Parallel Solving in Constraint Programming

Carl Christian Rolf and Krzysztof Kuchcinski
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.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Performance

Keywords

Constraint Programming, Parallel Search, Parallel Consistency

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.

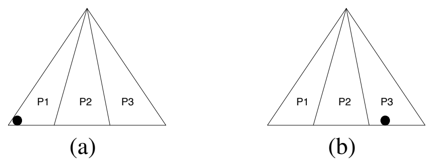

Figure 1: The position of the solution in a search tree affects the benefit of parallelism.

The examples in Fig. 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 Fig. 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 Fig. 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.

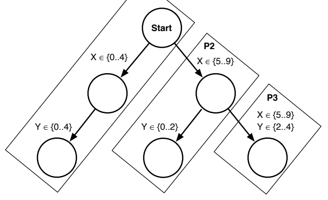

Figure 2: Parallel search in constraint programming.

In this paper, we will 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 handles $X_i \geq \frac{min(X_i)+max(X_i)}{2}$. An example of such data parallelism in CP is depicted in Fig. 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.

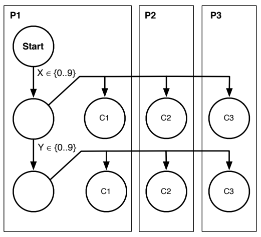

Figure 3: Parallel consistency in constraint programming.

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.

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 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.

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.

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 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, e.g., $X+Y \leq Z$.

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 usually 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 Fig. 4, this model is described in greater detail in [10] and in Fig. 6(b). 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.

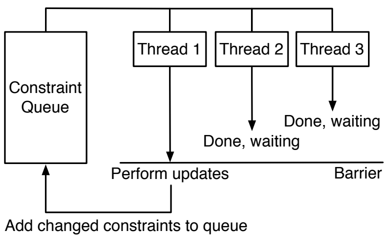

Figure 4: The execution model for parallel consistency.

Consistency enforcement is iterative. When the threads are ready, the constraint queue is split between them, and an iteration of consistency can begin. This procedure will be repeated until we reach a fixpoint, i.e., the constraints no longer changes any domain. 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 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 Fig. 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.

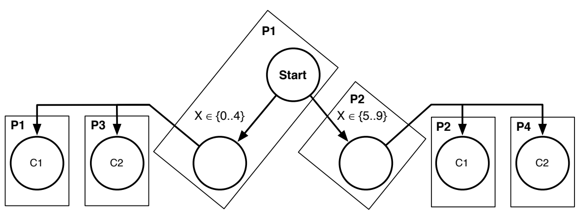

Figure 5: An example of combining parallel search and parallel consistency.

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 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, i.e., 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 a solution, or we proceed recursively with 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 Fig. 6. 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 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 ensure that the total number of consistency threads exceeds the number of processors.

As Fig. 6 shows, 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 Fig. 6(a), line 15 requires synchronization while we wait for the consistency threads to finish. In Fig. 6(b), 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 also have to be synchronized.

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

(a)

```
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 $PC \neq \emptyset$
7           $PC \leftarrow PC \setminus SC$
8           while $SC \neq \emptyset$
9               $SC \leftarrow SC \setminus c$
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                      $PC \leftarrow PC \cup cd$
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
```

(b)

Figure 6: The combined parallel search and parallel consistency algorithm. Parallel depth-first search (a), slave program for parallel consistency (b).

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 Fig. 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 usually 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.

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. Our parallelized 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 problem characteristics 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.

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 Fig. 7 and Fig. 8. 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.

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.

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

Consistency Threads	Search Threads			
per Search Thread	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	Search Threads			
per Search Thread	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

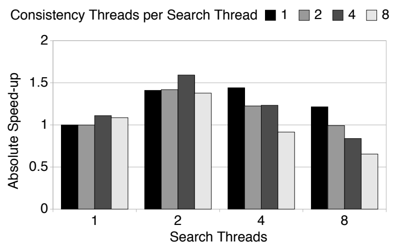

Figure 7: Speed-up when searching for 200 solutions to 100-Sudoku.

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 Fig. 9 and Fig. 10. 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.

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 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.

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

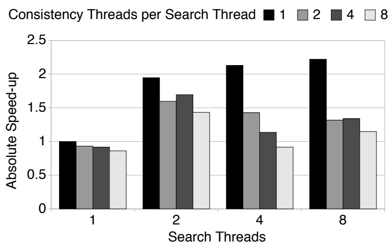

Figure 8: Speed-up when searching for $5\,000$ solutions to 100-Sudoku.

Table 4 and Table 5. As shown by Fig. 9, 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, Fig. 10 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	Search Threads			
per Search Thread	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

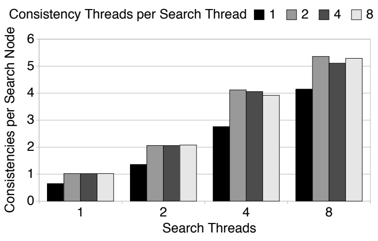

Figure 9: Consistency enforcements per search node when searching for 200 solutions to Sudoku.

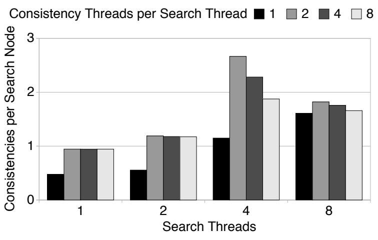

Figure 10: Consistency enforcements per search node when searching for $5\,000$ solutions to Sudoku.

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 Fig. 11, 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 Fig. 12 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.

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

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

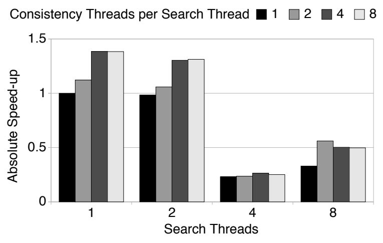

Figure 11: Speed-up when searching for one solution to 550-Queens.

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

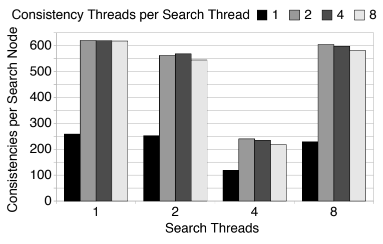

Figure 12: Consistency enforcements per search node when searching for one solution to Queens.

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 within the barrier. Since pruning is usually monotonic, this should be possible.

7. REFERENCES

[1] G. Chu, C. Schulte, and P. J. Stuckey. Confidence-based work stealing in parallel constraint programming. In I. Gent, editor, *Fifteenth International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241, Lisbon, Portugal, Sept. 2009. Springer-Verlag.

[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 (TODAES)*, 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. V. Hentenryck. Parallelizing constraint programs transparently. In C. Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 514–528. Springer, 2007.

[6] T. Nguyen and Y. Deville. A distributed arc-consistency algorithm. *Sci. Comput. Program.*, 30(1-2):227–250, 1998.

[7] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *AAAI/IAAI*, pages 359–366, 1998.

[8] V. N. Rao and V. Kumar. Superlinear speedup in parallel state-space search. In *Proceedings of the Eighth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 161–174, London, UK, 1988. Springer-Verlag.

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

[10] C. C. Rolf and K. Kuchcinski. Parallel consistency in constraint programming. *PDPTA '09: The 2009 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2:638–644, 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. *J. Parallel Distrib. Comput.*, 70(2):183–188, 2010.