parallel search and parallel consistency in constraint programming (CP). CP has the advantage of being declarative. Hence, the programmer does not have to make any significant changes to the program in order to solve it using parallelism. This means that the difficult aspects of parallel programming can be left entirely to the creator of the constraint framework.

Constraint programming has been used with great success to tackle different instances of NP-complete problems such as graph coloring, satisfiability (SAT), and scheduling [4]. A constraint satisfaction problem (CSP) can be defined as a 3-tuple $P = (X, D, C)$, where X is a set of variables, D is a set of finite domains where D_i is the domain of X_i, and C is a set of primitive or global constraints containing several of the variables in X. Solving a CSP means finding assignments to X such that the value of X_i is in D_i, while all the constraints are satisfied. The tuple P is referred to as a constraint store.

Finding a valid assignment to a constraint satisfaction problem is usually accomplished by combining backtracking search with consistency checking that prunes inconsistent values. In every node of the search tree, a variable is assigned one of the values from its domain. Due to time-complexity issues, the consistency methods are rarely complete [2]. Hence, the domains will contain values that are locally consistent, i.e., they will not be part of a solution, but we cannot prove this yet.

Figure 1 illustrates the problem of parallelism in CP. We use three processors: P1, P2, and P3 to find the solution. We assign the different parts of the search tree to processors as in the figure. The solution we are searching for is in the leftmost part of the search tree in Figure 1(a) and will be found by processor P1. Any work performed by processor P2 and P3 will therefore prove unnecessary and will only have added communication overhead. In this case, using P2 and P3 for parallel consistency will be much more fruitful. On the other hand, in Figure 1(b), the solution is in the rightmost part of the tree. Hence, parallel search can reduce the total amount of nodes explored to less than a third. In this situation, parallel consistency can still be used to further increase the performance.
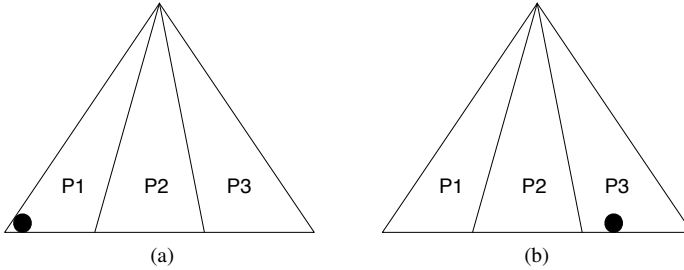
Figure 1: The position of the solution in a search tree affects the benefit of parallelism.

In this paper, we refer to parallel search (OR-parallelism) as data parallelism and parallel consistency (AND-parallelism) as task parallelism. Parallelizing 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 computer handles $X_i \geq \frac{min(X_i)+max(X_i)}{2}$. An example of such data parallelism in CP is depicted in Figure 2. The different possible assignments are explored by processors P1, P2, and P3. Clearly, we are not fully utilizing all three processors in this example. At the first level of the search tree, only two out of three processors are active. Near the leafs of the search tree, communication cost outweighs the benefit of parallelism. Hence, we often have a low processor load in later part of the search.

Figure 3 presents the model of parallel consistency in constraint programming which we will partly discus in this paper. In the example, the search process is sequential, but the enforcement of consistency is performed in parallel. Constraints C1, C2, and C3 can be evaluated independently of each other on different processors, as long as their pruning is synchronized. We do not share data during the pruning, hence, we may have to perform extra iterations of consistency. The cause of this implicit data dependency is that global constraint often rely on internal data-structures that become incoherent if variables are modified during consistency.
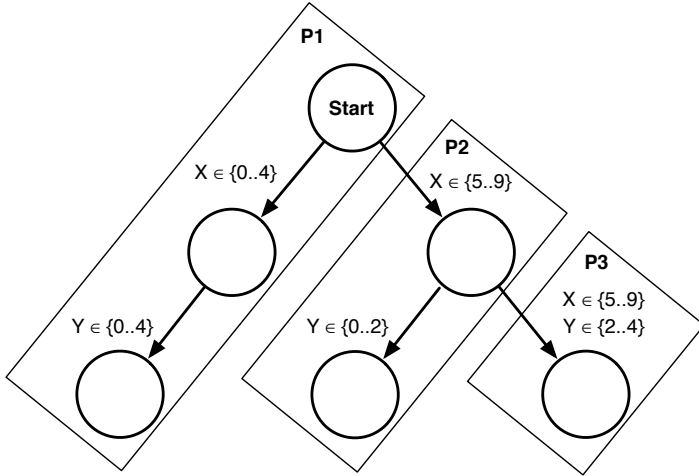
Figure 2: Parallel search in constraint programming.

The problem of idle processors during the latter parts of the search is pervasive [9, 1]. Regardless of the problem, the communication cost will eventually become too big.

Data parallelism can be problematic, or even unsuitable, for other reasons. Many problems modeled in CP spend a magnitude more time enforcing consistency than searching. Using data parallelism for these problems often reduces performance. In these cases, task parallelism is the only way to take advantage of multicore processors.

By combining parallel consistency with parallel search, we can further boost the performance of constraint programming.

The rest of this paper is organized as follows. In Section 2 the background issues are explained, in Section 3 the parallel consistency is described. Section 4 details how we combine parallel search and parallel consistency. Section 5 describes the experiments and the results, Section 6 gathers our conclusions.
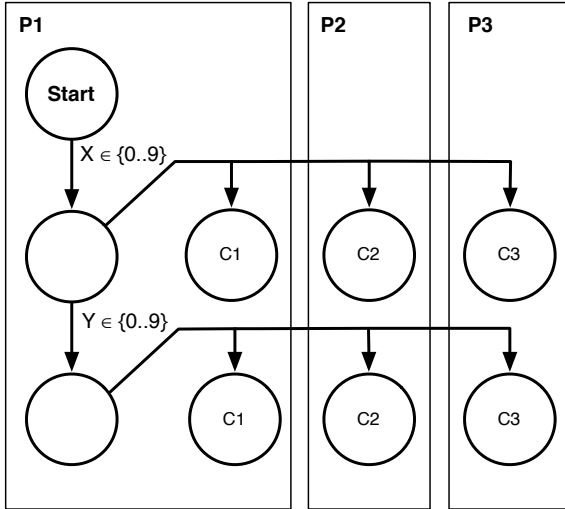
Figure 3: Parallel consistency in constraint programming.

2 Background

Most work on parallelism in CP has dealt with parallel search [12, 5]. While this offers the greatest theoretical scalability, it is often limited by a number of issues. Today, the main one is that processing disjoint data will saturate the memory bus faster than when processing the same data. In theory, a super-linear performance should be possible for depth-first search algorithms [8]. This, however, has only rarely been reported, and only for small numbers of processors [5]. The performance-limits of data parallelism in memory intense applications, such as CP, are especially apparent on modern multi-core architectures [14].

Task parallelism is the most realistic type of parallelism for problems where the time needed for search is insignificant compared to that of enforcing consistency. This happens when the consistency algorithms prune almost all of the inconsistent values. Such strong pruning is particularly expensive and in a greater need of parallelism. The advantage of these

large constraints over a massively parallel search is that the execution time may become more predictable. For instance, speed-up when searching for one solution often has a high variance when parallelizing search since the performance is highly dependent on which domains are split.

Previous work on parallel enforcement of consistency has mostly focused on parallel arc-consistency algorithms [6, 11]. The downside of such an approach is that processing one constraint at a time may not allow inconsistencies to be discovered as quickly as when processing many constraints in parallel. If one constraint holds and another does not, the enforcement of the first one can be cancelled as soon as the inconsistency of the second constraint is discovered.

The greatest downside of parallel arc-consistency is that it is not applicable to global constraints. These global constraints encompass several, or all, of the variables in a problem. This allows them to achieve a much better pruning than primitive constraints, which can only establish simple relations between variables, such as $X + Y \leq Z$.

We only know of one paper on parallel consistency with global constraints [10]. That paper reported a speed-up for problems that can be modeled so that load-balancing is not a big issue. For example, Sudoku gave a near-linear speed-up. However, in this paper we go further by looking at combining parallel search with parallel consistency.

3   Parallel Consistency

Parallel consistency in CP means that several constraints will be evaluated in parallel. Constraints that contain the same variables have data dependencies, and therefore their pruning must be synchronized. However, since the pruning is monotonic, the order in which the data is modified does not affect the correctness. This follows from the property that well-behaved constraint propagators must be both decreasing and monotonic [13]. In our finite domain solver this is guaranteed since the implementation makes the intersection of the old domain and the one given by the consistency algorithm. The result is written back as a new domain. Hence, the domain size will never increase.

Our model of parallel consistency is depicted in Figure 4, this model is described in greater detail in [10], in Figure 6 and Figure 7. At each level of the search, consistency is enforced. This is done by waking the consistency threads available to the constraint program. These threads will then retrieve work from the queue of constraints whose variables have changed. In order to reduce synchronization, each thread will take several constraints out of the queue at the same time. When all the constraints that were in the queue at the beginning of the consistency phase have been processed, all prunings are committed to the constraint store as the solver performs updates. If there were no changes to any variable, the consistency has reached a fix-point and the constraint program resumes the search. If an inconsistency is discovered, the other consistency threads are notified and they all enter the waiting state after informing the constraint program that it needs to backtrack.
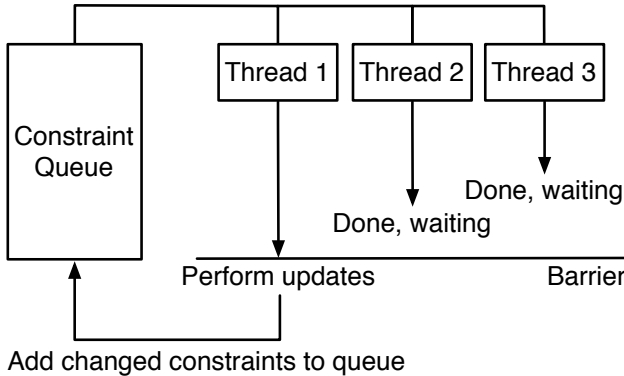
Figure 4: The execution model for parallel consistency.

Consistency enforcement is iterative. When the threads are ready, the constraint queue is split between them, and one iteration of consistency can begin. This procedure will be repeated until we reach a fixpoint, i.e., the constraints no longer change the domain of any variable. The constraints containing variables that have changes will be added to the constraint queue after the updates have been performed.

One of the main concerns in parallel consistency is visibility. Global constraints usually maintain an internal state that may become incoherent if some variables are changed while the consistency algorithm is running. If we perform the pruning in parallel, the changes will only be visible to the other constraints after the barrier. This reduces the pruning achieved per consistency iteration. Hence, in parallel consistency, we will usually perform several more iterations than in sequential consistency before we reach the fixpoint.

4 Combining Parallel Search and Parallel Consistency

The idea when combining parallel search and parallel consistency is to associate every search thread several consistency threads. A simple example is depicted in Figure 5. First the data is split from processor P1 and sent to processor P2. Then the search running on P1 will perform consistency by evaluating constraints C1 and C2 on processors P1 and P3 respectively. The search running on P2 will, completely independently, run consistency using processors P2 and P4. Each search has its own store, hence, constraints C1 and C2 can be evaluated by the two searches without any synchronization.
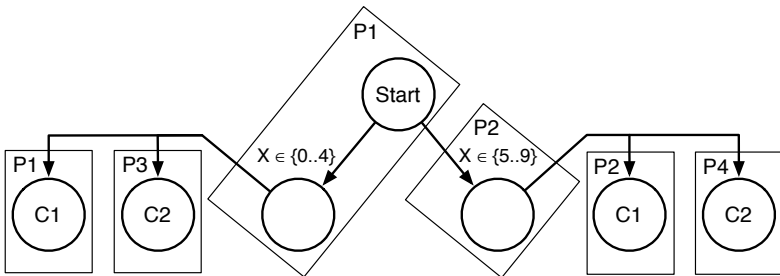
Figure 5: An example of combining parallel search and parallel consistency.

More formally, the execution of the combined search and consistency in CP proceeds as follows. We begin with a constraint store $P = (X, D, C)$ as defined earlier. This gives us a search space to explore, which can be represented as a tree. The children of the root node represent the values in D_i. In these nodes, we assign X_i one of its possible values and remove X_i from X. For example, assigning X_0 the value 5 gives a node n with $P_n = (X \setminus X_0, D \cup D_0 \cap \{5\}, C)$. After each assignment, we apply the the function $enforceConsistency$, which runs the consistency methods of C, changing our store to (X', D', C) where $X' = X \setminus X_i$. D' is the set of finite domains representing the possible values for X' that were not marked as impossible by the consistency methods of C. The method $enforceConsistency$ is applied iteratively until $D'' = D'$. Now there are two possibilities: either $\exists D_i' = \emptyset$, in which case we have a failure, meaning that there are no solutions reachable from this node, or we progress with the search. In the latter case, we have two sub-states. Either $X' = \emptyset$, in which case we have found a solution, or we need to continue recursively by picking a new X_i.

Parallel search means that we divide D_i into subsets and assign them to different processors. Each branch of the search tree starting in a node is independent of all other branches. Hence, there is no data dependency between the different parts of the search space. *Parallel consistency* means parallelizing the $enforceConsistency$ method. This is achieved by partitioning C into subsets, each handled by a different processor.

The pseudo code for our model is presented in Figure 6 and Figure 7. When a search thread makes an assignment it needs to perform consistency before progressing to the next level in the search tree. Hence, processors P1 and P2 in the example are available to aid in the consistency enforcement. The consistency threads are idle while the search thread works. If we only allocate one consistency thread per processor a lot of processors will be idle as we are waiting to perform the assignment. Hence, it is a good idea to make sure that the total number of consistency threads exceeds the number of processors.

As Figure 6 and Figure 7 show, the parallel search threads will remove a search node and explore it. In our model, a search node represents a set of possible values for a variable. The thread that removes this set guarantees

that all values will be explored. If the set is very large, the search thread can split the set to allow other threads to aid in the exploration. When there are no more search nodes to explore, the entire search space has been explored.

Since we have to wait for the different threads, some parts of the algorithm are, by necessity, synchronized. In Figure 6, line 15 requires synchronization while we wait for the consistency threads to finish. In Figure 7, lines 15 to 22, which represent the barrier, are synchronized. However, each thread may use its own lock for waiting. Hence, there is little lock contention. Furthermore, line 13 has to be synchronized in order to halt the other threads when we have discovered an inconsistency. Depending on the data structure, lines 6 and 7 may have to be synchronized.

```
1    // search nodes to be explored N
2    // variables to be labeled V, with FDV x_i ∈ V
3    // domain of x_i is d_i, list of slave computers S
4
5    while N ≠ ∅
6        Node ← N.first
7        N ← N \ Node
8        V ← Node.unlabeledVariables
9        while V ≠ ∅
10           V ← V \ x_i
11           select value a from d_i
12           x_i ← a
13           for each slave s in S
14               s.enforceConsistency
15           wait  //wait for all slaves to stop
16           if Inconsistent
17               d_i ← d_i \ a
18               V ← V ∪ x_i
19       end while
20       store solution
21   end while
```

Figure 6: The depth-first search of the combined parallel search and parallel consistency.

```
1    // set of constraints to be processed PC
2    // set of constraints processed in this slave SC
3    // returns result to the constraint program
4
5    boolean enforceConsistency
6        while PC ≠ ∅
7            PC ← PC \ SC
8            while SC ≠ ∅
9                SC ← SC \ c
10               c.consistency
11               if c.inconsistent
12                   for each slave s in S
13                       s.stop
14                   return Inconsistent
15               if all other slaves waiting
16                   perform updates
17                   for each changed constraint cd
18                       PC ← PC ∪ cd
19                   for each slave s in S
20                       s.wake
21               else
22                   wait  //wait for updates
23           end while
24       end while
25       return Consistent
```

Figure 7: The slave program for parallel consistency of the combined parallel search and parallel consistency algorithm.

4.1 Discussion

An alternative way to combine parallel search and consistency is to use a shared work-queue for both types of jobs. Threads that become idle could get new work from the queue, whether it was running consistency for a constraint or exploring a search space. However, the performance of such an approach would be heavily dependent on the priority given to the dif-

ferent types of work. If the priorities were just slightly incorrect, it would hurt the performance of the other threads. For instance, a thread wanting help with consistency might never get it because the idle threads are picking up search jobs instead. It might be possible to solve this problem using adaptive priorities. However, this is outside the scope of this paper

By combining parallel search and parallel consistency, we hope to achieve a better scalability. Unlike data parallelism for depth-first search, the splitting of data poses a problem in constraint programming. The reason is that the split will affect the domains of the variables that have not yet been assigned a value. In the example in Figure 2, with a constraint such as $X > Y$ the consistency will change the shape of the search tree by removing the value 4 from the domain of Y for processor P1. For more complex problems, the shape of both search trees may be affected in unpredictable ways. Since the consistency methods are not complete, there is no way to efficiently estimate the size and shape of the search trees after a split. Parallel consistency allows us to use the hardware more efficiently when parallel search runs into these kinds of problems.

In [10] we showed that parallel consistency scales best on very large problems consisting of many global constraints. Solving such problems is a daunting task, which makes it hard to combine parallel search with parallel consistency. Furthermore, finding just one solution to a problem often leads to non-deterministic speed-ups.

The biggest obstacle we faced when developing a scalable version of parallel consistency was the cost of synchronization. The problem comes from global constraints, these typically use internal data structures. For instance, the bounds consistency for *AllDifferent* constraint uses a list where the order of variables is given by Hall intervals [7]. If pruning is performed instantly by other threads, instead of being stored until a barrier, the integrity of these data structures may be compromised. Eliminating barrier synchronization would greatly increase the performance of parallel consistency.

5 Experimental Results

We used the JaCoP solver [3] in our experiments. The experiments were run on a Mac Pro with two 3.2 GHz quad-core Intel Xeon processors running Mac OS X 10.6.2 with Java 6. These two processors have a common cache and memory bus for each of its four cores. The parallel version of our solver is described in detail in [9].

5.1 Experiment Setup

We used two problems in our experiments: n-Sudoku, which gives an $n \times n$ Sudoku if the square root of n is an integer and n-Queens which consists in finding a placement of n queens on a chessboard so that no queen can strike another. Both problems use the AllDifferent constraint with bounds consistency [7], chosen since it is the global constraint most well spread in constraint solvers. The characteristics of the problems are presented in Table 1.

The results are the absolute speed-ups when searching for a limited number of solutions to n-Sudoku and one solution to n-Queens. For Sudoku we used $n = 100$ with 85 % of values set and searched for 200 and 5 000 solutions. For Queens we used $n = 550$ and searched for a single solution. We picked these problems in order to illustrate how the size of the search space affects the behavior when combining parallel search with parallel consistency, while still having a reasonable execution time.

For each problem we used between one and eight search threads. For each search thread we used between one and eight consistency threads. We used depth-first search with in-order variable selection for both problems. The sequential performance of our solver is lower than that of some others. However, this overhead largely comes from the higher memory usage of a Java based solver. On a multicore system this is a downside since the memory bus is shared. Hence, lower sequential performance does not necessarily make it simpler to achieve a high speed-up.

Table 1: Characteristics of the problems.

Problem	Variables	Primitive Constraints	Global Constraints
Sudoku	10 000	0	300
Queens	1 648	1 098	3

5.2　Results for Sudoku

The results for 100-Sudoku is presented in Table 2 and Table 3, the speed-ups are depicted in Figure 8 and Figure 9. The bold number in the table indicates the fastest time and the gray background marks the times slower than sequential. The results show that there is a clear difference in behavior as the search space increases. When we have to explore a larger search space, parallel search is better than parallel consistency. However, if we have a more even balance between search and consistency, combining the two types of parallelism increases the performance.

From the diagrams, we can see that it is good to use more consistency threads than there are processor cores. However, using many more threads is not beneficial, especially when there are several search threads.

It is noteworthy that there is little overhead for using parallel consistency when only running one search thread. Search for 200 solutions even increases the performance somewhat. This is important because it means that parallel consistency can be successful when it is difficult to extract data parallelism from the problem.

The reduction in performance when adding parallel consistency to the search for 5 000 solutions comes to some extent from synchronization costs. Synchronization in Java automatically invalidates cache lines that may contain data useful to other threads. With more precise control over cache invalidation, the execution time overhead added by the parallel consistency can be reduced.

Using too many threads will cause an undesirable amount of task-switching and saturation of the memory bus. We measured and analyzed how the number of active threads, and their type, affects performance. The average number of active threads when running two search threads and

four consistency threads per search thread for 200 solutions to n-Sudoku was 5.5. This is the average over the entire execution time. The same number for the slowest instance, eight by eight threads, was 59 active threads. The first case achieves a rather good balance given that it is hard to extract useful data parallelism for the search threads. The number of active threads for the search threads alone was 1.5 when using two search threads, and 7.1 when using eight search threads.

The main bottleneck for the performance is the increased workload to enforce consistency. The total number of times constraints are evaluated per explored search node is depicted in Figure 10 and Figure 11. Clearly, using parallel consistency increases the number of times we have to evaluate the constraints. This is because we cannot share data between constraints during their execution.

Table 2: Execution times in seconds when searching for 200 solutions to 100-Sudoku.

Consistency Threads per Search Thread	Search Threads			
	1	**2**	**4**	**8**
1	176	125	122	145
2	176	124	143	177
4	158	**110**	142	210
8	162	127	192	269

Table 3: Execution times in seconds when searching for 5 000 solutions to 100-Sudoku.

Consistency Threads per Search Thread	Search Threads			
	1	**2**	**4**	**8**
1	3 663	1 882	1 720	**1 649**
2	3 931	2 293	2 565	2 782
4	3 995	2 161	3 224	2 735
8	4 254	2 556	3 997	3 192

Consistency Threads per Search Thread ■ 1 ☐ 2 ▨ 4 ◹ 8
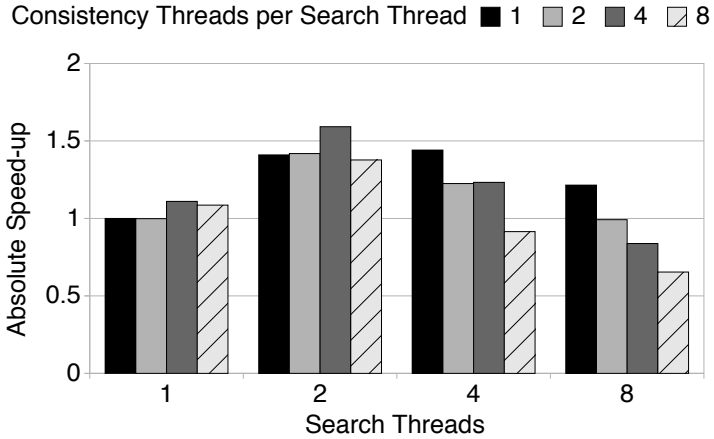
Figure 8: Speed-up when searching for 200 solutions to 100-Sudoku.

Consistency Threads per Search Thread ■ 1 ☐ 2 ▨ 4 ◹ 8
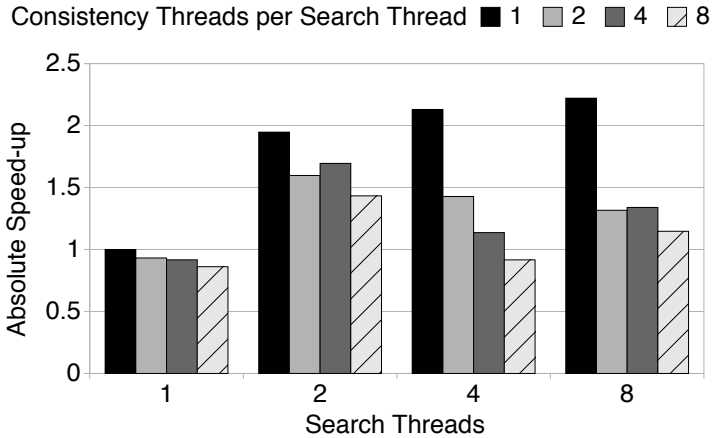
Figure 9: Speed-up when searching for 5 000 solutions to 100-Sudoku.

The second bottleneck for the performance of parallel consistency is synchronization. In our solution, we have several points of synchronization. The barrier before updates is particularly costly as the slowest consistency thread determines the speed.

The third bottleneck is the speed of the memory bus. Parallel search can quickly saturate the bus. Adding parallel consistency will worsen the performance. The performance clearly drops off towards the lower right hand corner of Table 2 and to the left of Table 3. This problem can to some extent be avoided by having a shared queue of tasks and a fix amount of threads in the program. These threads could then switch between performing consistency and search in order to adapt to the memory bus load.

The only way to fruitfully combine parallel search with parallel consistency is if we reduce the number of search nodes more than we increase their computational weight. The inherent problem in doing this is clear from the differences in results between Table 4 and Table 5. As shown by Figure 10, when the problem is small there is an almost linear increase in the number of consistency checks per search node as we add search threads. On the other hand, Figure 11 shows that the number of consistency checks varies a lot depending on the number of consistency threads. The reason is that when we have to explore a large search space we will run into more inconsistencies, which can be detected faster when using parallel consistency. However, inconsistent nodes have less computational weight. In conclusion, when parallel search starts to become useful, parallel consistency cannot pay off the computational overhead it causes.

Table 4: Number of times consistency was called for the constraints in 100-Sudoku when searching for 200 solutions.

Consistency Threads per Search Thread	Search Threads			
	1	2	4	8
1	23 475	47 487	122 587	217 339
2	36 585	73 017	171 613	243 754
4	36 585	72 833	169 849	231 745
8	36 585	73 369	160 696	242 317

Table 5: Number of times consistency was called for the constraints in 100-Sudoku when searching for 5 000 solutions.

Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	364 718	435 524	1 102 162	1 613 385
2	721 723	933 104	2 453 025	1 604 395
4	720 976	925 494	2 089 093	1 571 044
8	720 980	920 276	1 731 205	1 470 914

Table 6: Number of search nodes explored when searching for 200 solutions to 100-Sudoku.

Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	35 953	34 914	44 473	52 382
2	35 953	35 394	41 669	45 467
4	35 953	35 358	41 785	45 380
8	35 953	35 296	40 949	45 832

Table 7: Number of search nodes explored when searching for 5 000 solutions to 100-Sudoku.

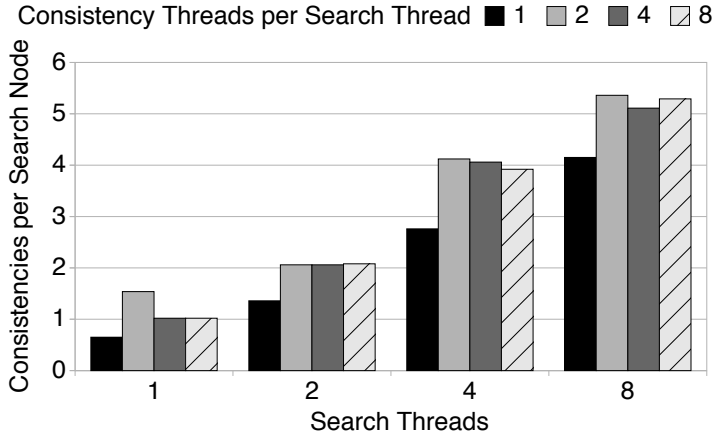| Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	763 827	784 204	958 969	1 002 032
2	763 827	784 980	920 223	881 475
4	763 827	785 547	915 305	894 489
8	763 827	784 443	923 470	886 828

Figure 10: Consistency enforcements per search node when searching for 200 solutions to Sudoku.
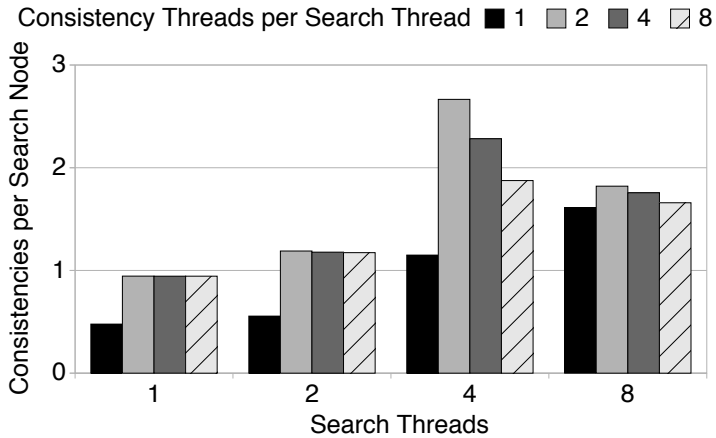
Figure 11: Consistency enforcements per search node when searching for 5 000 solutions to Sudoku.

5.3 Results for Queens

It is much harder to achieve an even load-balance for Queens than for Sudoku. The structure of Queens is quite different from Sudoku. In Sudoku we only have global constraints with a high time complexity. In Queens, there are lots of small constraints to calculate the diagonals. Hence, for most of the execution, we have a very low processor load if we only use parallel consistency [10].

We used Queens in order to illustrate how parallel consistency can be useful when parallel search is not. Problems with little need for parallel consistency have more room for the parallel search threads to execute. However, Queens is a highly constrained problem. Even with 550 queens, there are very few search nodes that need to be explored. Hence, parallel search will usually only add overhead. However, adding parallel consistency can compensate for the performance loss.

As shown in Table 8 and Figure 12, parallel search reduces performance. However, parallel consistency gives a speed-up even when we loose performance because of parallel search. We can also see that adding search threads can lead to sudden performance drops. This is largely because we end up overloading the memory bus and the processor cache. For eight search threads the performance increases compared to four threads. The reason is that we find a solution in a more easily explored part of the search tree.

Table 9, Table 10, and Figure 13 all support our earlier observation that the workload increases heavily if we use barrier synchronization. The results come from that we have to evaluate the simple constraints many more times if we do not share data between them and the alldifferent constraints. The reason why we still get a speed-up is that the alldifferent constraints totally dominate the execution time and do not have to be run that much more often in parallel consistency.

Table 8: Execution times in seconds when searching for one solution to 550-Queens.

Consistency Threads per Search Thread	Search Threads			
	1	**2**	**4**	**8**
1	107	109	464	325
2	95	101	454	191
4	**77**	82	405	213
8	**77**	82	426	215

Table 9: Number of times consistency was called for the constraints in 550-Queens when searching for one solution.

Consistency Threads per Search Thread	Search Threads			
	1	**2**	**4**	**8**
1	322 415	662 392	2 475 709	2 560 781
2	772 585	1 566 891	5 551 542	6 159 671
4	771 537	1 554 595	5 182 159	6 153 881
8	769 972	1 543 778	5 014 605	6 152 789

Table 10: Number of search nodes explored when searching for one solution to 550-Queens.

Consistency Threads per Search Thread	Search Threads			
	1	**2**	**4**	**8**
1	1 246	2 624	20 866	11 200
2	1 246	2 787	23 114	10 193
4	1 246	2 735	22 072	10 292
8	1 246	2 834	23 025	10 591

Consistency Threads per Search Thread ■ 1 ☐ 2 ■ 4 ▨ 8
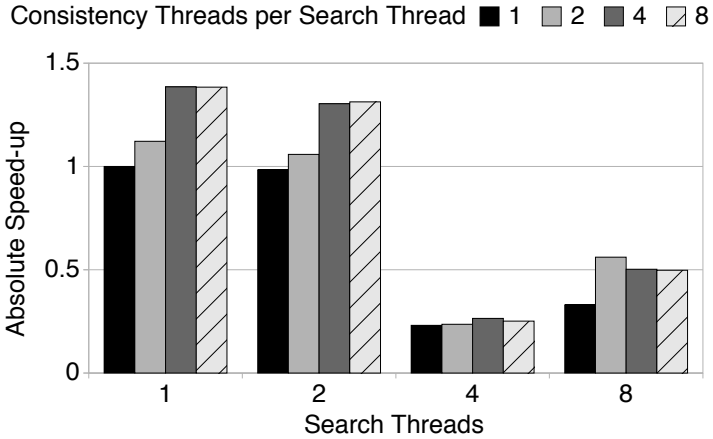
Figure 12: Speed-up when searching for one solution to 550-Queens.

Consistency Threads per Search Thread ■ 1 ☐ 2 ■ 4 ▨ 8

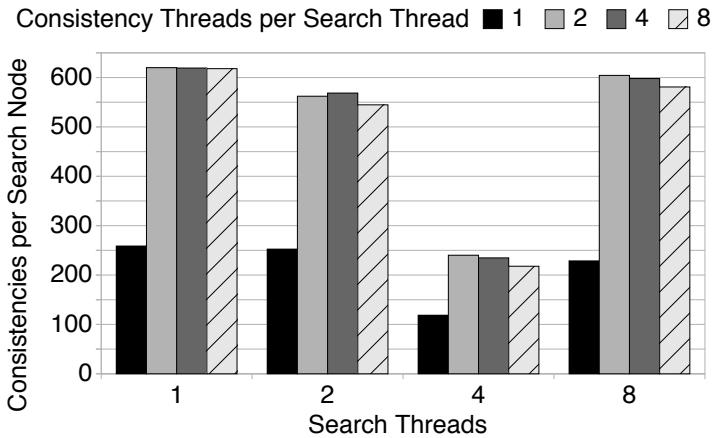Figure 13: Consistency enforcements per search node when searching for one solution to Queens.

6 Conclusions

The main conclusion is that it is possible to successfully combine parallel search and parallel consistency. However, it is very hard to do so. The properties of a problem, and size of the search space determines whether parallelism is useful or not. When trying to add two different types of parallelism, these factors become doubly important.

In general, if a problem is highly constrained, there is little room to add parallel search. If it is not constrained enough, there will be too many inconsistent branches for successfully adding parallel consistency. Finally, if a problem is reasonably constrained, the size of the search space, the uniformity of constraints, and the time complexity of the consistency algorithms determine whether fruitfully combining parallel search and parallel consistency is feasible.

In order to make sure that parallel consistency becomes less problem dependent, the need for synchronization must be reduced. This requires data to be shareable between global constraints during their execution. Since pruning is monotonic, this should be possible. However, it depends on the internal data structures used by the consistency algorithms. Hence, parallel consistency algorithms for each constraints may be a better direction of future research. Another interesting aspect is how much the order in which the constraints are evaluated matter to the performance. This is especially important for inconsistent states.

References

[1] G. Chu, C. Schulte, and P. Stuckey. Confidence-based work stealing in parallel constraint programming. In I. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241. Springer Berlin / Heidelberg, 2009.

[2] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[3] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, July 2003.

[4] K. Marriott and P. J. Stuckey. *Introduction to Constraint Logic Programming*. MIT Press, Cambridge, MA, USA, 1998.

[5] L. Michel, A. See, and P. Van Hentenryck. Parallelizing constraint programs transparently. In C. Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 514–528. Springer Berlin / Heidelberg, 2007.

[6] T. Nguyen and Y. Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1-2):227–250, 1998.

[7] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 359–366, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[8] V. Rao and V. Kumar. Superlinear speedup in parallel state-space search. In K. Nori and S. Kumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 338 of *Lecture Notes in Computer Science*, pages 161–174. Springer Berlin / Heidelberg, 1988.

[9] C. C. Rolf and K. Kuchcinski. Load-balancing methods for parallel and distributed constraint solving. In *IEEE International Conference on Cluster Computing*, pages 304–309, Sep/Oct 2008.

[10] C. C. Rolf and K. Kuchcinski. Parallel consistency in constraint programming. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 638–644. CSREA Press, July 2009.

[11] A. Ruiz-Andino, L. Araujo, F. Sáenz, and J. J. Ruz. Parallel arc-consistency for functional constraints. In *Implementation Technology for Programming Languages based on Logic*, pages 86–100, 1998.

[12] C. Schulte. Parallel search made simple. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, editors, *Proceedings of TRICS: Techniques for Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, Singapore, Sept 2000.

[13] C. Schulte and M. Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[14] X.-H. Sun and Y. Chen. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.

LOAD-BALANCING METHODS FOR PARALLEL AND DISTRIBUTED CONSTRAINT SOLVING

Load-Balancing Methods for Parallel and Distributed Constraint Solving

Carl Christian Rolf and Krzysztof Kuchcinski

Department of Computer Science, Lund University

Carl_Christian.Rolf@cs.lth.se, Krzysztof.Kuchcinski@cs.lth.se

Abstract

Program parallelization and distribution becomes increasingly important when new multi-core architectures and cheaper cluster technology provide ways to improve performance. Using declarative languages, such as constraint programming, can make the transition to parallelism easier for the programmer. In this paper, we address parallel and distributed search in constraint programming (CP) by proposing several load-balancing methods. We show how these methods improve the execution-time scalability of constraint programs. Scalability is the greatest challenge of parallelism and it is particularly an issue in constraint programming, where load-balancing is difficult. We address this problem by proposing CP-specific load-balancing methods and evaluating them on a cluster by using benchmark problems. Our experimental results show that the methods behave differently well depending on the type of problem and the type of search. This gives the programmer the opportunity to optimize the performance for a particular problem.

1 Introduction

In this paper, we discuss load-balancing methods for parallel and distributed solving of problems modeled in a constraint programming (CP) framework. CP has the advantage of being declarative [7]. Hence, the programmer does not have to make any significant changes to the problem declaration in order to solve it using parallelism. This means that the difficult aspects of parallel and distributed programming can be left entirely to the creator of the constraint solving framework.

Constraint programming has been used with great success to tackle different instances of NP-complete problems such as graph coloring, satisfiability (SAT), and scheduling [7]. A constraint satisfaction problem (CSP) can be defined as a 3-tuple $P = (X, D, C)$, where X is a set of variables, D is a set of finite domains where D_i is the domain of X_i, and C is a set of binary constraints where $C_{i,j}$ is a relation between X_i and X_j. Solving a CSP means finding assignments to X such that the value of X_i is in D_i, while all the constraints are satisfied. The tuple P is referred to as a constraint store.

Finding a valid assignment to a CSP is usually accomplished by combining backtracking search with consistency checking that prunes inconsistent values. To accomplish this, a variable is assigned one of the values from its domain in every node of the search tree. Due to time-complexity issues, the consistency methods are rarely complete [1]. Hence, the domains of the variables will contain values that are locally consistent, but cannot be part of a solution. This makes it difficult, if not impossible, to predict how much work remains in a search tree. In this paper, we discuss how CP-specific load-balancing methods can be used to address this.

The downside of parallelizing programs automatically is the difficulty of providing a good scalability. This problem is particularly complex in CP, since the pruning of values makes the work sizes hard to predict. For ordinary parallel search, it is often possible to estimate how much work the current process has to carry out. This makes it easier to efficiently split the workload between processes. In CP, however, the more inconsistent values are pruned, the harder it becomes to predict the size of the work. The reason is that the efficiency of the pruning changes as the domains are

split into smaller intervals. Hence, the change in work size depends on the depth in the search tree, the previous pruning, and the number of values and intervals in the domains.

One of the most efficient ways of improving the performance of the sequential solving process is to use constraints that maximize the pruning. The most powerful constraints, referred to as global constraints, include several or all of the variables in the problem. If a global constraint containing all variables is satisfied, finding a solution can often be quite simple. The downside for parallel search is that the pruning becomes even harder to predict.

In order to improve the scalability of parallel constraint programming, we present and evaluate several load-balancing methods designed specifically to suit CP. Most work on parallelism in constraint programming, e.g. [8], uses load-balancing methods designed for ordinary parallel search. While this works well for few processors, scalability problems become apparent when using many processors, especially in clusters where communication is more expensive. In our previous work [11], we tried to address this issue by achieving a better balance between communication and computation. However, the main issue is that in CP we cannot predict how much work is sent to another solver. Hence, work that is so small that it would be better to process locally will be sent unnecessarily. The more processors are used, the more of these instances of small work will be sent. By using load-balancing methods that utilize information available in the constraint solver, we are able to improve the scalability over general methods designed for ordinary parallel search.

Our research shows that the proposed CP-specific load-balancing methods improve the scalability of parallel constraint solving. In particular, our experiments show that methods taking the amount of labeled variables into account outperform naive methods such as random polling and fairness-based methods similar to round-robin.

The rest of this paper is organized as follows. In Section 2 the problem background is described, in Section 3 the related work is presented. Section 4 describes how the parallelism is achieved in our solver. Section 5 describes the different load-balancing techniques, Section 6 introduces the experiments and the results, and Section 7 gathers the conclusions.

2   Background

Most research on parallel search in CP has dealt with data-parallelism, since this is the most natural choice. It is easy to find split-points when variables can only assume a finite number of discrete values. The splitting of the work is independent of whether we use a cluster or a shared-memory machine.

As illustrated in Figure 1, splitting work in CP means that part of a domain will be sent to another processor. In the example, half the domain of X is sent from Processor 1 to Processor 2, together with the other variables and domains. After the data has been received, Processor 1 and 2 can explore their search trees independently of each other. The domains that have not been split will be sent intact to ensure that all possible assignments will be explored.
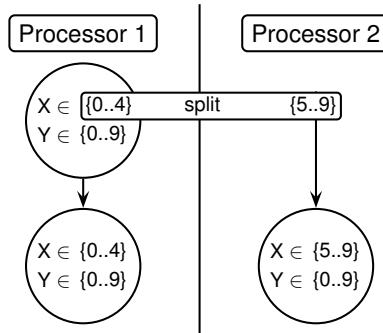
Figure 1: Example of a split, where half the domain of X is sent from Processor 1 to Processor 2.

There are two main principles of domain-splitting for parallel search: either split off a certain percentage of a domain at pre-defined points, or split off some part of the search tree whenever possible. Both [2] and [10] show that the latter approach is more effective. These methods can be quite efficient for depth-first search (DFS), but the unpredictability of the work size in CP makes them harder to use efficiently.

Unlike data-parallelism for DFS, the splitting of data between processes poses a problem in constraint programming. The reason is that the split will affect the domains of potentially all the variables that have not yet been assigned a value. In the example in Figure 2, with the constraint $Y > X$, the consistency methods can remove the value 0 from the domain of Y in Processor 1. In Processor 2, the consistency can remove values 0 to 5 from Y and the value 9 from the domain of X. This means that after splitting the work in CP, the search tree is likely to change shape. Since consistency methods are usually not complete, there is no way to efficiently calculate the sizes and the shapes of the search trees after a split.
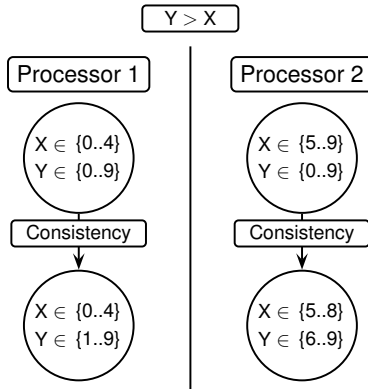
Figure 2: Example of consistency being performed with the constraint $Y > X$.

In conclusion, splitting work to achieve a good scalability is not trivial in constraint programming. While it is easy to find out where to split the work, it is hard to determine which computer should be allowed to do so. A lot of performance can be lost if, for example, computers with small work sizes get to send work more often than those with large work sizes. This is especially an issue for automatic parallelism where the programmer has not specified when and how the work should be split.

3 Related Work

Load-balancing is one of the most important aspects of parallelism. Several methods have been developed for parallel search [2]. Two of the most well-known methods are random polling and round-robin. Random polling is often more efficient than round-robin for parallel DFS [4, 5], but neither method was designed with CP problems in mind.

Previous work on parallelism and distribution in CP, e.g. [8, 11, 13], has relied on non-CP-specific methods such as random polling to decide which processor or computer is allowed to split its workload. These methods can be efficient for ordinary parallel search, or when only using few processors. However, the information available in a constraint solver can be used to create more advanced load-balancing methods. While the size of the work cannot be efficiently calculated, we can, for instance, let the computer that has labeled the fewest variables send work first. Such a method would not introduce a large overhead. No advanced computations are needed since the information is already available in the solver.

Automatic parallelism in constraint programming has been receiving increasing attention [8]. The reason is that the hardware development is moving quickly towards multi-core architectures. Lower cost technologies, such as clusters, are also likely to become interesting to CP. In this paper we try to improve the scalability of parallelism in CP by introducing several load-balancing methods for parallel and distributed constraint solving. In order to evaluate the performance we test these methods on a cluster.

4 Automatic Parallelism and Distribution

In our work we use the constraint solver *JaCoP* [3]. This solver is written entirely in Java, which makes it easy to add multithreading and distribution over network sockets. Figure 3 describes in detail how the parallelism works in JaCoP. The execution is split into three phases: initialization, search, and termination. The initialization prepares the solvers so that they can receive work. The search is a data-parallel depth-first search, and
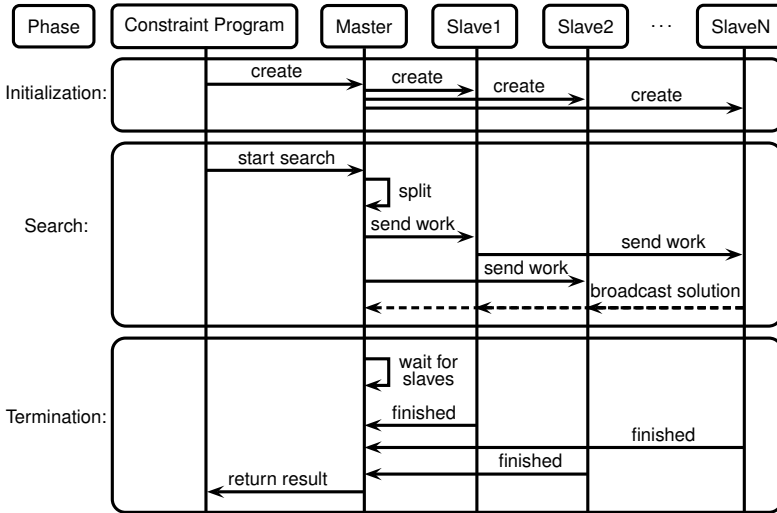
Figure 3: Model of the parallel solving in JaCoP.

the termination phase detects whether the search has finished. The only major simplification in Figure 3 is that the data is also split in the slaves when they send work. The user of the framework does not have to care about what goes on outside of the constraint program.

It is important to note that the model depicted in Figure 3 also applies to distributed search on a cluster. The main difference is the termination detection, which is more difficult on a non-shared memory architecture. When using distribution, the slaves report that they have finished to all other slaves as well as to the master. Furthermore, the creation of the slaves is not performed by the master.

In effect, the automatic parallelism and distribution acts as a middleware for transparently parallelizing constraint programs. The only change required to the original constraint code is replacing a single reference to the search object type. All other aspects of the parallel solving is left to the CP-framework. In the case of distributed search, like on a cluster, chang-

ing the search object type is enough. However, the programmer can select which load-balancing methods and communication models to use in order to tweak the performance for a particular problem.

We define the search phase to begin when all slaves are ready to receive work. The termination phase is defined to begin when the master is out of work. This is the most intuitive definition, since that process cannot receive work from the slaves in the distributed case.

The length of each phase depends on the problem. For small problems, the initialization can take longer time than the other phases combined. Problems with uneven search trees can spend most of their time in the termination phase. However, large problems spend most of their time in the search phase.

In both the parallel and the distributed case we use work sharing without a central controller. All the slaves can send work to each other and the master can send work to all slaves. The fact that the master cannot receive more work than it had from the start is not a significant limitation. In the parallel case, the master thread will be replaced by another slave when it runs out of work.

The solver is written in Java, hence, the parallelism was achieved using the built-in thread support. All processes hold a separate copy of the constraint store, this minimizes synchronization between the threads. In the distributed case, all computers have a copy of the store and communicate using TCP/IP sockets.

Communication between solvers can be performed by sending an entire copy of the store every time. However, sometimes it is preferable to send the store before the search begins and then communicate only a list of assignments. The downside of sending only the assignments is that the receiving computer needs to recompute consistency. As shown in [11] the scalability can be improved when choosing the model of communication that is preferable given the network load and expected time for recomputation.

5 Load-Balancing Methods

In order to use different load-balancing methods, each slave collects work-sending requests from other processors and then decides which one should be granted. If only one request is made within the specified timeout, that request will automatically be granted. Our heuristic load-balancing methods are presented below, and more formally in Table 1.

Table 1: Formal definitions of the load-balancing methods. Computer i is allowed to send work.

Method	Definition
Least Labeled First	$\exists_i \forall_j |L_i| \leq |L_j|$, where L_x is the list of labeled variables on computer x
Most Labeled First	$\exists_i \forall_j |L_i| \geq |L_j|$, where L_x is the list of labeled variables on computer x
Largest Domain First	$\exists_i \forall_j |D_i| \geq |D_j|$, where D_x is the domain to be split by computer x
Smallest Domain First	$\exists_i \forall_j |D_i| \leq |D_j|$, where D_x is the domain to be split by computer x
Least Pruning First	$\exists_i \forall_j P_i \leq P_j$, where P_x is the sum of percentages of pruned values divided by $|L_x|$
Most Pruning First	$\exists_i \forall_j P_i \geq P_j$, where P_x is the sum of percentages of pruned values divided by $|L_x|$
Least Sent First	$\exists_i \forall_j S_i \leq S_j$, where S_x is the number of times computer x has sent work
Random Polling	$\exists_i \exists_j R_{ij}$, where R_{xy} is a granted request from computer x to randomly selected computer y, $x \neq y$

Least Labeled First: This method allows the computer that has labeled the fewest variables to send work. The assumption is that the more variables are left to label, the more work there is left. This is not necessarily true, but it acts as a heuristic since the actual size of the work is hard to estimate.

Most Labeled First: This method is the opposite of Least Labeled First. While it may seem unintuitive, it could be better if reaching solutions quickly is important.

Largest Domain First: By allowing the computer with the largest current domain to send work we try to ensure that every computer has as little work as possible to process. This method will distribute the work more evenly across the solving processes.

Smallest Domain First: This is the opposite of Largest Domain First, here the computer with the smallest domain is selected. By prioritizing computers with narrow search trees, we may widen the search frontier. Thus reducing the probability of getting stuck in bad branches of the search tree.

Least Pruning First: This method prioritizes the computer that has achieved the least pruning during its labeling. The lower percentage of values that a computer can prune, the larger its search tree is likely to be. The pruning-measurement is the average percentage of values removed in the previous labeling.

Most Pruning First: This method prioritizes the computer with the highest amount of pruning. The argument for this method is similar to that of Smallest Domain First, quickly eliminating small branches can increase performance. Furthermore, the work that can be sent, despite good pruning, is more likely to need search.

Least Sent First: This is an approximation of round-robin. We did not implement the full round-robin algorithm, since it is usually slower than random polling [4, 5]. The measurement we use is the amount of times a computer has sent work in the past. This ensures that all computers that can send work will eventually get to do so.

Random Polling: This method is the standard random polling method, it does not use a timeout for the request. The first request to an idle machine will always be granted. Comparing our methods to Random Polling and Least Sent First first allows us to determine the benefit of CP-specific load-balancing methods.

5.1 Discussion of the Methods

Some of the load-balancing methods we introduce in this paper may seem unintuitive or unsuitable for depth-first search. For instance, Most Labeled First lets the computer that has the highest search depth to send work. Clearly this is unsuitable in regular DFS when exploring the entire search tree. However, if we have an optimization problem, we can prune many branches of the search tree for every solution that we find. Hence, finding good solutions fast can lead to a much better performance than trying to explore the entire search tree as fast as possible.

The load-balancing methods we present in this paper can serve to increase the performance in more ways than just sharing the work more efficiently. In constraint programming, heuristics are often used to optimize the performance. One common heuristic is first-fail, which picks the variable with the smallest domain to be the next node in the search tree. This may serve to reduce the overall size of the search tree. A CP-specific load-balancing method can operate in a similar way, especially when searching for one, or the optimal, solution. The programmer can pick a load-balancing method that finds solutions fast, or a method that shares the work more evenly. This affect the shape and size of the total search tree that is explored during the parallel search.

6 Experimental Results

6.1 Problem Set

In order to evaluate the performance we used two standard benchmark problems in constraint programming. The first problem is the n-Queens problem, which is the task of finding **all** possible ways to place n number of queens on an $n \times n$ chessboard. This task is interesting because the number of solutions is very large. Since, it is easy to find a solution, the difficulty is instead to minimize the backtracking by maximizing the pruning. The pruning is most easily improved by using the global *alldiff* constraint. The alldiff constraint takes an array of variables and applies a bounds consistency algorithm [6], which removes all inconsistent values in

the beginning and end of the domains of the variables in the array. Our implementation of alldiff uses the $O(n^2)$ algorithm, which is faster for small instances than the $O(nlog(n))$ algorithm [9]. In total we use three alldiff constraints, one for the columns and rows, and one for each diagonal. Thanks to the alldiff constraints for the diagonals, some of the inconsistent values inside the domains will also be pruned. Since the search is for **all** solutions, we will explore every consistent branch of the search tree. In our experiments we used $n = 15$.

The second problem is to find and prove the optimality of an Optimal Golomb Ruler. Golomb rulers are defined as a set of n positive integer values where the differences between every pair of variables are unique. The optimality is defined by the total size of the ruler, i.e., the largest number in the set. This optimization problem is interesting because it can be formulated using an alldiff constraint, ensuring the uniqueness of the differences. This constraint will then achieve the same level of pruning as an alldistinct constraint [12]. For this problem we used $n = 12$ in our experiments.

6.2 Experiment Setup

In our experiments we used the *JaCoP* solver [3], described in Section 4. The solver is written entirely in Java. The distribution was performed using TCP/IP sockets for the communication. We used a sender-initiated work-sharing without a central controller.

The experiments were performed on a cluster of AMD Opteron 148 processors with a clock frequency of 2.2 GHz and 1 MB of second level cache, each computer had a main memory of 1 GB. The operating system was CentOS Linux 4.4, and the machines were connected via a gigabit ethernet network. All tests were run 20 times, the presented results are the absolute speed-ups. The slaves use a timeout of 50 ms for the requests.

6.3 Results

The results are presented in Tables 2–5. The speed-up is sub-linear when using more than eight processors. Since this is also true for random polling, it is not caused by the request timeout but rather by the communication costs and unpredictable work sizes. The fastest method for 32 processors is marked in bold text and the slowest in italic.

Table 2: Execution times in seconds for n-Queens.

Load-Balancing Method	Processors				
	1	4	8	16	32
Least Labeled First	4125	1433	644	319	177
Most Labeled First	4125	1474	670	341	186
Largest Domain First	4125	1473	662	339	188
Smallest Domain First	4125	1418	670	322	189
Least Pruning First	4125	1435	740	367	219
Most Pruning First	4125	1486	648	318	180
Least Sent First	4125	1378	662	322	183
Random Polling	4125	1458	668	326	188

Table 3: Absolute speed-ups for n-Queens.

Load-Balancing Method	Processors			
	4	8	16	32
Least Labeled First	2.9	6.4	12.9	**23.3**
Most Labeled First	2.8	6.2	12.1	22.2
Largest Domain First	2.8	6.2	12.2	21.9
Smallest Domain First	2.9	6.2	12.8	21.8
Least Pruning First	2.9	5.6	11.2	*18.8*
Most Pruning First	2.8	6.4	13.0	22.9
Least Sent First	3.0	6.2	12.8	22.5
Random Polling	2.8	6.2	12.7	21.9

Table 4: Execution times in seconds for Golomb.

Load-Balancing Method	Processors				
	1	**4**	**8**	**16**	**32**
Least Labeled First	2144	623	448	291	190
Most Labeled First	2144	610	428	229	135
Largest Domain First	2144	620	461	274	161
Smallest Domain First	2144	657	399	261	175
Least Pruning First	2144	684	466	277	149
Most Pruning First	2144	618	409	267	173
Least Sent First	2144	685	494	322	177
Random Polling	2144	594	440	255	165

Least Labeled First is the fastest method for Queens, i.e., finding all solutions and exploring the entire search tree. This supports our assumption that computers with more variables left to label has more remaining work. When comparing the other methods, those that aim to eliminate small branches of the search tree have a slightly better performance than those trying to share the work more evenly.

As illustrated in Tables 2–5, there are some noteworthy differences between Queens and Golomb. In optimization problems, such as Golomb, we can bound the search tree by the best solution we have found. Hence, it may be more important to find solutions quickly than to explore the entire search tree as fast as possible. Since the pruning in Golomb is particularly good [12], it is better to measure the remaining work by the level of pruning rather than the size of the current domain.

Depending on the type of search we are performing, the search tree will have a different shape. The parallelization of the search will affect the shape further. Hence, we can use load-balancing techniques that fit the shape of the search tree better. In constraint programming the programmer can pick a heuristic used to select which variable should be next in the search tree. The CP-specific load-balancing methods we present here can act to amplify the benefits of these heuristics. This is in part why there are such large performance differences depending on the problem and the load-balancing method.

Table 5: Absolute speed-ups for Golomb.

Load-Balancing Method	Processors			
	4	**8**	**16**	**32**
Least Labeled First	3.4	4.8	7.4	*11.3*
Most Labeled First	3.5	5.0	9.4	**15.9**
Largest Domain First	3.5	4.6	7.8	13.3
Smallest Domain First	3.3	5.4	8.2	12.2
Least Pruning First	3.1	4.6	7.7	14.4
Most Pruning First	3.5	5.2	8.0	12.4
Least Sent First	3.1	4.3	6.7	12.1
Random Polling	3.6	4.9	8.4	13.0

Which method is best at approximating the size of the remaining work depends on the problem. If the pruning depends strongly on the number of labeled variables, Least Labeled First and Most Labeled First are more likely to be preferable.

Table 6 and Table 7 depict the performance advantage, or disadvantage, of the models compared to Random Polling. The differences can be significant even when only using few processors. For example, Smallest Domain First is 10.4 % faster than Random Polling for Golomb when using eight processors.

The fastest and slowest method compared to Random Polling are depicted in Figure 4 and Figure 5. As the graphs show, the difference in speed-up between the slowest and the fastest method is quite large. Furthermore, the model that is fastest for 32 computers is faster than the slowest method, regardless of how many computers are used. Lastly, we can see that the performance difference between Random Polling and the fastest and slowest methods increases the more computers are used.

Table 6: Performance change in percent compared to random polling for n-Queens.

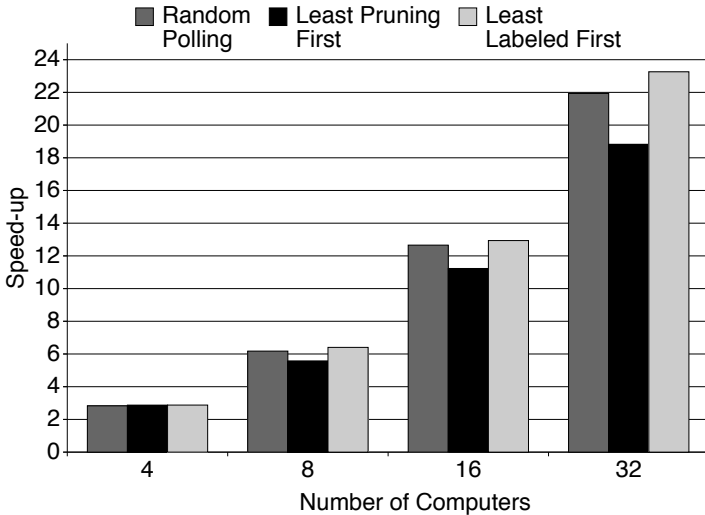| Load-Balancing Method	Processors			
	4	**8**	**16**	**32**
Least Labeled First	1.7%	3.7%	2.2%	**6.0%**
Most Labeled First	-1.1%	-0.4%	-4.6%	1.0%
Largest Domain First	-1.0%	1.0%	-3.8%	-0.1%
Smallest Domain First	2.8%	-0.3%	1.2%	-0.6%
Least Pruning First	1.6%	-9.7%	-11.3%	*-14.2%*
Most Pruning First	-1.9%	3.0%	2.3%	4.2%
Least Sent First	5.8%	0.9%	1.0%	2.8%

Figure 4: Speed-up of the fastest and slowest method for n-Queens compared to Random Polling.

Table 7: Performance change in percent compared to random polling for Golomb.

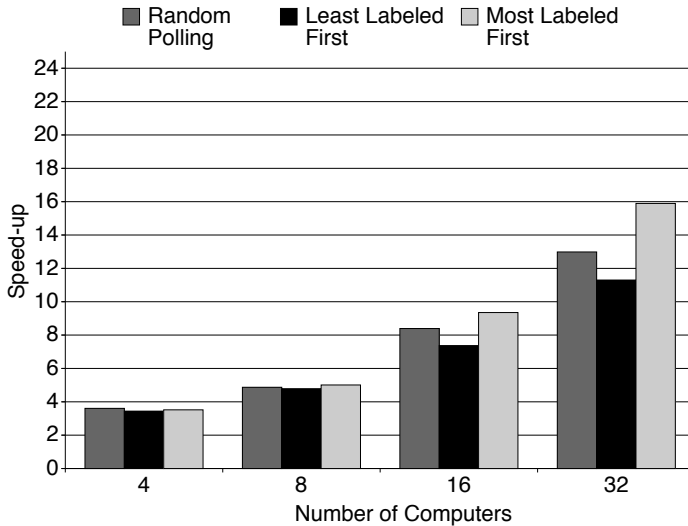| Load-Balancing Method	Processors			
	4	**8**	**16**	**32**
Least Labeled First	-4.6%	-1.7%	-12.2%	*-13.0%*
Most Labeled First	-2.5%	2.8%	11.4%	**22.4%**
Largest Domain First	-4.1%	-4.6%	-6.9%	2.7%
Smallest Domain First	-9.5%	10.4%	-2.1%	-5.8%
Least Pruning First	-13.1%	-5.5%	-7.8%	10.9%
Most Pruning First	-3.8%	7.7%	-4.5%	-4.8%
Least Sent First	-13.2%	-10.9%	-20.7%	-6.8%

Figure 5: Speed-up of the fastest and slowest method for Golomb compared to Random Polling.

7   Conclusions

The main conclusion of this paper is that load-balancing methods specific to constraint programming improve scalability. Different methods are good for optimization problems and search for all solutions. Hence, the programmer can select a method that is likely to suit the problem, thus increasing the performance by up to 20 %.

Which load-balancing method is best depends on both the problem and the type of search that is performed. These factors will affect the shape of the search tree, which in turn determines how efficient a particular load-balancing method will be.

Selecting a load-balancing method only requires the change of a single object reference in the problem declaration. However, the best approach for automatic parallelism would be to let the solver determine which model is best during the search. In future work, an adaptive load-balancing method could pick which method to apply depending on the current shape of the search tree and the network load.

References

[1] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[2] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):28–35, Jan/Feb 1999.

[3] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, July 2003.

[4] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: Design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.
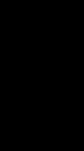
[5] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22:60–79, July 1994.

[6] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In G. Gottlob and T. Walsh, editors, *IJCAI*, pages 245–250. Morgan Kaufmann, 2003.

[7] K. Marriott and P. J. Stuckey. *Introduction to Constraint Logic Programming*. MIT Press, Cambridge, MA, USA, 1998.

[8] L. Michel, A. See, and P. Van Hentenryck. Parallelizing constraint programs transparently. In C. Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 514–528. Springer Berlin / Heidelberg, 2007.

[9] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 359–366, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[10] A. Reinefeld. Parallel search in discrete optimization problems. *Simulation Practice and Theory*, 4(2-3):169–188, 1996.

[11] C. C. Rolf and K. Kuchcinski. State-copying and recomputation in parallel constraint programming with global constraints. In *Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 311–317, Feb 2008.

[12] C. Schulte and P. J. Stuckey. When do bounds and domain propagation lead to the same search space? *ACM Transactions on Programming Languages and Systems*, 27:388–425, May 2005.

[13] J. Yang and S. D. Goodwin. High performance constraint satisfaction problem solving: State-recomputation versus state-copying. In *Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 117–123, Washington, DC, USA, 2005. IEEE Computer Society.

STATE-COPYING AND RECOMPUTATION IN PARALLEL CONSTRAINT PROGRAMMING WITH GLOBAL CONSTRAINTS

This paper is a reformatted version of *State-Copying and Recomputation in Parallel Constraint Programming with Global Constraints*, Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2008.

State-Copying and Recomputation in Parallel Constraint Programming with Global Constraints

Carl Christian Rolf and Krzysztof Kuchcinski

Department of Computer Science, Lund University

Carl_Christian.Rolf@cs.lth.se, Krzysztof.Kuchcinski@cs.lth.se

Abstract

In this paper we discuss parallelization and distribution of problems modeled in a constraint programming (CP) framework. We focus on parallelization of depth-first search methods, since search is the most time-consuming task in CP. The current hardware development is moving towards multi-core processors and the cost of building distributed systems is shrinking. Hence, parallelization and distribution of constraint solvers is of increasing interest when trying to improve performance.

One of the most important issues that arises in parallel computing is load-balancing, which requires a trade-off between processor load and communication. In this paper we present how reduced communication, at the cost of increased computation, can improve performance. Our experiments include global constraints, which are more powerful than binary constraints, but significantly more expensive to recompute in the average case. Our results show that recomputing data, rather than copying it, is sometimes faster even for problems that use global constraints. Given that copying is sometimes the better choice, we also present a method for combining copying and recomputation to create an even more powerful model of communication.

1　Introduction

In this paper we discuss parallelization and distribution of problems modeled in a constraint programming (CP) framework. We focus on parallelization of depth-first search (DFS) methods, since search is the most time-consuming task in CP. By parallelization we mean that the exploration of the search space is performed in parallel by several communicating constraint solvers. In our experiments we have achieved this by using distribution on a non-shared memory architecture, i.e., by running one solver per computer where inter-solver communication is performed via a network.

Constraint programming has been used with great success to solve many NP-complete problems such as graph coloring, satisfiability (SAT), and scheduling [9]. A constraint satisfaction problem (CSP) can be defined as a 3-tuple $P = (X, D, C)$, where X is a set of variables, D is a set of finite domains where D_i is the domain of X_i, and C is a set of binary constraints where $C_{i,j}$ is a relation between X_i and X_j. Solving a CSP means finding assignments to X such that the value of X_i is in D_i, while all the constraints are satisfied.

The most common method for finding a solution to a constraint problem is through a simple depth-first search that backtracks whenever an *inconsistency* has been found, i.e., when some constraint is violated. The search algorithm looks for a valid solution by organizing the search space as a search tree. In every node of the tree, a variable X_i is selected from X and assigned one of the values from D_i. The constraints are then recomputed, and any possible value in D_j that violates a constraint between X_i and X_j is removed. The process of removing inconsistent values is called *pruning*.

Due to time complexity issues, consistency in binary constraint networks is usually only performed between pairs of variables [2]. Hence, the domains may contain values that are *locally consistent*, but cannot be part of a solution.

One of the most efficient ways of improving the performance of the solving process is to use constraints that maximize the pruning. The most powerful constraints, referred to as *global constraints*, include several or

all of the variables in the problem. If a global constraint containing all variables is satisfied, finding a solution is often quite simple. The downside of global constraints is that they are computationally more expensive in the average case, since they often implement algorithms of rather high computational complexity [2]. Representing a global constraint with a set of binary constraints reduces the pruning power, but only the constraints whose variables change need to be recomputed, reducing the average time-complexity.

In this paper we use both the purely binary representation of problems and their description using global constraints. This provides us with an opportunity to relate our conclusions to previous studies, such as [18], that have focused on purely binary constraint problems.

Because of its declarative nature, constraint programming does not require the programmer to deal with how the actual solving takes place. Furthermore, given a programmer-friendly constraint framework, no significant changes need to be made to the problem declaration in order to solve the problem using parallelism. This means that synchronization and other difficult aspects of parallel programming can be left entirely to the creator of the constraint solving framework.

Parallelizing the solver, without changes to the problem specification, is not the only approach to parallel constraint programming. For instance, Mozart-Oz [16] or Parallel Prolog [1] lets the programmer specify in more detail how the parallelism should occur. This is useful when tuning the performance for a specific problem, but it does not help the programmer avoid the difficult aspects of parallelism. We use a Java framework that makes it possible for the programmer to run a parallel or distributed solving process by simply changing the search object type. This approach is similar to the work in [10], and our previous work [14].

The most studied form of parallelism in constraint programming is *data-parallelism*. This means that in a search tree node a certain amount of values in the domain of a variable are sent together with this variable to another solver, along with the domains of the previous nodes in the tree. This can be done in different ways. The most effective one is to always send a part of the domain from the current node in the search tree [13]. Parallel DFS has been studied quite extensively, but the pruning of

domains in CSP solving means that there is no way to usefully estimate how big the subtrees will be after the next assignment. This makes it very difficult to design an efficient *load-balancing* for CSP solving that takes the domain size into account. Instead, one can try to look at the depth of the search tree or the pruning.

Sending information over a network is much slower than, e.g., copying information in memory. This puts a strong limit on how much *speed-up* can be achieved. Speed-up is defined as the time needed for serial execution divided by the time for parallel execution [11]. Asymptotically, the maximum speed-up that can be achieved is *sub-linear*, i.e., when using n processors the program runs less than n times faster. The reason is that communication between processes is inherently serial.

Most studies on the performance of constraint solvers focus on randomly generated binary CSPs. These are generated using four variables n, m, p_1, and p_2. Where n is the number of variables, m is the domain size, p_1 is the density of constraints, and p_2 is the tightness of the constraints [3]. The advantage of using random problems is that the solution cannot be tweaked to suit the problem. The downside is that one leaves out one of the most important performance enhancing features: global constraints. In contrast to previous studies, we look at problems that use global constraints, and try to relate them to the binary representations. Given the difficulty of generating random global CSPs we will look at some standard benchmark problems.

Our contribution to the research is threefold. We perform a study of whether problems using only binary constraints can be used to draw conclusions about parallel constraint solving in general. We also present and evaluate a combined communications model based on an algorithm for determining which model of communication should be used during solving, something that has been lacking in previous studies. Finally we determine which model of communication is faster for some common problems and if this holds for global constraints as well as for binary CSPs.

The rest of this paper is organized as follows. In Section 2 the related work is presented, in Section 3 state-copying and state-recomputation are described in detail. Section 4 introduces the experiments and the results, and Section 5 gathers the conclusions.

2   Related Work

Most of the research on parallel search in CSP solving has dealt with data-parallelism. Other methods of parallelism are certainly possible, but splitting the work based on data is the most natural choice when dealing with depth-first search, especially when the variables can only assume a finite number of discrete values.

As seen in Figure 1, splitting of domains means that a part of a domain will be sent to another processor. In the example, half the domain of X is sent from Processor 1 to Processor 2. After the data has been received, Processor 1 and Processor 2 can explore their subtrees independently of each other. The domains that have not been split will be sent intact to ensure that all possible assignments will be explored. There are two main principles of domain-splitting: either split off a certain percentage of a domain at pre-defined points, or split off some part of the tree whenever possible. Both [5] and [13] show that the latter approach is the more effective. Using the more static method of splitting at predefined points of the tree can be quite efficient for DFS, but the pruning in constraint solving makes it difficult or even impossible to efficiently estimate the size of a particular subtree. Instead one has to use a dynamic way to split the work so that computers that become idle can find new work. The worst case situation is when one part of the tree dominates all other parts. As [6] shows, the simplest way to reduce the probability of this is to split off parts of the tree at the first levels of the search, this will be done automatically when splitting whenever possible. We use the method of splitting whenever possible, but only when there is a computer ready to handle the work. Our experiments indicate that except for the last nodes of the search tree, there is almost always a free computer to handle the work. The only case where no computer is free to receive work is when there is a full processor load, which is quite rare in our experiments.

The single most important part of parallelization is load-balancing, which is the method used to determine when to send the work, and to which processor. There are several different methods for performing load-balancing, divided into *receiver-initiated* and *sender-initiated* [5]. In the former case, a processor that becomes idle requests work from another
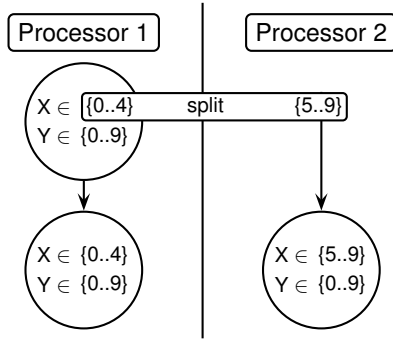
Figure 1: Example of domain splitting.

processor, while in the latter an overloaded processor tries to find an idle processor with which to share the work. In both cases, there are different methods for selecting a processor to communicate with. Two common methods are random polling, and round-robin [4]. As [13] shows, the best method depends on the structure of the computer system. In [18] a receiver-initiated method with a work-managing computer was used. In our experiments we used a *decentralized sender-initiated* approach, in order to reduce the overall network load. Hence, there was no master process that coordinated the sharing of work.

There are three main models of communication in parallel CSP solving: *state-copying*, *trailing*, and *state-recomputation*. Most work, including [6], [13], and [15], have used state-copying or trailing. State-copying means that a full copy of the state is sent to the new search process. State-recomputation, on the other hand, only sends a list of the decisions made by the original search process, guaranteeing that they are valid. If the state takes up a large amount of space, then the communication will take a lot longer than sending the list of assignments.

In trailing, only the changes between states are stored, providing a compromise between copying and recomputation. The problem is that identifying the changes often takes too long if there is a lot of pruning [17].

Previous studies, including [18], have shown a correlation between the variables used to generate a random CSP and the performance of the different models of communication. Especially the tightness of the constraints affected the speed-up of the models. However, [18] did not look at problems with global constraints, which are of particular concern for recomputation. Since we are interested in the likely worst case for recomputation we looked at global constraint problems.

3 Models of Communication

There are some questions that have not been fully explored in the field of parallel CSP solving. In this paper we have focused on the issue of determining which model of communication, copying or recomputation, is better for a certain problem, a question that has arisen quite recently.

The reason why state-copying dominates the research is that bandwidth has in the past been cheaper than processor power. As [18] shows, this is no longer the case. Today, one of the cheapest method for building a high-performance computer system is to buy regular PCs and connect them via an ethernet network. The performance of networking hardware may increase faster than that of processors, but the cost might not drop correspondingly. The reason is that only a few network architectures scale well with maintained bandwidth when increasing the number of nodes in the network. In the absence of an advanced architecture the performance advantage of recomputation is likely to increase.

In this paper we argue that state-recomputation is sometimes faster than state-copying, even for a computationally demanding task such as constraint solving with global constraints. This statement is backed-up by comparisons made in previous work for non-parallel, single processor search. Schulte [17], for instance, compared different solvers, showing that a solver using limited recomputation can outperform a solver that uses state-copying or trailing.

While increasing the speed-up of a parallel program is of significant interest, the different models of communication also have an impact on load-balancing. If there is less communication then there is more room for an advanced load-balancing scheme to further increase the performance.

It is important to realize that achieving a high processor load is not a goal in itself. Often the time needed for communication is longer than the time saved by letting an otherwise idle processor take over part of the work. Therefore, the efficiency of the load-balancing is best measured by the achieved speed-up, not by the average load. During our tests, we noticed that achieving a significantly higher load mostly led to a sharp decline in speed-up, which is why we decided to aim for the load that led to the highest speed-up for the problem.

3.1   Determining Which Model to Use

The ideal formula that determines whether recomputation or copying is better is as follows.

$$C = \frac{T_{RC} + T_R}{T_{CC} + T_L} \tag{1}$$

T_{RC} is the time to communicate the list of assignments and T_R is the time to recompute after receiving the list. T_{CC} is the time to communicate a copy of the state, and T_L is the time needed to load the state. If C is larger than one then copying is faster, and vice versa. The greater than one C is, the larger the gain of copying is, and the smaller than one it is, the greater the advantage of recomputation.

Given the difficulty of estimating the values in (1) at runtime, we need an approximation formula, preferably one that can be recomputed in real-time from the information collected from earlier parts of the search. There are several performance-variables that can be measured during the search, leading us to the following approximation.

$$C \approx \frac{T_{MR} \cdot L_A / L_{MR}}{T_{MC} \cdot C_C / C_{MC}} \tag{2}$$

T_{MR} is the average time needed for communication using recomputation. L_A is the length of the current assignment list, i.e., the number of assignments we have made to reach the current depth in the search tree. L_{MR} is the average length of the assignment lists that have been sent using

recomputation. T_{MC} is the average time for communication using copying. C_C is the current number of computers that are executing. C_{MC} is the average number of computers that were executing when state-copying have been used.

The reasoning behind (2) is that copying requires more communication than recomputation, while recomputation needs more computations than copying. Therefore, the performance bottleneck for copying should be the network. For recomputation, on the other hand, the bottleneck should be the time it took to prune the values at the receiving end of the split. Ideally the expected network load would be estimated using data from the network switch, but this data is not always possible to retrieve. Hence, we try to estimate the network load by looking at how many computers are likely to be sending work at the same time. The copy of the state is rarely big enough to significantly affect the sending time as much as the number of computers that are trying to send work. For recomputation, the time needed for pruning depends on the number of assignments rather than, e.g., the amount of values pruned at the machine that is sending the work. The reason is that the time for pruning on the sending machine is not necessarily related to the time needed for recomputation on the machine that is receiving work. This is because the pruning at the receiver can take place after all the assignments in the assignment list have been performed, reducing the need for several iterations of consistency checking.

In our experiments, we used (2) as a measure for whether to use state-copying or recomputation during the search. C was calculated every time a split of the work was possible. If C was below one, then we chose recomputation, and vice versa. We call this dual communication model *Dual Com*.

The algorithm we used for CSP solving, depicted in Figure 2, is a recursive backtracking depth-first search algorithm with forward checking. There are many ways to combine the pruning with the search, and several different ways to backtrack. However, as [2] shows, the combination of forward checking, i.e., pruning after assignments, combined with a simple backtracking is highly efficient for most problems. When using global constraints, the advantage of a simple backtracking scheme is likely to increase since there will be fewer inconsistent values after pruning.

LABEL($VarList$)
    if $ReceivedData \neq [\ ]$ **then**
        if $CommunicationMode = Recomputation$ **then**
            RECOMPUTE-STATE($ReceivedAssignmentList$)
        else
            LOAD-STATE($ReceivedState$)
    if $VarList = [\ ]$ **then**
        return $Solution$
    $Variable \leftarrow VarList_i$
    $VarList \leftarrow VarList - \{Variable\}$
    if $FreeComputers \neq [\ ]$ **then**
        $\{DomainLeft, DomainRight\} \leftarrow$
            SPLIT-DOMAIN(GET-DOMAIN($Variable$))
        if COMPUTE-DUAL-COM() < 1 **then**
            $CommunicationMode \leftarrow Recomputation$
            SEND-WORK($DomainRight, AssignmentList,$
               $CommunicationMode$)
        else
            $CommunicationMode \leftarrow Copying$
            SEND-WORK($DomainRight, State,$
               $CommunicationMode$)
    $Variable \leftarrow$ SELECT-VALUE($DomainLeft$)
    PRUNE-DOMAINS($State$)
    if ANY-EMPTY-DOMAIN($VarList$) = **true** **then**
        BACKTRACK($VarList$)
    else
        LABEL($VarList$)

Figure 2: The depth-first search algorithm with selection of model of communication.

As seen in Figure 2, if we have received data we will either recompute the state from an assignment list or load the received state from the sender. Then we proceed to select a variable. If there is a computer available, we split the domain of the variable into two parts, one to send and one that we keep processing locally. Then we decide whether to use recomputation or copying and send the work to the receiving computer. In the last part we select a value for the variable and check if this leads to an inconsistency, if this is not the case we proceed recursively. Otherwise we backtrack and select a different value for the variable. Finally, when the list of variables is empty, we have a consistent solution.

The most time-consuming tasks in the algorithm are the pruning and the recursive call on the last line. Since a lot of pruning will be performed when recomputing a state, this step takes about an order of a magnitude longer than loading a state. The time required for the recursion comes mostly from the need to allocate memory to store the information needed for the backtracking. Although we use a CSP solver written in Java, the average time needed for memory allocation may not be significantly faster with another language. The reason is that the allocation-size is different for each level of the search and therefore unpredictable. The time used by the send operations can be quite high for some network architectures. For example, if one uses a tree-like network topology, a message from one leaf to another, that passes all levels of the tree, might need to wait several times for the channel between levels to become free.

4 Experimental Results

Most of the previous studies on parallel and distributed constraint solving have relied on randomly generated binary problems to test performance. While it may be preferable to use random problems in some cases, we are interested in one of the worst cases for recomputation, namely when using constraints with a high time complexity.

4.1   Problem Set

The first problem is the n-Queens (nQ) problem, which is the task of finding all possible ways to place n number of queens on an $n \times n$ chessboard. This task is interesting because the number of solutions is very large. Since, it is easy to find a solution, the difficulty is instead to minimize the backtracking by maximizing the pruning. The pruning is most easily improved by using the global *alldiff* constraint instead of using only binary constraints. The alldiff constraint takes an array of variables and applies a *bounds consistency* algorithm [8], which removes all inconsistent values in the beginning and end of the domains of the variables in the array. Our implementation of alldiff uses the $O(n^2)$ algorithm, which is faster for small instances than the $O(nlog(n))$ algorithm [12]. The reason why we did not use the *alldistinct* constraint, which also prunes inconsistent values inside the domains rather than just the bounds, is that the alldistinct representation generally runs slower for this problem. In total we use three alldiff constraints, one for the columns, and one for each diagonal. Thanks to the alldiff constraints for the diagonals, some of the inconsistent values inside the domains will also be pruned. Since the search is for *all* solutions, we will explore every consistent branch of the search tree.

The second problem is to find and prove the optimality of an Optimal Golomb Ruler (OGR). Golomb rulers are defined as a set of n positive integer values where the differences between every pair of variables are unique. The optimality is defined by the total size of the ruler, i.e., the largest number in the set. This optimization problem is interesting because it can be formulated using an alldiff constraint, ensuring the uniqueness of the differences. This constraint will then have a direct impact on all the variables in the ruler, making it quite expensive to recompute. Since it is the differences that are in the alldiff constraint, the pruning will also be performed inside the domains, making an alldistinct constraint unnecessary.

In order to test the effect of the global constraints we also performed experiments on versions of nQ and OGR that only use binary constraints. The global constraints were then replaced by n^2 binary constraints, which are less expensive to recompute in the average case, since not all the constraints may need to be evaluated. The downside is that it is more expensive to communicate all the binary constraints than a global constraint.

4.2 Experiment Setup

We have used the *JaCoP* solver [7] for our experiments. This solver is written entirely in Java 5. The distribution was performed using TCP/IP sockets for the communication. We used a sender-initiated work-sharing without a central controller.

The experiments were performed on a cluster of AMD Opteron 148 processors, with a clock frequency of 2.2 GHz and 1 MB of second level cache. The main memory was 1 GB per machine and the operating system was CentOS Linux 4.4. The machines were connected via a gigabit network, with stacked switches communicating at 10 gigabit. All tests were run 10 times, and the presented results are the absolute speed-up based on the average execution time.

4.3 Results

Figure 3 depicts the absolute speed-up for the n-Queens problem while Figure 4 presents similar results for the Optimal Golomb Ruler. The results for finding all solutions to n-Queens are somewhat different from the results for Golomb. For n-Queens, the differences in speed-up are slightly bigger and recomputation is almost never better than copying. Dual Com is the fastest in both representations, except for the binary one with four computers. The reason why recomputation is less competitive for n-Queens than for Golomb is that there are three global constraints in n-Queens and only one in Golomb. This makes n-Queens a more difficult problem for recomputation. Dual Com is the best because we will use recomputation when the network load is very high. This reduces the cost of sending very small amounts of work, that might have been better to process locally.
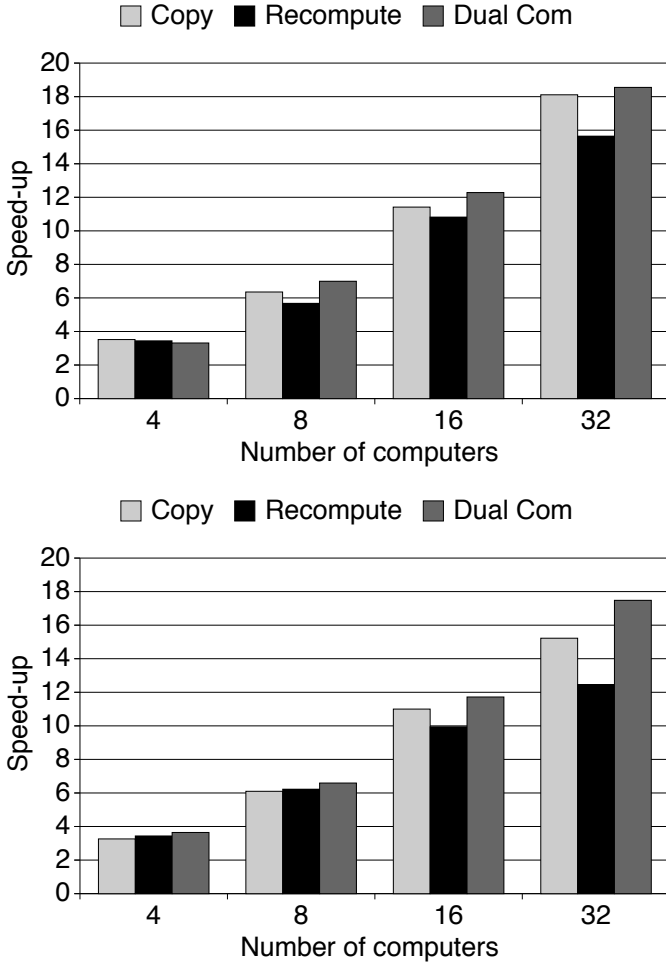
Figure 3: Absolute speed-up of finding all solutions to the binary representation (top) and the global representation (bottom) of n-Queens with $n = 15$.
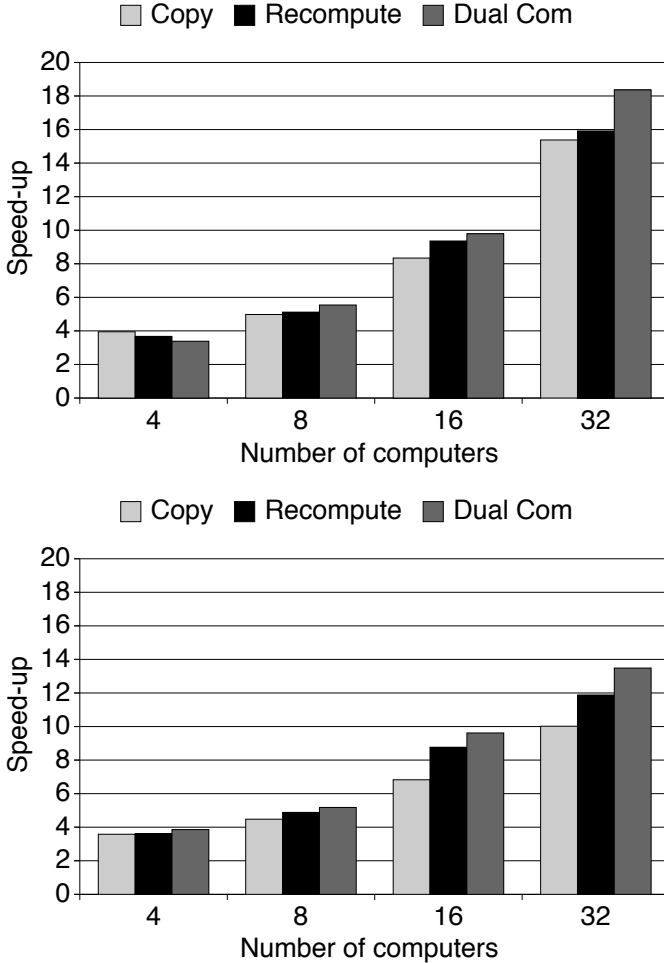
Figure 4: Absolute speed-up of the binary representation (top) and the global representation (bottom) of proving the optimal Golomb ruler with $n = 12$.

As Figure 4 shows, recomputation and copying are nearly equally fast when using eight computers or less. But in the more interesting case, when using many computers, recomputation is significantly faster, at least for the global representation. However, Dual Com is the fastest model for all instances of Golomb except the binary representation with four computers. In general the differences between the models are smaller for fewer computers.

When looking at smaller problems, such as Golomb with $n = 11$, the differences between the models of communication are smaller. The more communication is performed, i.e., the more splits are performed during the solving, the greater the differences between the models. Generally, the more values are left after the pruning, the more work can be shared with other computers, increasing the total amount of communication and the potential benefit of Dual Com.

The binary representation often has a higher speed-up than the global representation. The reason is that there is more need for processing power, since the pruning is a lot weaker. In actual time, the binary representations of n-Queens and Golomb take more than twice as long to finish as the global representation. The speed-up gained from using global constraints instead of using only binary constraints is presented in Figure 5.

The reason why the differences between the global and the binary representation are shrinking in Figure 5 is that there are more computers to compensate for the low pruning. With enough computers there would be no difference at all, since the global representation would not have as many values to send to other solvers as the binary representation.

As seen in Table 1, the average processor load is sometimes fairly low. The reason is that the communication in those cases is particularly expensive. Therefore the computers will spend a lot of time sending or loading the data rather than processing it. Also, the 32 computer case is the most difficult in our studies to achieve a high load for. The load is naturally higher when fewer computers are used.

The execution times, shown in Table 2, vary quite a lot. The reason we did not use larger problems is that the execution time grows significantly with the problem size. We also wanted our experiments to reflect the performance of real world usage of constraint programming outside of

a supercomputer environment. This performance may not be reflected by problems taking several hours to solve even when using many computers.

The reason why the execution time is the same for the different models when using only one processor is that we looked at the absolute speed-up. If the absolute speed-up is higher for one model than the others, that model is by necessity the fastest for that problem instance. The reason is that the speed-up is calculated by comparing the execution time of the parallel program to the fastest available serial program for the same problem.
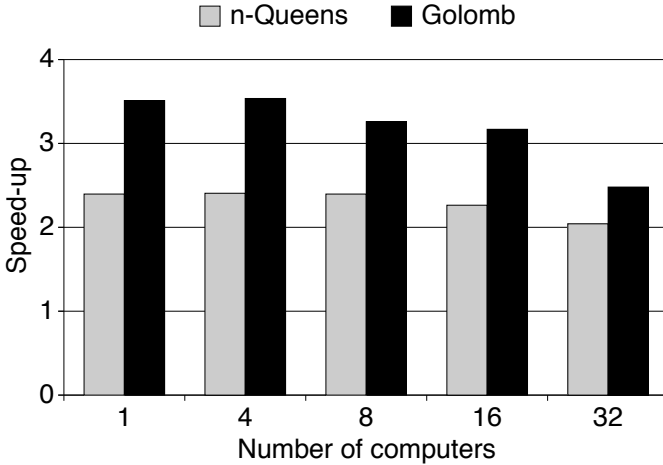

Figure 5: The speed-up of using the global representation for the problems showed in Figure 3 and Figure 4.

Table 1: Average processor load during search with 32 computers.

Model	Problem			
	nQ-Bin	nQ-Glob	OGR-Bin	OGR-Glob
Copy	45%	71%	48%	65%
Recompute	56%	70%	80%	83%
Dual Com	50%	72%	71%	74%

Table 2: Execution times in seconds for the problems and models.

	Processors				
Problem	**1**	**4**	**8**	**16**	**32**
nQ-Binary Copy	9887	2807	1556	866	546
nQ-Binary Recompute	9887	2870	1741	914	632
nQ-Binary Dual Com	9887	2980	1413	805	533
nQ-Global Copy	4125	1266	676	375	271
nQ-Global Recompute	4125	1201	663	415	331
nQ-Global Dual Com	4125	1131	626	352	236
OGR-Binary Copy	7528	1906	1513	902	490
OGR-Binary Recompute	7528	2051	1471	805	473
OGR-Binary Dual Com	7528	2225	1358	769	410
OGR-Global Copy	2144	599	479	314	214
OGR-Global Recompute	2144	593	439	245	181
OGR-Global Dual Com	2144	556	414	223	159

5   Conclusions

Given the experimental results, we can claim that our original thesis that recomputation can sometimes be faster than copying holds even for parallel CSP solving with global constraints. The claim may seem rather strong given that we are often quite far from the theoretical limit of speed-up. But one of the main reasons why recomputation can be faster is that it achieves a significantly higher average load than copying, as seen in Table 1. In other words, the bottleneck of our experimental setup is not the computing power, but the network latency, precisely the situation that can be observed in the design of many modern super-computer systems.

Using a different memory architecture, such as cache-coherent non-uniform memory access would not necessarily render a greater speed-up. The usage of global constraints, with the significant changes to the domains that usually follow, means that in a distributed shared memory architecture there would be a lot of extra communication compared to using

non-shared memory. In the case of shared memory, the best case would be if we hardly ever had to use the network to access the memory, which is essentially the same situation as using non-shared memory.

We can also claim that the models scale approximately the same, independently of whether we use the global or the binary representation. This is good news given that most of the research on solver performance has focused on binary constraint problems. It also means that our conclusions about the different models of communication are more likely to hold for randomly generated binary CSPs than if the differences had been larger.

From our experiments we can also draw the conclusion that our formula, (2), works quite well for determining which model of communication to use. On average about 70% of the work was sent using copying and the remaining part using recomputation. For almost all of the problems, combining copying and recomputation led to a higher speed-up. While the difference between Dual Com and the other models was sometimes small, Dual Com serves to reduce the network load compared to copying, and the processor load compared to recomputation. Hence, in a situation where several different processes are competing for resources, a lower load will help the other processes to achieve a higher performance.

References

[1] L. Araujo and J. Ruz. A parallel Prolog system for distributed memory. *Journal of Logic Programming*, 33(1):49–79, 1997.

[2] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[3] I. Gent, E. Macintyre, P. Prosser, B. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6:345–372, 2001.

[4] A. Grama and V. Kumar. A survey of parallel search algorithms for discrete optimization problems. *ORSA Journal of Computing*, 7(4):365–385, 1995.

[5] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):28–35, Jan/Feb 1999.

[6] Z. Habbas, F. Herrmann, P.-P. Merel, and D. Singer. Load balancing strategies for parallel forward search algorithm with conflict based backjumping. In *ICPADS '97: Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pages 376–381, Washington, DC, USA, 1997. IEEE Computer Society.

[7] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8:355–383, July 2003.

[8] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In G. Gottlob and T. Walsh, editors, *IJCAI*, pages 245–250. Morgan Kaufmann, 2003.

[9] K. Marriott and P. J. Stuckey. *Introduction to Constraint Logic Programming*. MIT Press, Cambridge, MA, USA, 1998.

[10] L. Michel, A. See, and P. Van Hentenryck. Parallelizing constraint programs transparently. In C. Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 514–528. Springer Berlin / Heidelberg, 2007.

[11] G. Mitra, I. Hai, and M. T. Hajian. A distributed processing algorithm for solving integer programs using a cluster of workstations. *Parallel Computing*, 23(6):733–753, 1997.

[12] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 359–366, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[13] A. Reinefeld. Parallel search in discrete optimization problems. *Simulation Practice and Theory*, 4(2-3):169–188, 1996.

[14] C. C. Rolf. Parallel and distributed search in constraint programming. Master Thesis, Department of Computer Science, Lund University, June 2006.

[15] C. Roucairol. Parallel processing for difficult combinatorial optimization problems. *European Journal of Operational Research*, 92(3):573–590, 1996.

[16] P. V. Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, and C. Schulte. Logic programming in the context of multiparadigm programming: The Oz experience. *Theory and practice of logic programming*, 3(06):715–763, 2003.

[17] C. Schulte. *Programming constraint services: High-level programming of standard and new constraint services*. Springer-Verlag, Berlin, Heidelberg, 2002.

[18] J. Yang and S. D. Goodwin. High performance constraint satisfaction problem solving: State-recomputation versus state-copying. In *Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 117–123, Washington, DC, USA, 2005. IEEE Computer Society.

DISTRIBUTED CONSTRAINT PROGRAMMING WITH AGENTS

This paper is a reformatted version of *Distributed Constraint Programming with Agents*, International Conference on Adaptive and Intelligent Systems, 2011.
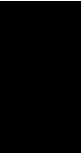
Distributed Constraint Programming with Agents

Carl Christian Rolf and Krzysztof Kuchcinski

Department of Computer Science, Lund University

Carl_Christian.Rolf@cs.lth.se, Krzysztof.Kuchcinski@cs.lth.se

Abstract

Many combinatorial optimization problems lend themselves to be modeled as distributed constraint optimization problems (DisCOP). Problems such as job shop scheduling have an intuitive matching between agents and machines. In distributed constraint problems, agents control variables and are connected via constraints. We have equipped these agents with a full constraint solver. This makes it possible to use global constraint and advanced search schemes.

By empowering the agents with their own solver, we overcome the low performance that often haunts distributed constraint satisfaction problems (DisCSP). By using global constraints, we achieve far greater pruning than traditional DisCSP models. Hence, we dramatically reduce communication between agents.

Our experiments show that both global constraints and advanced search schemes are necessary to optimize job shop schedules using DisCSP.

1   Introduction

In this paper, we discuss distributed constraint programming with agents (DCP). We introduce advanced agents with global constraints, and advanced search to solve distributed constraint satisfaction problems (DisCSP), in particular distributed constraint optimization problems (DisCOP). DCP is a special form of constraint programming (CP), where variables belong to agents and can only be modified by their respective agents.

We differentiate DCP from DisCSP since DisCSP has traditionally assumed one variable per agent [18]. In contrast, we study the case where agents can control several variables, making it possible to use global constraints. Such constraints are often needed when solving complex CP problems, such as job shop scheduling problems (JSSP).

Using global constraints in DCP requires that each agents has its own constraint solver. Having a full solver in each agent also makes modeling and communication more efficient. As far as we know, no published work on DisCSP has studied global constraints. In earlier work, these constraints were transformed into equivalent primitive or table constraints. This led to inefficient solving and model representation.

There are two main contributions in this paper:

- We empower the individual agents with a full constraint solver; and

- We introduce an advanced search scheme.

A major advantage of each agent having a full solver is that we can create advanced search methods by adding constraints during search. In this paper, we study JSSP and search that adds ordering-constraints before the actual assignments, significantly increasing the performance. We are not aware of any previous research on DisCSP that studies this type of search.

Formally, a constraint satisfaction problem (CSP) can be defined as a 3-tuple $P = (X, D, C)$, where X is a set of variables, D is a set of finite domains where D_i is the domain of x_i, and C is a set of primitive or global constraints containing several of the variables in X. Solving a CSP means finding assignments to X such that the value of $x_i \in D_i$, while all the constraints are satisfied. P is referred to as a constraint store.

Finding a valid assignment to a constraint satisfaction problem is usually accomplished by combining backtracking search with consistency checking that prunes inconsistent values. In every node of the search tree, a variable is assigned one of the values from its domain. Due to time-complexity issues, the consistency methods are rarely complete. Hence, the domains will contain values that are locally consistent, i.e., they will not be part of a solution, but we cannot prove this yet.

DisCSP, as used in [17], can be defined similarly to CSP with the 4-tuple $P = (A, X, D, C)$, where A is a set of agents and X is a set of variables so that $x_i \in a_i$. D is a set of finite domains, and C is a set of sets of *binary* constraints. Each variable x_i has a finite domain d_i, and each set of constraints c_{ij} connects two agents a_i and a_j, where $i \neq j$. Furthermore, each variable is controlled by exactly one agent. Lastly, the constraint network builds a connected graph. In other words, each agent is connected to another agent. Hence, there is at least one path from agent a_i to agent a_j.

Our model of DCP extends the DisCSP definition to a higher level. We retain the properties that a variable is controlled by exactly one agent, and that there is a path from any agent a_i to agent a_j. Now, however, X is a set of sets of variables, C is a set of sets of n-ary constraints, and D is a set of sets of finite domains. Every agent a_i has a set of variables x_i and a set of constraints c_i. In [13], a similar definition is introduced, but not expanded upon. In fact, we are not aware of anyone actually using the main advantage of having many variables per agent. The fact that our agents can have global constraints enables us to use the full power of modeling and pruning in CP.

In DCP, we can perform the consistency and search phase asynchronously [20]. First, we let each agent establish consistency internally, then send its prunings to the agents that are connected via constraints. Figure 1 depicts the structure of the constraint network for a small JSSP. Each agent holds two variables and ensures no overlap between tasks via a cumulative constraint [2]. The constraints between agents are the precedence constraints, stated at the edges. Whenever a variable that is part of a connected constraint changes, the prunings will be propagated to the connected agents. Using our formal model, we, e.g., have $A = \{X, Y, Z\}$,

$x_x = \{Xa, Xb\}, c_x = \{cumulative([Xa, Xb]), Xa > Ya, Xb < Yb\}$
and $c_{xy} = \{Xa > Ya, Xb < Yb\}$.

The rest of this paper is organized as follows. Section 2 introduces the background and the related work. In Section 3, our model of DCP with global constraints and advanced search is described. Section 4 describes our experiments and results. Finally, Section 5 gathers our conclusions.
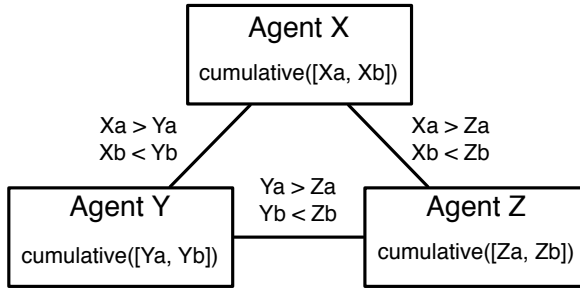

Figure 1: Model of a distributed JSSP, where each agent holds several variables.

2   Background and Related Work

Most work on DisCSP deals with the scenario where each agent holds a single variable and only binary constraints exist between the agents [18]. These problems are typically solved with an asynchronous search, where consistency is enforced between the connected agents [20]. One notable exception is [13]. However, that paper mentions neither global constraints nor advanced search methods.

The model of each agent only controlling one variable and only having binary constraints can technically be used to model any problem. However, even the latest search algorithms need to send a huge amount of messages to other agents [6] to solve such problems. This makes such a limited model less feasible when dealing with large or complex problems. This is especially problematic for optimization problems, since there is a greater need for search than for simple satisfiability problems.

One main difference between our model and previous work, such as [20, 6, 4, 17, 13], is that we can communicate entire domains. When a domain has been received, the prunings it carries are evaluated. This is much more efficient than sending one value from a domain at a time and getting a *Good* or *NoGood* message back.

Privacy is often used to motivate distribution of variables. Previous work, such as [21] shows that perfect privacy is possible for DisCSP. However, in the real world, complex encryption and minimal communication are impractical if they decrease performance too much. Our ultimate goal is to use our work for scheduling in autonomous unmanned aerial vehicles [9]. Hence, we focus more on performance than privacy.

A great limitation of previous work is that the problem model is usually translated into a table form [11]. These tables represent all possible assignments by the cartesian product of the domains in the constraint. For many problems, this representation is unfeasibly large [17]. In scheduling, a single cumulative constraint, ensuring no overlap of tasks [2], would have to be translated to binary constraints for every single time point. Even small scheduling problems would need thousands of constraints.

Many complex optimization problems need global constraints to solve in reasonable time. Some papers on DisCSP build advanced structures of agents. Others add a master-like agent that controls the global limits of the problem [12]. However, as far as we know, no one provides global constraints in each agent.

In order for DCP to solve large problems which are relevant to the real world, like JSSP, we need more advanced agents. Theoretically, one variable per agent is sufficient to model any DisCSP. However, just as global constraints can be reduced to binary constraints, the decreased pruning makes such an approach unrealistic for large optimization problems. This paper introduces agents with full constraint solvers, in order to make DCP feasible for industry use.
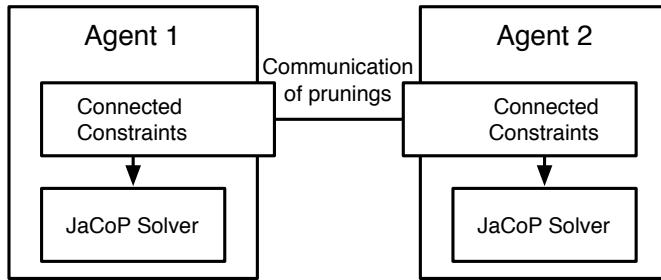
Figure 2: Our model of DCP, each agent holds a full constraint solver.

3   Distributed Solvers

Figure 2 depicts our model of DCP. Each agent holds a separate copy of the JaCoP solver [8], and only controls the variables that are needed for that part of the problem model. For instance, in JSSP, each agent holds the variables representing the tasks on the machine that the agent models. The precedence constraints between tasks assigned to different machines are stored in the connected constraints, since they constrain tasks controlled by different agents. This is how prunings are sent between agents.

Figure 3 depicts a simplified view of the distributed constraint evaluation process and the search. All time consuming steps in our solving are parallel. As depicted in Figure 3, the algorithm evaluates consistency and the agents vote on the next master in parallel. However, in order to guarantee synchronicity, the agents must wait for all prunings to be finished before they can move on to select the next master. Hence, the algorithm moves from synchronous to asynchronous execution of the agents, and back again, with every assignment.

When consistency is evaluated, all prunings are sent directly between the agents that are part of the connected constraint. Hence, the master agent is not controlling communication. It serves only to make an assignment decision and ensure that all agents are synchronized for the next step in the execution. The next step after an assignment may be to backtrack, or locate the next master, or to communicate a solution.
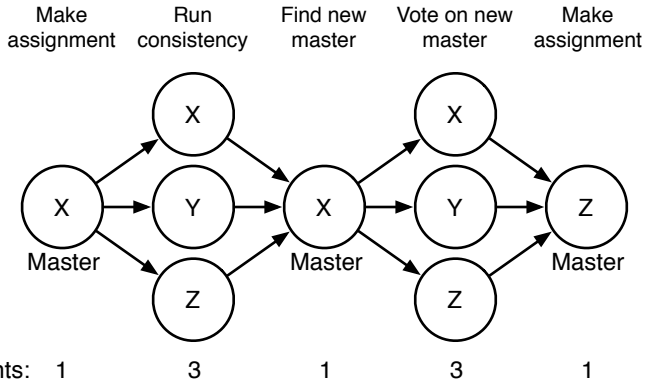
Figure 3: The progress from assignment to next assignment. X, Y, and Z are agents.

An example of the operations of our model is depicted in Figures 4–5, which show all execution steps. The execution progresses as follows.

1. When the solving is initialized, all agents start to run consistency of their constraints, see Figure 4(a).

2. If there are changes to a variable that is in a connected constraint, those prunings are sent to the agent holding the other variable of the connected constraint, see Figure 4(b). In this paper, we only study the case of binary constraints between agents. Each agent holds a queue of pruning messages, when changes have been received, consistency is again evaluated in the agent. This process continues iteratively until there are no more prunings sent between agents.

3. As soon as the consistency is finished, a negotiation determines which agent will start the search, see Figure 4(c). This follows the principles of distributed election [5].

4. The agent holding the variable with the highest priority, defined by a user configurable heuristic, gets the master token, see Figure 4(d) and Figure 5(a). In this paper, we look at synchronized search. This means that only one agent holds the master token and only this master gets to make the next assignment decision.

5. The master makes an assignment and enforces consistency, see Figure 5(b).

6. The master sends the prunings to the agents that have connected constraints containing changed variables, see Figure 5(c).

7. When the agents receive prunings, they automatically run consistency, see Figure 5(d).

8. When consistency has finished again, we are at the same position as in Figure 4(c). We renegotiate which agent is to be the new master.

The procedure above continues until all variables have been assigned a value. When a master finds a solution, the cost of the solution can be shared amongst all agents by propagating it to all agents connected to the master. These agents then propagate it further, and so on, until all agents are aware of the solution cost. This is similar to the communication in [3]. Sharing solution costs is necessary in order to use branch and bound search.

If backtracking is necessary, we will undo the assignment leading to the inconsistency. If the current master has run out of possible assignments, it will send a message to the previous master telling it to backtrack. Hence, all agents that have been masters keep track of which agent was master before itself. Furthermore, since agents have several variables, an agent can become master several times in the same search tree branch. Agents therefore also need to keep track of backtracking to themselves.

The pseudo-code for our model is shown in Figure 6 and Figure 7. The `receive` method will be called automatically by the agent whenever a message has been received. Communication between agents are performed by similar syntax to that of [7]. All communication of costs is handled by connected constraints and is therefore controlled by the problem model. This gives great versatility to our model.

The biggest challenge in our distributed model is to detect that all agents are synchronous. For instance, detecting that consistency has reached a fixpoint and it is time to make the next assignment. That detection takes place in the handling of the message `Wait_For_Consistency`. Verifying whether agents are running and consistent can be done as for DisCSP, by using the process of [3].

## 3.1 Advanced Search in Distributed Constraint Programming

In order to solve complex JSSP, we need the more advanced search that is made possible by our model. The algorithm presented in Figure 6 and Figure 7 is somewhat simplified. For JSSPs, we use a sequence of two search methods. The first orders the tasks on each machine by adding precedence constraints. The second assigns actual start times for each task. This is based on the principles described in [1]. While some problems may solve without the ordering, many require an ordering to solve in reasonable time.

Figure 8 depicts the algorithm for the ordering search. During the ordering, the machine with the least slack in the tasks scheduled on it is selected. Then we pick the task, running on that machine, with the smallest start time. Finally, we impose that the selected task has to execute before the other tasks on that machine, and we remove it from the list used to calculate slack. This procedure is repeated recursively.

This type of advanced search is not possible in all DisCSP solvers. Many DisCSP solvers cannot impose constraints during the search. Even solvers that can impose new constraints, are often limited by mostly supporting table constraints [11]. If only table constraints are supported, the memory use of the solver will increase greatly whenever new constraints are imposed for every time unit of the schedule.
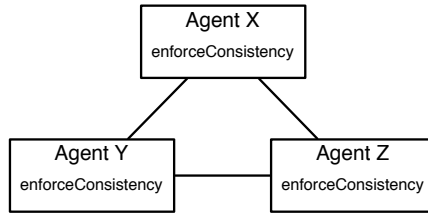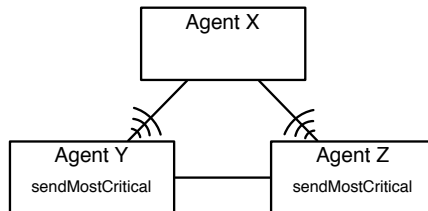
(a) Start: Enforce consistency
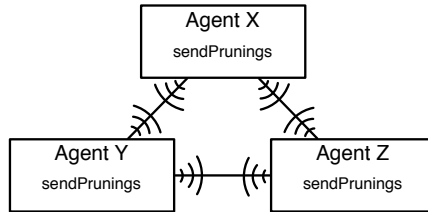
(b) Communicate prunings
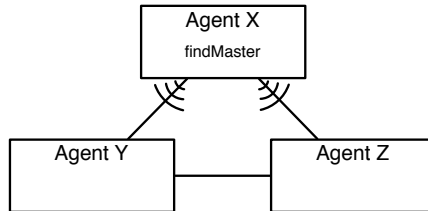
(c) Elect new master

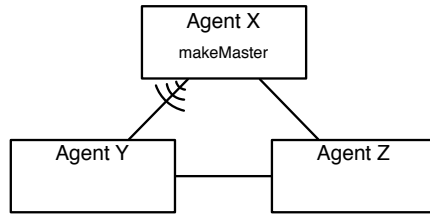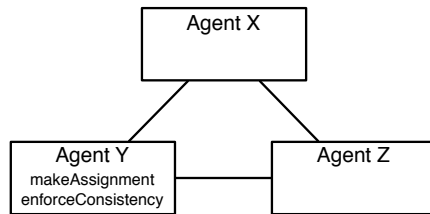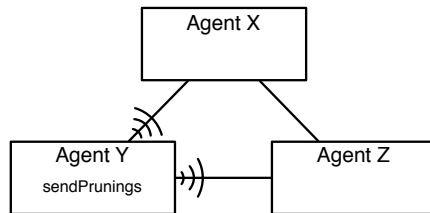(d) Reply with measurement

Figure 4: The first part of the operating sequence for consistency and search in our model. The waves along the edges indicate communication.
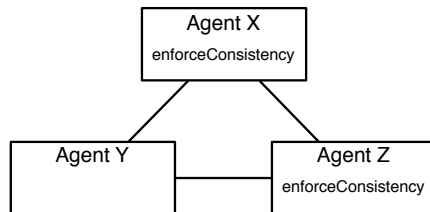
(a) Make Agent Y new master

(b) Master assigns and runs consistency

(c) Master communicates prunings

(d) Affected agents enforce consistency

Figure 5: The second part of the operating sequence for consistency and search in our model.

```
 1  // variables controlled by the agent V
 2  // actors that participate in the problem A
 3
 4  receive(message) {
 5    switch (message.type) {
 6
 7      case Make_Master(oldMaster):
 8        master = true
 9        previousMaster = oldMaster
10        this ! Start_Search
11
12      case Start_Search:
13        v = selectionHeuristic.selectVariable
14        if (v == null)
15          storeSolution
16          this ! Backtrack
17        else
18          k = valueHeuristic.selectValue
19          store.makeAssignment(v, k)
20          this ! Enforce_Consistency
21
22      case Enforce_Consistency:
23        running = true
24        if (store.enforceConsistency)
25          forall (c in connectedConstraints)
26            forall (v in c.remoteVariables)
27              if (v.hasChanged)
28                v.remoteAgent !
29                  Pruning(v.name, v.domain)
30        else
31          consistent = false
32        if (master)
33          this ! Wait_For_Consistency
34        running = false
35
36      case Pruning(varName, domain):
37        v = store.findVariable(varName)
38        v.domain = domain
39        this ! Enforce_Consistency
```

Figure 6: Part one of the pseudo code for the agents. The receive method is called whenever a message arrives. Exclamation mark indicates communication to an agent.

```
1     case Wait_For_Consistency:
2       forall (a in A)
3         if (a.isRunning)
4           this ! Wait_For_Consistency
5           return
6       forall (a in A)
7         if (a.inconsistent)
8           this ! Backtrack
9           return
10      this ! Select_Next_Master
11
12    case Backtrack:
13      store.forbidLastAssignment
14      store.undoLastAssignment
15      if (store.stillInconsistent)
16        previousMaster ! Backtrack
17      else
18        this ! Start_Search
19
20    case Select_Next_Master:
21      forall (a in A)
22        a ! Find_Best_Variable(this)
23
24    case Find_Best_Variable(theMaster):
25      v = selectionHeuristic.selectVariable
26      k = v.fitness
27      theMaster ! Fitness(k, this)
28
29    case Fitness(fitness, actor):
30      if (fitness > bestFitness)
31        bestFitness = fitness
32        bestActor = actor
33      fitnessReplies += 1
34      if (fitnessReplies == A.size)
35        master = false
36        bestActor ! Make_Master(this)
37  }
38 }
```

Figure 7: Part two of the pseudo code for the agents.

```
1  // M is a vector of vectors representing tasks assigned
2  // to a machine. Each task is specified by its starting
3  // task start time t, task duration d
4
5  boolean Jobshop_Search(M)
6    if store.enforceConsistency
7      if M ≠ ∅
8        m ← selectCriticalMachine(M)
9        sort tasks in m in ascending values of t.min()
10       for each i = 1,...,n
11         for each j = 1,...,n
12           if (i ≠ j)
13             impose mᵢ.t + mᵢ.d ≤ mⱼ.t
14         M' ← M \ mᵢ
15         if Jobshop_Search(M')
16           return true
17         else
18           return false
19       return false
20     else
21       store solution
22       return true
23   else
24     return false
25
26 vector selectCriticalMachine(M)
27   for each mᵢ ∈ M
28     min ← min(min(mᵢ.t₀), min(mᵢ.t₁), ..., min(mᵢ.tₙ))
29     max ← max(max(mᵢ.t₀ + mᵢ.d₀), max(mᵢ.t₁ + mᵢ.d₁), ...,
30               max(mᵢ.tₙ + mᵢ.dₙ))
31     supply ← max − min
32     demand ← ∑ mᵢ.dᵢ
33     critical ← supply − demand
34   return machine mᵢ with the lowest value of critical
```

Figure 8: The pseudo code for the ordering search.

4 Experimental Evaluation

For our experiments, we used the JaCoP solver [8]. The agent system is written using actors in Scala [14]. The experiments were run on a Mac Pro with two 3.2 GHz quad-core Intel Xeon processors running Mac OS X 10.6.2 with Java 6 and Scala 2.8.1. These two processors have a common cache and memory bus for each of their four cores. The parallel version of our solver is described in detail in [16]. We used a timeout of 30 minutes for all the experiments. All experiments were run 20 times, giving a standard deviation of less than 5 %.

We ran several standard benchmark scheduling problems described in [19, 10]. The characteristics of the problems are listed in Table 1. These are all JSSP, where n jobs with m tasks are to be scheduled on m different machines. We study the case of non-preemptive scheduling.

Table 1: Characteristics of the problems for the global constraint model.

Problem	Jobs	Tasks	Variables	Constraints	Optimum
LA01	10	5	61	56	666
LA04	10	5	61	56	590
LA05	10	5	61	56	593
MT06	6	6	43	43	55

We created two DisCOP models of each problem: one for our version of DCP with global constraints, and the other representing the traditional case with only primitive constraints. When using our model, each agent represents one machine. It contains one global cumulative constraint with n tasks [2], to ensure no overlap of tasks.

In the primitive model, each agent represents one variable. For the problems we studied, the primitive constraints are binary in the sense that they only contain two variables. Our primitive constraint models did not use table constraints. Instead, they used the constraint $start_i + duration_i \leq start_j \vee start_j + duration_j \leq start_i$ for every pair of tasks, to ensure no overlap. This constraint is technically a binary constraints, since the duration is a constant. These primitive constraints replace cumulative for JSSP since we only have one instance of each resource.

Each problem was started with no prior knowledge of the optimal solution. Hence, the domains of the variables representing the start time tasks were $\{0..1000\}$. When using many resources, translating a single cumulative constraint into a table constraint requires primitive constraints in every time point. For many problems, this could result in an excessive number of rows in the table constraint. This is often infeasible due to memory size.

4.1   Experimental Results

The results for finding and proving the the optimal solution are shown in Table 2 and Table 3. Clearly, the primitive representation of the problems rarely found the optimal within the 30 minute timeout. The only exception was MT06, the simplest problem we tested. Still, finding the solution for MT06 took almost 30 times as long as the global model.

Table 2: Execution time in seconds that the global constraint model took to find the optimal solution and the best solution found within the 30 minute timeout.

Problem	Time to find optimum	Time to prove optimum	Best solution
LA01, Global	3.8	4.0	666
LA04, Global	10.8	12.1	590
LA05, Global	0.7	0.97	593
MT06, Global	3.0	3.0	55

Our model of DCP with global constraints in each agent gives superior performance in our experiments. The traditional model with only one variable per agent never managed to prove the optimality within the timeout. This performance increase comes partly from the fact that we can order variables before we start search. When agents control only one variable, this type of ordering is not possible. In this case, adding the ordering constraints will mostly serve to increase the number of pruning messages that need to be sent.

Table 3: Execution time in seconds for the primitive constraint model and best solution found within the timeout.

Problem	Time to find optimum	Time to prove optimum	Best solution
LA01, Primitive	Timeout	Timeout	936
LA04, Primitive	Timeout	Timeout	976
LA05, Primitive	Timeout	Timeout	720
MT06, Primitive	87.7	Timeout	55

When we turn off the ordering of tasks, the performance drops significantly for the global model. However, even though we could not prove optimality without ordering, we found better solutions within the timeout than the primitive model for almost all problems. Hence, the benefit of our model is not simply in the use of advanced search, but also in the use of global constraints.

Although our search is synchronous, using asynchronous search would probably not benefit the traditional primitive model much. Our model would probably still be better, because in our experiments we use a simulated distribution, thus minimizing the penalty of sending many messages. The primitive model communicates many more messages to reach the consistency fixpoint. When using a network, the communication would be an order of magnitude more time consuming than on a shared-memory multicore machine.

Using asynchronous search would bring benefits to both the global constraint model and the primitive one. However, the search space of CP is exponential with regard to domain size. Parallel search only gives a polynomial speed-up [15]. Hence the performance advantage of the global constraint model is likely to remain, even though the model with one variable per agent allows for more parallelism.

We also created a third model, where each agent control several variables, but have no global constraints. Just as for the global representation, each agent models one machine. However, the cumulative constraint has been replaced by the same kind of constraints as in the primitive model for every pair of tasks.

The performance of this third model, shown in Table 4, was better than that of the single variable per agent model. However, the performance was usually much lower than of the global constraint representation. For the simplest problem it was slightly faster. But for the most difficult problem, it did not find the optimum within the timeout.

The performance benefit of our model of DCP is not simply because of our advanced search. The ordering of tasks on each machine is possible in the model with several variables per agent but without global constraints. However, the pruning is much weaker when there are no global constraints.

Our results for the model in Table 4, compared to the results in Table 3, illustrate the cost of communication. We get much better performance than the scenario of one variable per agent, despite using the same constraints. Hence, the difference between the performance of these two models comes mostly from the communication of prunings.

The cost of communication depends on the agent framework. However, we ran our experiments on a shared-memory machine. Running on a cluster, with network communication, would increase the performance penalty of communication severely. If anything, our experiments over estimate the competitiveness of traditional DisCSP models.

Table 4: Results for multi-variable agents, *without* global constraints, but *with* ordering.

Problem	Time to find optimum	Time to prove optimum	Best solution
LA01	3.8	6.9	666
LA04	Timeout	Timeout	667
LA05	0.47	9.7	593
MT06	2.5	2.6	55

5   Conclusions

In this paper, we have introduced a completely new model of distributed constraint programming. Unlike any work we are aware of, we equip each agent with a full constraint solver. Our model is the the only one we have seen published that can use global constraints. It also allows advanced search, during which we can order tasks before assigning actual start times of scheduling problems.

By equipping each agent with a full constraint solver, we allow much more efficient modeling of problems. Unlike most work on DisCSP, we do not translate our models into table constraints. This allows us to communicate domains and constraints between agents during the search. Such communication is much more efficient than that of traditional DisCSP. Reducing communication is a major concern in DisCSP solving.

Our main conclusion of this paper is that both global constraints and advanced search are needed in order to solve complex scheduling problems using distributed constraint programming. Traditional work on DisCSP has focused on agents that only control one variable and only have primitive constraints. We conclude that these older models are unlikely to offer good performance for real world use, even when using asynchronous search.

Another conclusion is that using the traditional approach to DisCSP of one variable per agent should be very well motivated. Using one variable per agent may provide better robustness and privacy. However, we show that letting agents control several variables, using global constraints, and using advanced search methods are all important for good performance.

References

[1] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[2] Y. Caseau and F. Laburthe. Improving branch and bound for job-shop scheduling with constraint propagation. In M. Deza, R. Euler,

and I. Manoussakis, editors, *Combinatorics and Computer Science*, volume 1120 of *Lecture Notes in Computer Science*, pages 129–149. Springer Berlin / Heidelberg, 1996.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63–75, Feb 1985.

[4] R. Ezzahir, C. Bessiere, M. Belaissaoui, and E. Bouyakhf. DisChoco: A platform for distributed constraint programming. In *Proceedings of IJCAI-07 Workshop on Distributed Constraint Reasoning*, pages 16–27, 2007.

[5] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5:66–77, Jan 1983.

[6] A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward bounding for distributed COPs. *Journal of Artificial Intelligence Research*, 34:61–88, Feb 2009.

[7] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, Aug 1978.

[8] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, July 2003.

[9] J. Kvarnstrom and P. Doherty. Automated planning for collaborative uav systems. In *2010 11th International Conference on Control Automation Robotics Vision (ICARCV)*, pages 1078–1085, Dec 2010.

[10] S. R. Lawrence. Resource-constrained project scheduling: An experimental investigation of heuristic scheduling techniques. Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh PA, 1984.

[11] T. Léauté, B. Ottens, and R. Szymanek. FRODO 2.0: An open-source framework for distributed constraint optimization. In *Proceedings of IJCAI-09 Workshop on Distributed Constraint Reasoning*, pages 160–164, Pasadena, California, USA, July 2009.

[12] A. Meisels and E. Kaplansky. Scheduling agents - distributed timetabling problems (DisTTP). In *Practice and Theory of Automated Timetabling IV*, volume 2740 of *Lecture Notes in Computer Science*, pages 166–177. Springer Berlin / Heidelberg, 2003.

[13] A. Meisels and R. Zivan. Asynchronous forward-checking for DisCSPs. *Constraints*, 12:131–150, 2007.

[14] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[15] V. Rao and V. Kumar. Superlinear speedup in parallel state-space search. In K. Nori and S. Kumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 338 of *Lecture Notes in Computer Science*, pages 161–174. Springer Berlin / Heidelberg, 1988.

[16] C. C. Rolf and K. Kuchcinski. Load-balancing methods for parallel and distributed constraint solving. In *IEEE International Conference on Cluster Computing*, pages 304–309, Sep/Oct 2008.

[17] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier Science Inc., New York, NY, USA, 2006.

[18] M. Salido. Distributed CSPs: Why it is assumed a variable per agent? In I. Miguel and W. Ruml, editors, *Abstraction, Reformulation, and Approximation*, volume 4612 of *Lecture Notes in Computer Science*, pages 407–408. Springer Berlin / Heidelberg, 2007.

[19] G. L. Thompson. *Industrial scheduling / edited by J.F. Muth and G.L. Thompson with the collaboration of P.R. Winters*. Prentice-Hall, Englewood Cliffs, N.J., 1963.

[20] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3:185–207, 2000.

[21] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In P. Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 43–66. Springer Berlin / Heidelberg, 2006.

APPENDIX

CONTROLLING THE PARALLELISM

The performance of parallel solving can sometimes be greatly improved by configuring the solving for that problem. Automatic parallelism in CP will often provide a speed-up when searching for the optimal solution. However, even better performance can be achieved if the parallelization is tuned specifically for that problem.

Ideally, automatic parallelism would not require *any* hints to, or configuration of, the solver by the problem modeler. However, in the absence of dynamic adaptation during the solving process, a few simple parameters can empower the programmer without requiring too much effort. The design of parallelism control in our solver is fairly similar to related work [10, 34].

To the provider of the constraint framework, the main challenge with automatic parallelism is to extract good parallel performance from the problem. It is very simple for problem modelers to use the parallelism, but sometimes choosing good values for the parameters in Table 1 and Table 2 is central in order to get a speed-up.

1   Controlling the Parallel Consistency

Table 1 presents the variables used to control the parallel consistency in our solver. There are fewer parameters compared to parallel search. In parallel consistency, we cannot decide the size of work unless we split up the constraint algorithms. However, constraint specific parallel consistency algorithms are not within the scope of this thesis.

The first parameter, `con_thread_count`, determines how many threads are available for running consistency. When we combine parallel search and parallel consistency, as described in detail in Paper II, this parameter determines the number of consistency threads per search thread. Without parallel search, we always have one search thread. For example, if we have one search thread and set `con_thread_count` to four, there will at most be four threads running in parallel. If we have eight search threads and set `con_thread_count` to eight, we will at most have 64 threads running at the same time. Hence, this parameter is useful for tuning the processor usage. In conjunction with `thread_count`, see Table 2, we have a good control over how much resources we allocate to parallel search compared to parallel consistency.

The second parameter, `con_split`, is used to control how many constraints are taken out of the constraint queue each time a consistency thread needs more work. If this is too low, the consistency threads will be idle more often while they are trying to get work out of the central queue. If setting it too high, the work load will be split unevenly between the threads. Setting a low value is not a problem if using a wait-free queue. A high value is not a major problem, since threads can take constraints from each other. However, getting a speed-up from parallel consistency can be a delicate process. Hence, this parameter may have an impact on the performance.

Table 1: The parameters for controlling the parallel consistency.

Parameter	Description	Unit
con_thread_count	The number of threads for consistency per solver	Absolute
con_split	The number of constraints per consistency thread	Absolute

2   Controlling the Parallel Search

The parameters we use to control parallel search are presented in Table 2. The first parameter, thread_count, determines how many solvers will be running search. If no value is given, there will be as many search threads as there are processor cores available to the Java runtime.

The second parameter, start_split, is the point from which we allow splitting of work. This value is absolute, for instance, setting it to two will prevent splitting of work at the first two levels of the search tree. This parameter may increase performance if the first variable should always have its minimal value. This is useful when,for example, finding optimal Golomb rulers.

The third search parameter, end_split, controls the depth in the search tree to which splitting is allowed. This parameter can be very useful for ensuring that we do not send work that is too small to benefit from parallelism. It is given as a percentage of the total number of variables that we label in the problem. For instance, setting it to 0.8 for a labeling with 20 variables will disallow splitting after depth 16.

The fourth parameter, split_min, determines the minimum size of a domain that we send. For instance, if a domain has five values, and split_min is five we will not split this domain since we have to keep at least one value for the sending solver. When used in conjunction with end_split, this variable allows more finely tuned control of the amount of work we send.

The fifth variable, `split_size`, controls how much of the current domain will be sent. The value has to be larger than zero and smaller than one, as it represents the percentage of values in the split domain. If the domain size multiplied with `split_size` is larger than `split_min`, we will not send work. The main use of `split_size` is to ensure that we do not send too much work to one solver, but rather split the same domain several times if there are several free solvers.

Table 2: The parameters for controlling the parallel search.

Parameter	Description	Unit
thread_count	The number of search threads	Absolute
start_split	The depth at which sharing of work starts to be allowed	Absolute
end_split	The depth at which sharing of work is no longer allowed	Percent
split_min	The minimum size of a domain that can be shared	Absolute
split_size	The part of the domain that is to be shared	Percent