Parallel Consistency in Constraint Programming

Carl Christian Rolf and Krzysztof Kuchcinski
Department of Computer Science, Lund University, Sweden

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.

Keywords: Parallel Consistency, Constraint Programming

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.

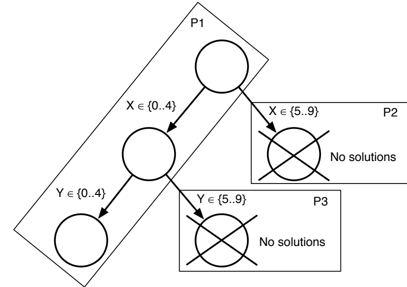

Fig. 1: Parallel search in constraint programming.

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 Fig. 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.

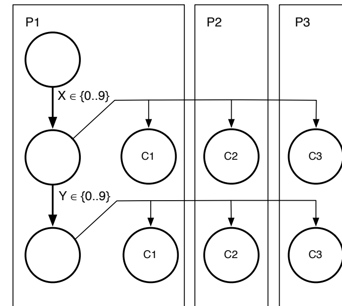

Fig. 2: Parallel consistency in constraint programming.

Fig. 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.

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.

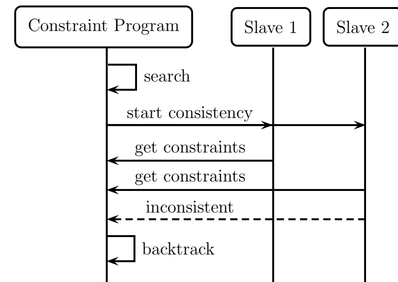

Fig. 3: Model of parallel consistency with two consistency threads. The dashed line indicates the final return to the constraint program. In this example it leads to a backtrack of the search procedure.

Our model of parallel consistency is depicted in Fig. 3. The pseudo-code for our model is presented in Fig. 4. 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.

```
// 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


// 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
```

Fig. 4: The parallel depth-first search algorithm. Constraint program (top), slave program (bottom).

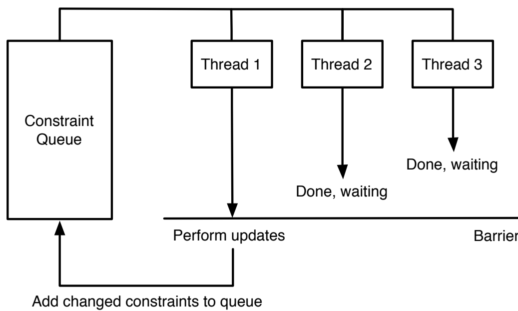

Fig. 5: The execution model for parallel consistency.

As depicted in Fig. 5, 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.

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.

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, then 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 $O(nlog\,n)$. 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.

The domain that is used during update at the barrier in Fig. 5 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, otherwise 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 depicted in Fig. 6. 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.

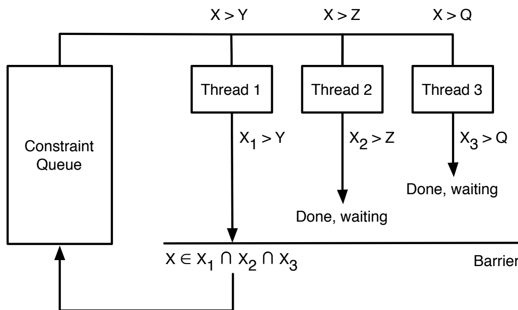

Fig. 6: The model of thread local updates.

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 Fig. 7. The results for thread local intermediate domains are presented in Table 3 and Fig. 8. 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 7 and Fig. 8 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.

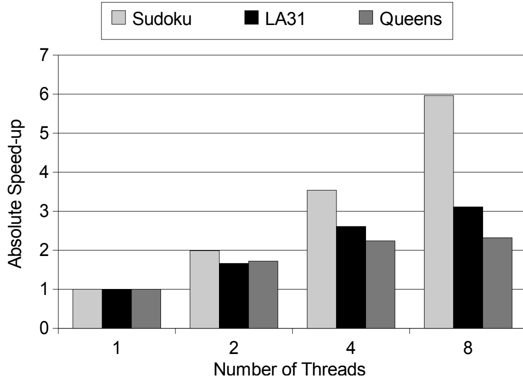

Fig. 7: Absolute speed-up when using shared intermediate domains.

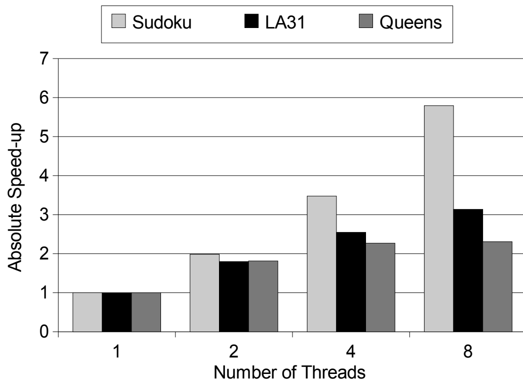

Fig. 8: Absolute speed-up when using thread local intermediate domains.

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. 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.

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 Fig. 9 and Fig. 10. 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

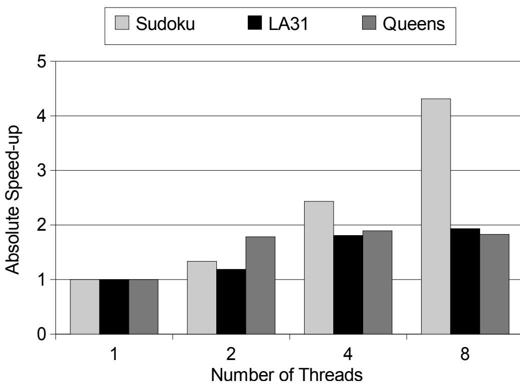

Fig. 9: Absolute speed-up when using shared intermediate domains.

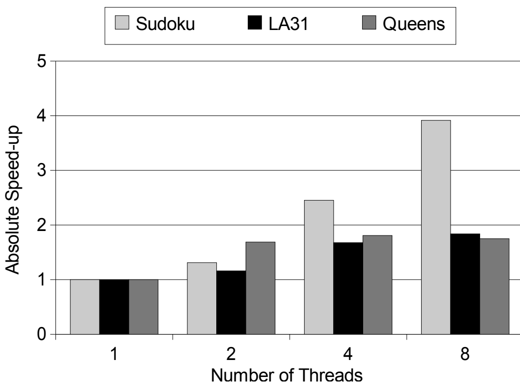

Fig. 10: Absolute speed-up when using thread local intermediate domains.

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.

4.4 Processor Load

As depicted in Figures 11 to 13, 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 Fig. 12. Queens needs two iterations, which is the cause of the spike in Fig. 13.

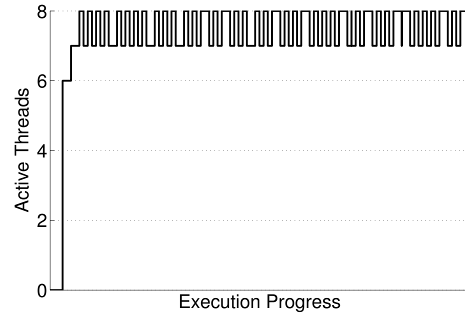

Fig. 11: The processor load of Sudoku using eight threads.

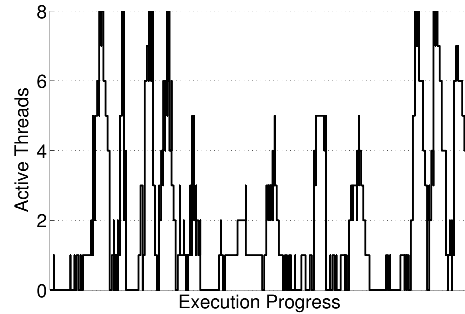

Fig. 12: The processor load of LA31 using eight threads.

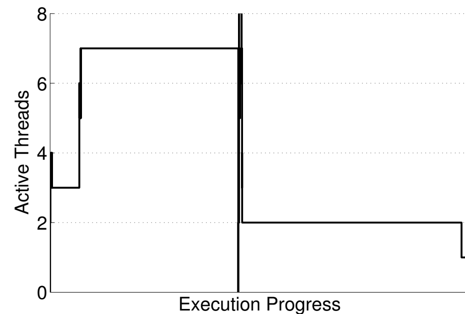

Fig. 13: The processor load of Queens using eight threads.

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.

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.

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. L. Pape, and W. Nuijten, *Constraint-Based Scheduling*. Norwell, MA, USA: Kluwer Academic Publishers, 2001.

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

[3] K. Kuchcinski, “Constraints-driven scheduling and resource assignment,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 3, pp. 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*. Cambridge, MA, USA: MIT Press, 1998.

[6] L. Michel, A. See, and P. V. Hentenryck, “Parallelizing constraint programs transparently,” in *CP*, ser. Lecture Notes in Computer Science, C. Bessiere, Ed., vol. 4741. Springer, 2007, pp. 514–528.

[7] T. Nguyen and Y. Deville, “A distributed arc-consistency algorithm,” *Sci. Comput. Program.*, vol. 30, no. 1-2, pp. 227–250, 1998.

[8] L. Perron, “Search procedures and parallelism in constraint programming,” in *CP ’99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*. London, UK: Springer-Verlag, 1999, pp. 346–360.

[9] J.-F. Puget, “A fast algorithm for the bound consistency of alldiff constraints,” in *AAAI/IAAI*, 1998, pp. 359–366.

[10] 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*. London, UK: Springer-Verlag, 1988, pp. 161–174.

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

[12] ——, “State-copying and recomputation in parallel constraint programming with global constraints,” in *PDP ’08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, Feb 2008, pp. 311–317.

[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*, 1998, pp. 86–100.

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

[15] C. Schulte and M. Carlsson, “Finite domain constraint programming systems,” in *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence, F. Rossi, P. van Beek, and T. Walsh, Eds. Amsterdam, The Netherlands: Elsevier Science Publishers, 2006, ch. 14, pp. 495–526.