Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence

# Parallel Restarted Search

**Andre A. Cire**
Carnegie Mellon University
Pittsburgh, PA 15213, USA
acire@andrew.cmu.edu

**Serdar Kadioglu**
Oracle America Inc.
Burlington, MA 01803, USA
serdar.kadioglu@oracle.com

**Meinolf Sellmann**
IBM Research
Yorktown Heights, NY 10598, USA
meinolf@us.ibm.com

**Abstract**

We consider the problem of parallelizing restarted backtrack search. With few notable exceptions, most commercial and academic constraint programming solvers do not learn no-goods during search. Depending on the branching heuristics used, this means that there are little to no side-effects between restarts, making them an excellent target for parallelization. We develop a simple technique for parallelizing restarted search deterministically and demonstrate experimentally that we can achieve near-linear speed-ups in practice.

## Introduction

The strong trend of more cores per CPU and the advent of cloud computing motivates efficient, parallel, low communication approaches for constraint satisfaction and optimization problems. Efforts toward parallelizing tree-search in constraint programming (CP) and operations research (OR) have been undertaken for many years (Yun and Epstein 2012; Xu, Tschoke, and Monien 1995; Gendron and Crainic 1994). The most recent advances in parallel search in CP regard search-tree parallelization. In particular, (Moisan, Gaudreault, and Quimper 2013) devise a scheme for deterministic low-communication parallel limited discrepancy search (LDS) which scales nicely for their application. Moreover, (Régin, Rezgui, and Malapert 2013) devise a low-communication parallel approach based on the generation of very many subproblems.

Early on it was found that parallelizing branching algorithms can lead to super-linear speed-ups (Rao and Kumar 1988). This of course meant nothing else than that the sequential algorithms were not optimally efficient. Randomization and restarting the search have largely eliminated this defect and mark the current state-of-the-art in logic programming. In CP, restarted search is by now commonly employed in solvers like Gecode (Schulte, Tack, and Lagerkvist 2010) or IBM Ilog CP Optimizer (IBM 2011). Consequently, we need to devise methods to parallelize restarted search algorithms. As an additional complication, industrial users of CP solvers frequently require deterministic parallelization. With "deterministic" it is not meant that randomization through random number generators could not be used, but that, provided with the same seed, the computation should advance in the same fashion as the sequential search would, no matter how many processors are employed.

With few noteworthy exceptions (Jain, Sabharwal, and Sellmann 2011; Ohrimenko, Stuckey, and Codish 2009; Katsirelos 2009) most solvers do not conduct no-good learning in CP. Learning branching heuristics aside, this means that there are no side-effects between different restarts. For this case, we introduce a very easy-to-use, low communication parallelization technique for restarted constraint solvers. We prove in theory and demonstrate experimentally that this technique allows the solver to realize near-linear speed-ups. Another advantage of the proposed method is that the resulting parallel search is deterministic, i.e. the solver will return the *same solution*, no matter how many compute cores we use and in what order messages between them arrive. This is a very important property for commercial clients of solvers like Ilog CP Optimizer as they commonly employ the CP solver within a larger framework. To debug these algorithmic environments it is absolutely essential that the solution provided by the solver does not change in a re-run.

## Parallel Restarted Search

In CP we often observe that runtime distributions are *heavy-tailed* (Gomes, Selman, and Kautz 2000). An effective way to overcome this undesirable property is to restart the search after some fail-limit is reached. Of course, the question then is how we should set these consecutive restart fail-limits. Based on the analysis of so-called bandit problems, the Luby restart strategy (Luby, Sinclair, and Zuckerman 1993) is provably optimal.

Restarting search is one of the core advances in systematic SAT and CP solving in the past decade. In SAT, restarts in combination with no-good learning have led to significant performance improvements. In CP, where constraint propagators typically do not provide the reasons for the domain filtering that they conduct, no-good learning is not commonly employed. While there is currently an effort to bring the strengths of no-good learning and generalized constraint propagation together (Jain, Sabharwal, and Sellmann 2011; Ohrimenko, Stuckey, and Codish 2009; Katsirelos 2009), solvers like Choco (CHOCO-Team 2010), Gecode (Schulte, Tack, and Lagerkvist 2010), and IBM Ilog CP Optimizer (IBM 2011) do not employ no-good learning.

One advantage of *not* performing no-good learning is that the different restarts offer potential for parallelization. Note that standard tree-search parallelization is somewhat problematic for searches that have fail-limits. This is particularly true when the parallelization is required to be deterministic, which is frequently the case for debugging purposes and industrial users as outlined above. To replicate the sequential behavior we would need to collect the *exact same* failures that a sequential run encounters. Since we cannot know a-priori where these failures will occur we either run the risk of searching a superfluous part of the search tree, or we need to parallelize work close to the leaf level of the search tree – which results in a high communication overhead from sharing small sub-problems, especially in a highly distributed hardware environment.

## Parallel Luby

We can avoid all this by executing the restarts, as a whole, in parallel. Note that this is different from a strategy that some parallel SAT solvers follow like, e.g., ManySAT (Hamadi, Jabbour, and Lakhdar 2009) which executes different *restart strategies* in parallel and achieves about a factor 2 speed-up on 8 processors.

A recent proposal for parallelizing systematic search in constraint programming was presented in (Yun and Epstein 2012). As in ManySAT, this method, called SPREAD, runs a portfolio of different search strategies in the first phase. After a fixed time-limit, SPREAD enters a master-slave work-sharing parallelization of restarted search, whereby the search experience from the portfolio-phase is used to determine the way how the search is partitioned in the splitting phase. While exact speed-ups over sequential search are not explicitly mentioned in (Yun and Epstein 2012), from the scatter plot Mistral-CC vs. SPREAD-D in Fig. 4 we can glean that the method achieves roughly a factor 10 speed-up when using 64 processors, whereby the authors mention that the speed-ups level off when using more than 64 workers. Note also that neither ManySAT, nor SPREAD, nor the method from (Régin, Rezgui, and Malapert 2013) achieve deterministic parallelization. This is notably different from the method developed by (Moisan, Gaudreault, and Quimper 2013) who parallelize least discrepancy search and achieve about 75% efficiency for their application.

Focusing on restarted search methods, we propose to use the (potentially scaled) Luby restart strategy and execute it, as a whole, in parallel. Parallel universal restart strategies have been studied in the literature before (see (Luby and Ertel 1994) and its correction in (Shylo, Middelkoop, and Pardalos 2011)). In (Shylo, Middelkoop, and Pardalos 2011), non-universal strategies are studied, which are not directly applicable in practice. Moreover, speed-ups over the best non-universal sequential strategy are inherently sub-linear.

We will show: In contrast to optimal non-universal restart strategies, the *universal* Luby strategy *can* be parallelized with near-perfect efficiency. Furthermore, we can achieve this with an extremely simple method that requires no sophisticated parallel programming environment such as MPI. To boot, we can achieve a deterministic parallelization,

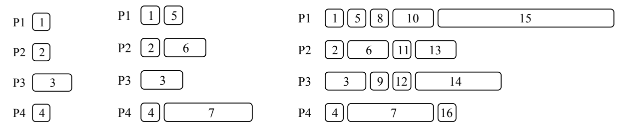


Figure 1: Parallel Luby Schedules Considered by Core 4 in a 4 Core Machine. Widths correspond to the length of a restart. Numbers give the index of the restart in the sequential Luby sequence.

which neither (Luby and Ertel 1994) nor (Shylo, Middelkoop, and Pardalos 2011) provide.

The Luby strategy works as follows. We start with fail limit $\{1\}$. Now we repeat the following until a solution is found: 1. Repeat all fail-limits considered so far. 2. Run a restart with double the fail-limit considered last. This results in a sequence like this: $\{1\}$. Repeat all so far: $\{1\}$. Double the last fail-limit: $\{2\}$. Repeat all so far: $\{1, 1, 2\}$. Double the last: $\{4\}$. Repeat all: $\{1, 1, 2, 1, 1, 2, 4\}$. Double: $\{8\}$. Since it does not affect the proof of optimality, in practice we may multiply each of these Luby fail-limits with some constant $a \in \mathbb{N}$ (typical values are $a = 32$ or $a = 100$).

For the purpose of parallelizing restarts, the Luby strategy is very attractive. Consider the situation when using, for example, geometric restarts: The majority of the time is spent within the last restart. If the geometric factor is 2, up to 50% of the total time is spent in the last restart. Without parallelizing this final search, the most we can ever hope for is a speed-up of two, no matter how many processors we employ. The Luby strategy does not suffer from this drawback.

To parallelize Luby, we propose the following. Each parallel search process has *its own local copy* of a scheduling class which assigns restarts and their respective fail-limits to processors. This scheduling class computes the next Luby restart fail-limit and adds it to the processor that has the lowest number of accumulated fails so far, following an *earliest-start-time-first* strategy. Like this, the schedule is filled and each process can infer which is the next fail-limit that it needs to run based on the processor it is running on – without communication. Overhead is negligible in practice since the scheduling itself runs extremely fast compared to CP search, and communication is limited to informing the other processes when a solution has been found.

In Figure 1 we show three parallel Luby schedules that the process on processor 4 would consider. To get its first fail-limit (see leftmost schedule) it would fill the schedule by assigning consecutive Luby fail-limits to processors. Assuming ties are broken by smaller processor number, we would assign fail-limit $\{1\}$ to processor 1, $\{1\}$ to processor 2, and $\{2\}$ to processor 3, and again $\{1\}$ to processor 4. To get its second fail-limit (see middle schedule) the schedule would again need to be filled until the next fail-limit is assigned to processor 4. We assign fail-limit $\{1\}$ to processor 1 and fail-limit $\{2\}$ to processor 2. The following fail-limit $\{4\}$ is assigned to processor 4 because it has the lowest accumulated fail-limit: 2 fails accumulated for processor 1, 3 fails for processor 2, 2 fails for processor 3, and only 1 fail so far for processor 4. The rightmost schedule shows how the process continues.

## Analysis

The question now is obviously how well the simple earliest-start-time-first strategy parallelizes our search. Let us study the speed-up that we may expect by this method.

**Theorem 1.** *The parallel Luby restart schedule achieves asymptotic linear speed-ups when the number of processors $p$ is constant and the number of restarts grows to infinity.*

*Proof.* The core to the proof is the realization that, before the Luby strategy considers the first restart of length $2^k$, it incurs at least $k2^k$ failures (*not restarts!*). For example, before the first restart with fail-limit $4 = 2^2$ Luby will run restarts with fail-limits 1, 1, 2, 1, 1, 2. In total, it has encountered $8 = 2 * 2^2$ failures earlier.

To assess the speed-up of our parallelization method we will need to bound from above the ratio of the total elapsed parallel time over the total sequential time. Note that this ratio will only grow when the sequential runtime is lower and the parallel time stays the same. Consequently, w.l.o.g. we can assume that the successful restart is the one that determines the parallel makespan. To give an example, consider the rightmost schedule in Figure 1. Assume the successful restart is the 16th. By assuming that we already get a solution at the very end of Restart 15 we have lowered the sequential runtime, but the parallel runtime (the width of the schedule) remains the same.

Now, consider the situation when the last restart was scheduled. By the earliest-start-time rule, we know that all processors are working in parallel on earlier restarts up to the elapsed time when the last restart begins. If we consider again the right-most schedule in Figure 1, note that processors 1 through 4 are all busy with restarts 1 to 14 from the start to the begin of Restart 15.

Let $T_B$ be the total number of failures before the final restart, and let $T_S$ be the failures allowed in the final, successful restart. Finally, assume that $2^k$ is the longest restart we encounter sequentially before we find a solution. Note this implies $T_S \leq 2^k$.

By our observations above, we know that the work on $T_B$ is perfectly parallelized. Consequently, the parallel time is at most $\frac{T_B}{p} + T_S$. Now let us consider the ratio of parallel elapsed time over sequential time: $\frac{\frac{T_B}{p}+T_S}{T_B+T_S} \leq \frac{1}{p} + \frac{T_S}{T_s+T_B}$. And thus, using our initial observation that $T_B \geq k2^k \geq kT_S$: $\frac{\frac{T_B}{p}+T_S}{T_B+T_S} \leq \frac{1}{p} + \frac{1}{1+k}$. Consequently, as the number of restarts (and hence $k$) grows, the speed-up approaches $p$. □

In modern cloud computing we often deal with a heterogeneous pool of compute cores that could be heavily distributed. Thanks to the fact that parallelizing restarts requires communication only upon termination, in our method we can employ loosely coupled compute clusters that could even be located on different continents. However, we will need to take into account differing hardware characteristics that would lead to different average speeds per failure.

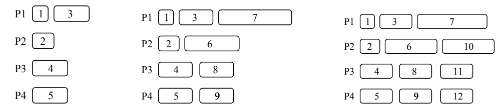


Figure 2: Parallel Luby Schedules Considered by Core 4 on a 4-Core distributed system with different processor speed. Widths correspond to the time of a restart needed on the respective core. Numbers give the index of the restart in the Luby sequence.

Assuming that we know the speeds of the compute cores available to us, we will modify our parallelization method slightly as follows: Given are $p$ processors, each associated with a different slow-down factor $s_i \geq 1$ which measures the slow-down of the processor $i$ compared to the fastest core available to us which has slow-down factor 1.0. The estimated time needed to complete $2^k$ failures on core $i$ is then $s_i 2^k$. Thus, the time it takes to complete a restart differs depending on the core we assign it to. We take this into account when building the schedule by assigning the next fail-limit to a processor such that the total makespan of the schedule is minimized. That is, when assigning a restart to a core we multiply by that core's slow-down factor and follow an *earliest-completion-time-first* strategy.

We illustrate the method in Figure 2. We assume the slow-down factor of cores 1 and 2 is 1.0 and 1.5, respectively, and the slow-down of cores 3 and 4 is 2.0. Our first fail-limit is $\{1\}$ which requires time 1 on core 1, time 1.5 on core 2, and time 2 on either core 3 or 4. By the earliest-completion-time-first rule this fail-limit is assigned to core 1. The second fail-limit is also $\{1\}$. The fastest way to complete this restart is to assign it to core 2 where it will take time 1.5. The third fail-limit is $\{2\}$. If we assign it to core 1 it will be completed after 2 additional time units and 3 time units since we began our computation. If we assign it to core 2 would require 3 time units and would complete after 4.5 time units since the start of the computation. Finally, if we assigned this restart to either core 3 or 4 it would require 4 time units and thus complete later than by assigning it to core 1. Hence, we assign the third restart to core 1. The fourth fail-limit is $\{1\}$ and will complete earliest at time 2 when assigned to core 3. The fifth restart, finally, also has fail-limit $\{1\}$ and is the first restart assigned to core 4.

Accordingly we schedule restart 6 with fail-limit $\{2\}$ on processor 2, and restarts 7 and 8 with fail-limit $\{1\}$ on processor 3 and 4, respectively. Figure 2 shows the first three schedules that core 4 would consider.

**Theorem 2.** *The adjusted parallel Luby restart schedule achieves asymptotic maximum speed-ups when the number of processors $p$ is constant and the number of restarts grows to infinity.*

*Proof.* Going back to the proof of Theorem 1 we make two observations: First, in a homogeneous cluster where all processors have the same speed, the earliest-start-time-first scheduling rule is the same as the earliest-completion-time-first rule. Second, the key to the proof is really the relation of the work that is fully parallelized compared to the work that is not or only imperfectly parallelized.

To assess the latter, we will consider the total work that could have been performed in parallel to the last successful restart (which we will again assume, w.l.o.g., determines the makespan). Due to the earliest-completion-time scheduling rule we know that, when the successful restart is scheduled on processor $i$, each other processors must already be busy with earlier restarts to a point where they could not conduct the work required to complete the final restart before processor $i$ is done with it. If we denote with $T_S$ the time required for the final restart on the fastest processor, then the total work that could be conducted while processor $i$ is working on the last restart is bounded from above by $T_S(p-1)$.

Denote with $T_{par}$ the total elapsed time when using all $p$ processors, and denote with $T_{seq}$ the sequential time needed on the fastest processor with slow-down factor 1. Then, $T_{seq} \geq T_{par} \sum_j \frac{1}{s_j} - (p-1)T_S$. Set again $k$ such that $2^k$ is the highest fail-limit in any restart before we find a solution. We have, $T_S \leq 2^k$ and $T_{seq} \geq (k+1)2^k$. Then, $1 \geq \frac{T_{par}}{T_{seq}} \sum_j \frac{1}{s_j} - \frac{(p-1)2^k}{k+1}$ and thus $\frac{T_{par}}{T_{seq}} \leq \frac{1}{\sum_j \frac{1}{s_j}} + \frac{p-1}{(k+1)\sum_j \frac{1}{s_j}}$. Again, as the number of restarts grows to infinity, $k$ grows to infinity, and we asymptotically achieve a speed-up of $\sum_j \frac{1}{s_j}$, which is the maximum we can hope for for the given compute power. $\square$

## Deterministic Parallel Execution

Finally, let us consider the problem of executing the parallel schedule deterministically. As discussed earlier, in industrial practice we often need to ensure that the solutions provided by the program are identical, no matter how many processors are assigned to solve the problem or in what order messages between processes arrive. In Figure 1, on top of the actual fail-limit expressed by the width of the blocks, we denote the index of each restart in the Luby sequence. For the scheduler it is obviously easy to keep track of this index, since it fills the schedule consecutively anyway.

Now let us assume that we have access to as many deterministic random number generators as restarts needed to solve the problem. While this is not perfect in theory, in practice we could, for example, use the same deterministic random number generator with a different deterministically randomized seed for each restart. Then, we can deterministically assign each restart its own random number generator – and thus make the parallel execution deterministic, no matter how many processors are used. The only caveat is that a restart that follows later in the Luby sequence may find a solution before the first successful restart in the sequence (note that the latter may have a larger fail-limit!). To ensure that the first solution is returned that would be found if the restarts were executed in sequential order, all we need to do is to complete restarts with lower indices even when a restart with higher index reports that a solution had been found. Note that this additional work does not affect our proofs of Theorems 1 and 2, so we can still expect near-linear speed-ups in practice.

## Comparison with Existing Techniques

Typically, parallel CP approaches focus on parallelizing the tree-search itself, see e.g., (Perron 1999). As said earlier, this creates problems when using restarted search methods when deterministic parallelization is required. Moreover, parallelizing tree-search requires sophisticated load balancing schemes. Although simple in concept, parallelizing tree-search is a real challenge and not easy to implement efficiently. (see e.g. (Rolf and Kuchcinski 2010; Boivin, Gendron, and Pesant 2008; Yun and Epstein 2012)). In a heterogeneous compute environment with potentially very high communication costs this task becomes even more daunting.

(Bordeaux, Hamadi, and Samulowitz 2009) propose a different technique for parallelizing search in CP, namely to split the search space "evenly" first and then search the different parts in parallel. Experiments on $n$-queens problems show very good scaling behavior. The issue with this technique is of course that, in general, an "even" split of the search space is hardly achievable by any generic technique: some parts will be proven unsatisfiable more quickly, while others take a long time to search through. In fact, if we could guarantee an even split, then we could also count the numbers of solutions effectively — a problem that is known to be #P-complete, which is believed to be strictly harder than the original NP-hard problem we were trying to solve. Fittingly, the results reported in (Bordeaux, Hamadi, and Samulowitz 2009) show a much more modest scaling behavior for real-world SAT problems, where speed-ups of about a factor 8 are achieved with 64 cores (whereby no-good learning additionally complicates things).

(Régin, Rezgui, and Malapert 2013) attack this problem by generating very many subproblems and then handing them to workers dynamically as these become idle. Experiments show that this leads to reasonable load-balancing, that is slightly outperforming work-stealing, at very low communication costs. However, no determinism is achieved this way, nor is the basis of parallelization a restarted algorithm which is the state-of-the-art in sequential CP. (Moisan, Gaudreault, and Quimper 2013)'s scheme for parallelizing LDS also enjoys very low communication costs and is furthermore deterministic under the same no-side-effects assumption we make. For cases where LDS works this is a great technique, with reported speed-ups around 75% on a special application. On the downside, we know that LDS can exhibit the same heavy-tailed runtime distributions that randomizations and restarts overcome.

In summary, parallelizing restarts offers an easy-to-implement alternative that can provide deterministic solutions based on the state-of-the-art sequential algorithm. However, we need to be sure that there are no side-effects between restarts as they would be caused by no-good learning or learning branching methods. In the following experimental section we will therefore analyze the real speed-ups achieved in practice.

| | | QCP | | Magic Square | | | | | Costas Array | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 30 | 40 | 12 | 13 | 14 | 1 5 | 16 | 15 | 16 | 17 | 18 | 19 | 20 |
| Proc. | % Solved | 76 | 86 | 100 | 100 | 54 | 8 | 2 | 100 | 100 | 100 | 44 | 8 | 0 |
| 1 | #Cpoints | 32.3M | 8.1M | 4M | 23.6M | 72.9M | 94.5M | 83.2M | 0.16M | 0.64M | 7M | 28.1M | 31.6M | 33.7M |
| | avg. Time | 1378 | 784 | 58.1 | 364 | 1354 | 1951 | 1987 | 6.52 | 28.9 | 320 | 1523 | 1868 | 2000 |
| Proc. | % Solved | 82 | 100 | 100 | 100 | 82 | 26 | 4 | 100 | 100 | 100 | 68 | 10 | 2 |
| 2 | #Cpoints | 49.8M | 9.4M | 4.1M | 23.8M | 111M | 185M | 175M | 0.16M | 0.64M | 7M | 46.6M | 68.1M | 63.8M |
| | avg. Time | 940 | 338 | 27.1 | 176 | 944 | 1807 | 1961 | 2.93 | 12.4 | 151 | 1132 | 1849 | 1989 |
| Proc. | % Solved | 86 | 100 | 100 | 100 | 98 | 34 | 4 | 100 | 100 | 100 | 92 | 24 | 2 |
| 4 | #Cpoints | 64M | 9.6M | 4.1M | 24.1M | 139M | 332M | 318M | 0.17M | 0.67M | 7.1M | 57.5M | 112M | 115M |
| | avg. Time | 749 | 165 | 12.8 | 86.1 | 566 | 1590 | 1942 | 1.56 | 6.94 | 87.5 | 799 | 1761 | 1977 |
| Proc. | % Solved | 90 | 100 | 100 | 100 | 100 | 60 | 12 | 100 | 100 | 100 | 94 | 50 | 6 |
| 8 | #Cpoints | 96.6M | 9.9M | 4.2M | 24.3M | 143M | 504M | 614M | 0.18M | 0.69M | 7.2M | 69.3M | 185M | 209M |
| | avg. Time | 555 | 76.4 | 6.8 | 44.6 | 313 | 1314 | 1876 | 0.79 | 3.12 | 40.8 | 482 | 1522 | 1959 |
| Proc. | % Solved | 98 | 100 | 100 | 100 | 100 | 82 | 16 | 100 | 100 | 100 | 98 | 78 | 14 |
| 16 | #Cpoints | 146M | 9.8M | 4.2M | 24.1M | 141M | 714M | 1170M | 0.19M | 0.77M | 7.3M | 72.6M | 265M | 374M |
| | avg. Time | 310 | 27.83 | 3.75 | 24.7 | 159 | 930 | 1813 | 0.54 | 2.20 | 22 | 298 | 1150 | 1879 |
| Proc. | % Solved | 98 | 100 | 100 | 100 | 100 | 94 | 30 | 100 | 100 | 100 | 100 | 95 | 16 |
| 32 | #Cpoints | 167M | 9.9M | 4.2M | 24.1M | 141M | 881M | 2091M | 0.05M | 0.24M | 1.4M | 7.9M | 73.2M | 27.0M |
| | avg. Time | 208 | 17.66 | 2.21 | 12.9 | 81.6 | 595 | 1677 | 0.38 | 1.36 | 11.9 | 138 | 694 | 1783 |

Table 1: Performance on Quasigroup Completion, Magic Square and Costas Array Problems of increasing size utilizing between 1 and 16 processors. The time limit was set to 2,000 seconds.

## Experimental Results

We first evaluated our proposed approach on 3 distinct problem domains: Quasi-Group Completion (QCP), Magic Squares, and Costas Arrays (Costas 1984). All experiments were conducted on Intel Xeon 2.4 GHz computers with varying number of cores as required by each run. We employed Gecode solver with 2,000 second timeout and all experiments presented use a Luby scaling factor of 64.

The results of our first experiments are summarized in Table 1. We present the percent of solved instances, number of choice points, and average time for problem type under varying number of processors. These experiments followed minimum domain variable ordering with random tie breaking and minimum value heuristic, except that the first five variables are selected uniformly at random.

For QCP, we generated two sets of 50 instances of orders 30 and 40 with 288 and 960 holes respectively. We observe that instances in the first set are harder to solve. Furthermore, the number of solved instances increases when we have more processors available. At the same time, the average elapsed time decreases. We cannot assess whether speed-ups are linear however, since instances that were not solved with fewer processors contribute a mere 2,000 seconds to the average, even though their actual solving times would be higher. With 16 and 32 processors, we can almost solve all of the instances. When we consider the instances of order 40, we observe that we can solve them all starting from 2 processors. The average elapsed time on these instances drops from 338 seconds on 2 processors to 17.66 seconds on 32 processors, which is slightly super-linear.

We next considered Magic Squares and for each order used 50 random seeds. We see that magic square problems of smaller orders are indeed easy to solve. For order 12 and 13, we find that 2,000 seconds are enough to solve the instance with just one processor, for all 50 random seeds. This gives a clear idea how well our technique scales when we can assess speed-ups between runs that solve all instances. For example for order 13, the runtime drops from 364 sequential seconds to 12.9 seconds on 32 cores, which is 88% efficient. For higher orders, we need more processors, e.g., for order 14, 8 processors are needed to solve all instances. From here on we observe almost linear speed-ups: The average elapsed time using 8 processors is 313 seconds, and for 32 processors, the time is down to 81.6 seconds – almost exactly 4-times faster than the average time with 8 processors.

The results for the Costas arrays confirm the observations made on magic squares problem. Again, speed-ups almost perfectly linear: Solving order 17 takes 320 sequential seconds, 151 on two, 87.5 on four, 40.8 on eight, 22 on 16, and 11.9 seconds on 32 processors.

Our last experiment was performed on the instances from the Minizinc Challenge using Gecode solver. This solver uses a search strategy based on active failure count with a decay of 0.99 and geometric restarts (scale factor 1,000 and base 2), and implements a tree-search parallelization technique. We compare GE code's default parallelization with our parallel Luby restart strategy (PLR). We use a static failure count reset at the beginning of each restart. To randomize the restarts, again, the first five branching variables are selected uniformly at random. Notice that the default Gecode

| Gecode | 1 Proc. | 2 Proc. | | 4 Proc. | | 8 Proc. | | 16 Proc. | | 32 Proc. | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Solved | Solved | Speed-up | Solved | Speed-up | Solved | Speed-up | Solved | Speed-up | Solved | Speed-up |
| Default | 9 | 9 | 3.71 | 9 | 1.19 | 9 | 2.5 | 9 | 1.55 | 8 | 2.27 |
| PLR | 10 | 10 | 2.59 | 10 | 3.86 | 10 | 12.5 | 11 | 13.2 | 11 | 35.7 |
| PLR+L | 9 | 8 | 1.42 | 9 | 3.14 | 10 | 5.45 | 11 | 15.4 | 11 | 11.2 |

Table 2: Performance on instances from Minizinc 2012 Challenge utilizing between 1 and 32 processors. The time limit was set to 2,000 seconds.

solver learns failure counts of variables for branching, i.e., it carries information between restarts, while PLR resets these values at the beginning of each restart and thus does not have any side-effects. We also considered a variant of our technique that allows learning between trials that are performed on the same processor[1], denoted by PLR+L.

Table 2 presents our results averaged over 11 instances from the Minizinc 2012 Challenge that can be solved with at least one of the approaches within 2,000 seconds. These instances are from sb, non-fast, non-awful and amaze2 classes. Since the default Gecode solver is not deterministic in parallel mode, we report the median values from five runs.

The sequential Gecode solver solved 9 of these instances within the time limit. When 2 processors are employed, the run time improves super-linearly as we can see by the average super-linear speed-up of 3.71 on 2 cores. Surprisingly, using more cores then decays the default performance.

Parallel Luby restarts allows the solver to realize near-linear speed-ups when more processors are utilized. There is a visible improvement as we scale the parallelization. Curiously, PLR+L does not perform better and actually decays the performance for large numbers of processors, which could be a hint why the default's performance also decays. In any case, we see that the method presented here scales very well even for larger numbers of processors for these challenging benchmarks.

## Conclusion

We introduced a simple technique for parallelizing restarted search in CP that requires almost no communication and is very easy to implement. Theoretical analysis and experimental results on various CP benchmarks showed that executing Luby restarts in parallel leads to linear speed-ups in theory and in practice. Most importantly, the method provides deterministic parallelization.

## Acknowledgments

The second author was supported by Paris Kanellakis fellowship at Brown University when conducting the work contained in this document. This document reflects his opinions only and should not be interpreted, either expressed or implied, as those of his current employer.

[1]We thank Christian Schulte for providing us with a version of Gecode solver that enabled implementation of this strategy.

## References

Boivin, S.; Gendron, B.; and Pesant, G. 2008. A load balancing procedure for. parallel constraint programming. CIRRELT-2008-32.

Bordeaux, L.; Hamadi, Y.; and Samulowitz, H. 2009. Experiments with massively parallel constraint solving. *IJCAI*.

CHOCO-Team. 2010. choco: an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes.

Costas, J. 1984. A study of a class of detection waveforms having nearly ideal range–doppler ambiguity properties. *Proceedings of the IEEE* 72(8):996–1009.

Gendron, B., and Crainic, T. 1994. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research* 42:1042–2066.

Gomes, C.; Selman, B.; and Kautz, H. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning* 67–100.

Hamadi, Y.; Jabbour, S.; and Lakhdar, S. 2009. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation* 6:245–262.

IBM. 2011. IBM ILOG CPLEX Optimization Studio 12.2.

Jain, S.; Sabharwal, A.; and Sellmann, M. 2011. A general nogood-learning framework for pseudo-boolean multi-valued sat. *AAAI*.

Katsirelos, G. 2009. *Nogood Processing in CSPs*. Ph.D. Dissertation, University of Toronto.

Luby, M., and Ertel, W. 1994. Optimal parallelization of las vegas algorithms. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '94, 463–474.

Luby, M.; Sinclair, A.; and Zuckerman, D. 1993. Optimal speedup of las vegas algorithms. *Information Processing Letters* 47:173–180.

Moisan, T.; Gaudreault, J.; and Quimper, C.-G. 2013. Parallel discrepancy-based search. In Schulte (2013), 30–46.

Ohrimenko, O.; Stuckey, P.; and Codish, M. 2009. Propagation via lazy clause generation. *Constraints* 14:357–391.

Perron, L. 1999. Search procedures and parallelism in constraint programming. *CP* 346–360.

Rao, V., and Kumar, V. 1988. Superlinear speedup in parallel state-space search. *Foundations of Software Technology and Theoretical Computer Science* 161–174.

Régin, J.-C.; Rezgui, M.; and Malapert, A. 2013. Embarrassingly parallel search. In Schulte (2013), 596–610.

Rolf, C., and Kuchcinski, K. 2010. Parallel solving in constraint programming. In *MCC 2010: Third Swedish Workshop on Multi-Core Computing*.

Schulte, C.; Tack, G.; and Lagerkvist, M. Z. 2010. Modeling and programming with gecode.

Schulte, C., ed. 2013. *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*. Springer.

Shylo, O.; Middelkoop, T.; and Pardalos, P. 2011. Restart strategies in optimization: parallel and serial cases. *Parallel Computing* 37(1):60–68.

Xu, C.; Tschoke, S.; and Monien, B. 1995. Performance evaluation of load distribution strategies in parallel branch and bound computations. *SPDP* 402–405.

Yun, X., and Epstein, S. 2012. A hybrid paradigm for adaptive parallel search. *CP* 720–734.