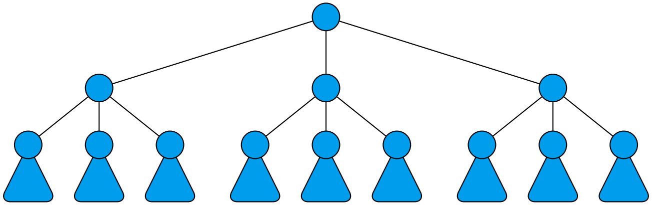
# Parallel Search

Ciaran McCreesh and Patrick Prosser

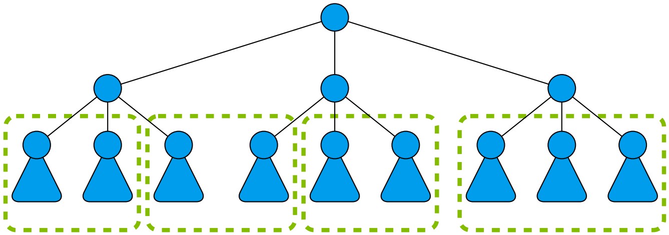
## This Week's Lectures

- Search and Discrepancies
- Parallel Constraint Programming
- Parallel Search
    - Embarrassingly Parallel Search
    - Work stealing
    - Confidence Based Work Stealing

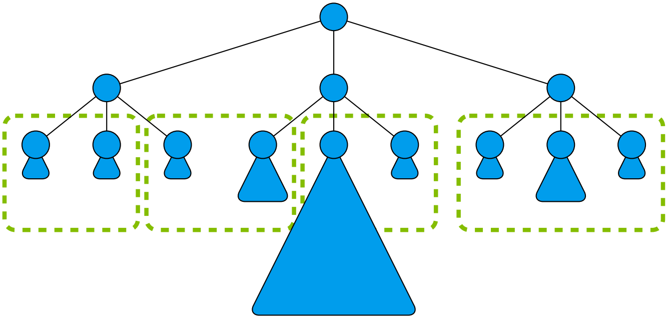
## Fixed Parallel Tree Search, Again



## Fixed Parallel Tree Search, Again



## Fixed Parallel Tree Search, Again



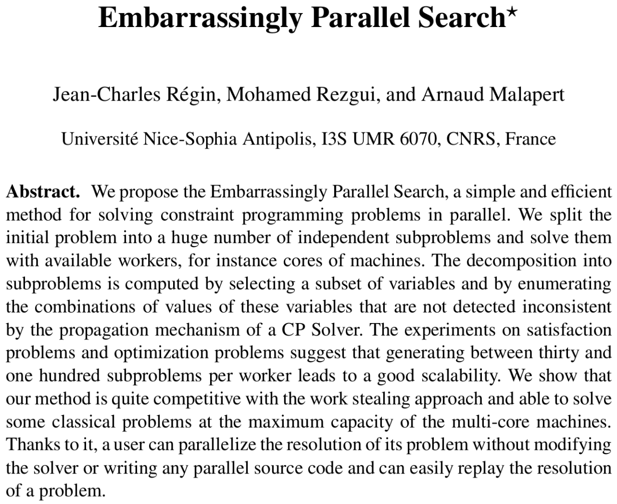
## What We Need

- We need a large parallelisable portion of the algorithm, *and* good work balance, or we don't get much of an improvement.

## Embarrassingly Parallel Search

- If we create *n* subproblems, chances are we'll get poor balance.
- We can't tell beforehand where the really hard subproblems will be.
- What if we create *lots* of subproblems, and distribute them dynamically?

# Embarrassingly Parallel Search



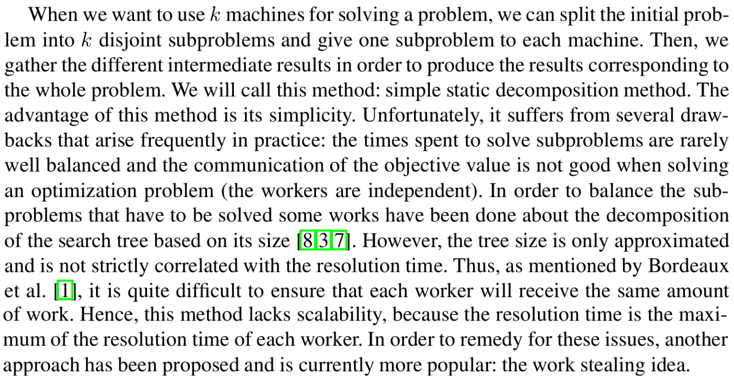
**Embarrassingly Parallel Search**★

Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert

Université Nice-Sophia Antipolis, I3S UMR 6070, CNRS, France

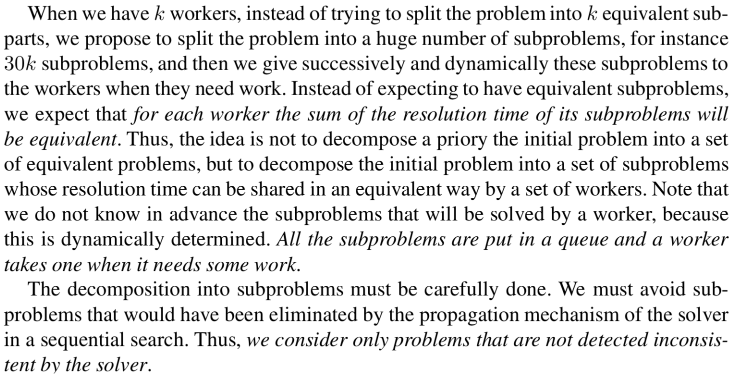
**Abstract.** We propose the Embarrassingly Parallel Search, a simple and efficient method for solving constraint programming problems in parallel. We split the initial problem into a huge number of independent subproblems and solve them with available workers, for instance cores of machines. The decomposition into subproblems is computed by selecting a subset of variables and by enumerating the combinations of values of these variables that are not detected inconsistent by the propagation mechanism of a CP Solver. The experiments on satisfaction problems and optimization problems suggest that generating between thirty and one hundred subproblems per worker leads to a good scalability. We show that our method is quite competitive with the work stealing approach and able to solve some classical problems at the maximum capacity of the multi-core machines. Thanks to it, a user can parallelize the resolution of its problem without modifying the solver or writing any parallel source code and can easily replay the resolution of a problem.

## Embarrassingly Parallel Search



When we want to use $k$ machines for solving a problem, we can split the initial problem into $k$ disjoint subproblems and give one subproblem to each machine. Then, we gather the different intermediate results in order to produce the results corresponding to the whole problem. We will call this method: simple static decomposition method. The advantage of this method is its simplicity. Unfortunately, it suffers from several drawbacks that arise frequently in practice: the times spent to solve subproblems are rarely well balanced and the communication of the objective value is not good when solving an optimization problem (the workers are independent). In order to balance the subproblems that have to be solved some works have been done about the decomposition of the search tree based on its size [8,3,7]. However, the tree size is only approximated and is not strictly correlated with the resolution time. Thus, as mentioned by Bordeaux et al. [1], it is quite difficult to ensure that each worker will receive the same amount of work. Hence, this method lacks scalability, because the resolution time is the maximum of the resolution time of each worker. In order to remedy for these issues, another approach has been proposed and is currently more popular: the work stealing idea.

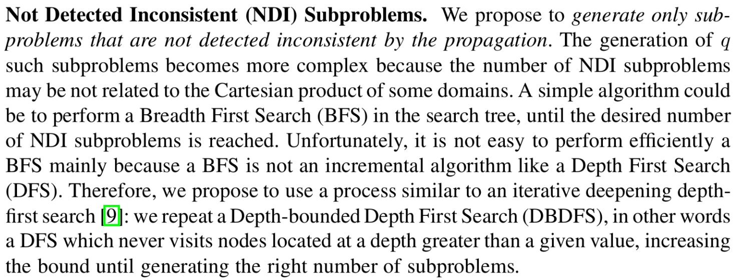
## Embarrassingly Parallel Search



When we have $k$ workers, instead of trying to split the problem into $k$ equivalent subparts, we propose to split the problem into a huge number of subproblems, for instance $30k$ subproblems, and then we give successively and dynamically these subproblems to the workers when they need work. Instead of expecting to have equivalent subproblems, we expect that *for each worker the sum of the resolution time of its subproblems will be equivalent*. Thus, the idea is not to decompose a priory the initial problem into a set of equivalent problems, but to decompose the initial problem into a set of subproblems whose resolution time can be shared in an equivalent way by a set of workers. Note that we do not know in advance the subproblems that will be solved by a worker, because this is dynamically determined. *All the subproblems are put in a queue and a worker takes one when it needs some work.*

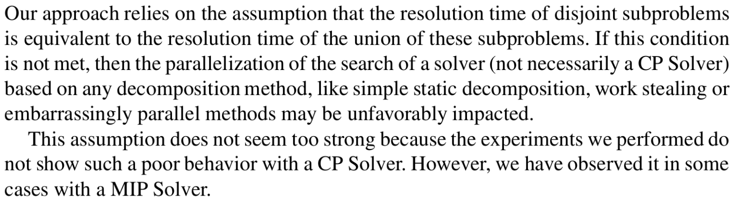
The decomposition into subproblems must be carefully done. We must avoid subproblems that would have been eliminated by the propagation mechanism of the solver in a sequential search. Thus, *we consider only problems that are not detected inconsistent by the solver*.

## Embarrassingly Parallel Search



**Not Detected Inconsistent (NDI) Subproblems.** We propose to *generate only subproblems that are not detected inconsistent by the propagation*. The generation of $q$ such subproblems becomes more complex because the number of NDI subproblems may be not related to the Cartesian product of some domains. A simple algorithm could be to perform a Breadth First Search (BFS) in the search tree, until the desired number of NDI subproblems is reached. Unfortunately, it is not easy to perform efficiently a BFS mainly because a BFS is not an incremental algorithm like a Depth First Search (DFS). Therefore, we propose to use a process similar to an iterative deepening depth-first search [9]: we repeat a Depth-bounded Depth First Search (DBDFS), in other words a DFS which never visits nodes located at a depth greater than a given value, increasing the bound until generating the right number of subproblems.

## Embarrassingly Parallel Search



Our approach relies on the assumption that the resolution time of disjoint subproblems is equivalent to the resolution time of the union of these subproblems. If this condition is not met, then the parallelization of the search of a solver (not necessarily a CP Solver) based on any decomposition method, like simple static decomposition, work stealing or embarrassingly parallel methods may be unfavorably impacted.

This assumption does not seem too strong because the experiments we performed do not show such a poor behavior with a CP Solver. However, we have observed it in some cases with a MIP Solver.

## Embarrassingly Parallel Search

**Optimization Problems**

In case of optimization problems we have to manage the best value of the objective function computed so far. Thus, the operations are slightly modified.

- The TaskDefinition operation consists of computing a partition of the initial problem $P$ into a set $S$ of subproblems.
- The TaskAssignment operation is implemented by using a queue. Each time a subproblem is defined it is added to the back of the queue. The queue is also associated with the best objective value computed so far. When a worker needs some work, the master gives it a subproblem from the queue. It also gives it the best objective value computed so far.
- The TaskResultGathering operation manages the optimal value found by the worker and the associated solution.

Note that there is no other communication, that is when a worker finds a better solution, the other workers that are running cannot use it for improving their current resolution. So, if the absence of communication may increase our performance, this aspect may also lead to a decrease of performance. Fortunately, we do not observe this bad behavior in practice.

## Embarrassingly Parallel Search

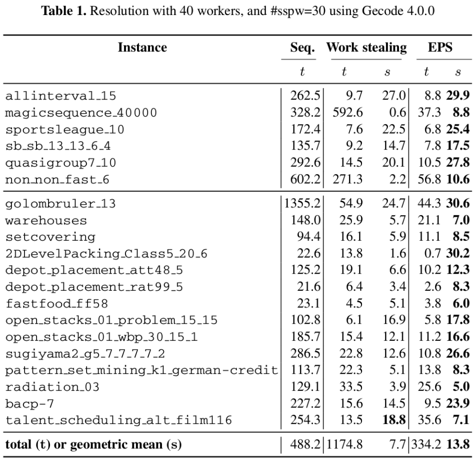
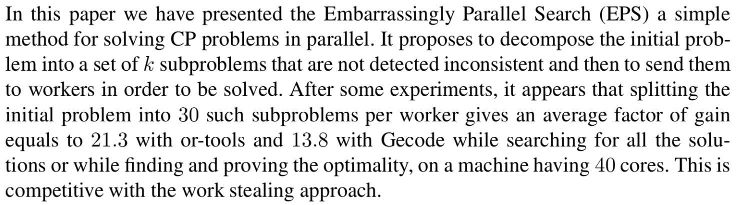


**Table 1.** Resolution with 40 workers, and #sspw=30 using Gecode 4.0.0

| **Instance** | **Seq.** | **Work stealing** | | **EPS** | |
|---|---|---|---|---|---|
| | $t$ | $t$ | $s$ | $t$ | $s$ |
| `allinterval_15` | 262.5 | 9.7 | 27.0 | 8.8 | **29.9** |
| `magicsequence_40000` | 328.2 | 592.6 | 0.6 | 37.3 | **8.8** |
| `sportsleague_10` | 172.4 | 7.6 | 22.5 | 6.8 | **25.4** |
| `sb_sb_13_13_6_4` | 135.7 | 9.2 | 14.7 | 7.8 | **17.5** |
| `quasigroup7_10` | 292.6 | 14.5 | 20.1 | 10.5 | **27.8** |
| `non_non_fast_6` | 602.2 | 271.3 | 2.2 | 56.8 | **10.6** |
| `golombruler_13` | 1355.2 | 54.9 | 24.7 | 44.3 | **30.6** |
| `warehouses` | 148.0 | 25.9 | 5.7 | 21.1 | **7.0** |
| `setcovering` | 94.4 | 16.1 | 5.9 | 11.1 | **8.5** |
| `2DLevelPacking_Class5_20_6` | 22.6 | 13.8 | 1.6 | 0.7 | **30.2** |
| `depot_placement_att48_5` | 125.2 | 19.1 | 6.6 | 10.2 | **12.3** |
| `depot_placement_rat99_5` | 21.6 | 6.4 | 3.4 | 2.6 | **8.3** |
| `fastfood_ff58` | 23.1 | 4.5 | 5.1 | 3.8 | **6.0** |
| `open_stacks_01_problem_15_15` | 102.8 | 6.1 | 16.9 | 5.8 | **17.8** |
| `open_stacks_01_wbp_30_15_1` | 185.7 | 15.4 | 12.1 | 11.2 | **16.6** |
| `sugiyama2_g5_7_7_7_7_2` | 286.5 | 22.8 | 12.6 | 10.8 | **26.6** |
| `pattern_set_mining_k1_german-credit` | 113.7 | 22.3 | 5.1 | 13.8 | **8.3** |
| `radiation_03` | 129.1 | 33.5 | 3.9 | 25.6 | **5.0** |
| `bacp-7` | 227.2 | 15.6 | 14.5 | 9.5 | **23.9** |
| `talent_scheduling_alt_film116` | 254.3 | 13.5 | **18.8** | 35.6 | **7.1** |
| **total (t) or geometric mean (s)** | 488.2 | 1174.8 | 7.7 | 334.2 | **13.8** |

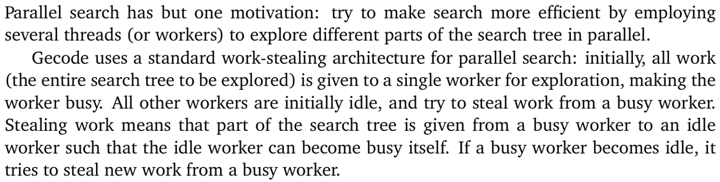
## Embarrassingly Parallel Search


In this paper we have presented the Embarrassingly Parallel Search (EPS) a simple method for solving CP problems in parallel. It proposes to decompose the initial problem into a set of $k$ subproblems that are not detected inconsistent and then to send them to workers in order to be solved. After some experiments, it appears that splitting the initial problem into 30 such subproblems per worker gives an average factor of gain equals to 21.3 with or-tools and 13.8 with Gecode while searching for all the solutions or while finding and proving the optimality, on a machine having 40 cores. This is competitive with the work stealing approach.

## Randomised Work Stealing

- What if we split work entirely dynamically?
- Whenever a worker is idle, have it steal a subproblem from another randomly selected worker.

## Randomised Work Stealing



Parallel search has but one motivation: try to make search more efficient by employing several threads (or workers) to explore different parts of the search tree in parallel.

Gecode uses a standard work-stealing architecture for parallel search: initially, all work (the entire search tree to be explored) is given to a single worker for exploration, making the worker busy. All other workers are initially idle, and try to steal work from a busy worker. Stealing work means that part of the search tree is given from a busy worker to an idle worker such that the idle worker can become busy itself. If a busy worker becomes idle, it tries to steal new work from a busy worker.

## Randomised Work Stealing

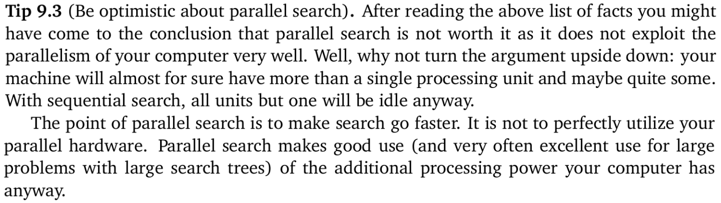
When using parallel search one needs to take the following facts into account (note that some facts are not particular to parallel search, check Tip 9.1: they are just more likely to occur):

- The order in which solutions are found might be different compared to the order in which sequential search finds solutions. Likewise, the order in which solutions are found might differ from one parallel search to the next. This is just a direct consequence of the indeterministic nature of parallel search.
- Naturally, the amount of search needed to find a first solution might differ both from sequential search and among different parallel searches. Note that this might actually lead to super-linear speedup (for $n$ workers, the time to find a first solution is less than $1/n$ the time of sequential search) or also to real slowdown.
- For best solution search, the number of solutions until a best solution is found as well as the solutions found are indeterministic. First, any better solution is legal (it does not matter which one) and different runs will sometimes be lucky (or not so lucky) to find a good solution rather quickly. Second, as a better solution prunes the remaining search space the size of the search space depends crucially on how quickly good solutions are found.

## Randomised Work Stealing

- As a corollary to the above items, the deviation in runtime and number of nodes explored for parallel search can be quite high for different runs of the same problem.
- Parallel search needs more memory. As a rule of thumb, the amount of memory needed scales linearly with the number of workers used.
- For parallel search to deliver some speedup, the search tree must be sufficiently large. Otherwise, not all threads might be able to find work and idle threads might slow down busy threads by the overhead of unsuccessful work-stealing.
- From all the facts listed, it should be clear that for depth-first left-most search for just a single solution it is notoriously difficult to obtain consistent speedup. If the heuristic is very good (there are almost no failures), sequential left-most depth-first search is optimal in exploring the single path to the first solution. Hence, all additional work will be wasted and the work-stealing overhead might slow down the otherwise optimal search.

## Randomised Work Stealing



**Tip 9.3** (Be optimistic about parallel search)**.** After reading the above list of facts you might have come to the conclusion that parallel search is not worth it as it does not exploit the parallelism of your computer very well. Well, why not turn the argument upside down: your machine will almost for sure have more than a single processing unit and maybe quite some. With sequential search, all units but one will be idle anyway.

The point of parallel search is to make search go faster. It is not to perfectly utilize your parallel hardware. Parallel search makes good use (and very often excellent use for large problems with large search trees) of the additional processing power your computer has anyway.

## Randomised Work Stealing

```
GolombRuler
        m[12] = {0, 1, 3, 7, 12, 20, 30, 44, 65, 80, 96, 122}
        m[12] = {0, 1, 3, 7, 12, 20, 30, 44, 65, 90, 105, 121}
        m[12] = {0, 1, 3, 7, 12, 20, 30, 45, 61, 82, 96, 118}
        ··· (additional solutions omitted)
        m[12] = {0, 2, 6, 24, 29, 40, 43, 55, 68, 75, 76, 85}

Initial
        propagators: 58
        branchers:   1

Summary
        runtime:      1:47.316 (107316.000 ms)
        solutions:    16
        propagations: 692676452
        nodes:        5313357
        failures:     2656663
        restarts:     0
        no-goods:     0
        peak depth:   24
```

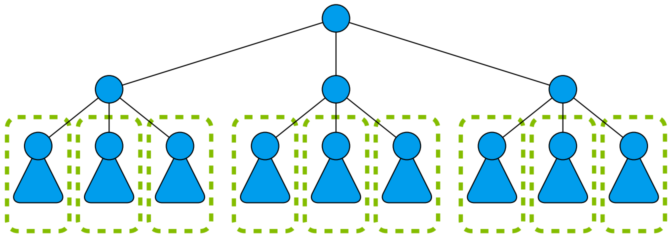
## Randomised Work Stealing

```
GolombRuler
        m[12] = {0, 1, 3, 7, 12, 20, 30, 44, 65, 80, 96, 122}
        m[12] = {0, 1, 3, 7, 12, 20, 30, 44, 65, 90, 105, 121}
        m[12] = {0, 1, 3, 7, 12, 20, 30, 45, 61, 82, 96, 118}
        ··· (additional solutions omitted)
        m[12] = {0, 2, 6, 24, 29, 40, 43, 55, 68, 75, 76, 85}

Initial
        propagators: 58
        branchers:   1

Summary
        runtime:      14.866 (14866.000 ms)
        solutions:    17
        propagations: 519555681
        nodes:        3836351
        failures:     1918148
        restarts:     0
        no-goods:     0
        peak depth:   26
```

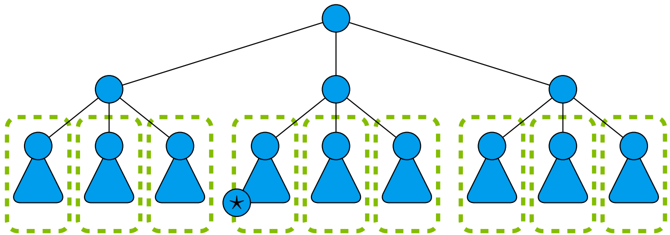
## Speedups from Parallel Tree Search

- If a decision problem is satisfiable, or for an optimisation problem, our speedups could be arbitrary: one worker might find a feasible or strong solution very quickly. In particular, *superlinear* speedups are possible, as are no speedups at all.
- For an unsatisfiable decision problem, or for an enumeration problem, our speedups can be at best linear (assuming we do not change the search tree): we *are* dividing up a fixed amount of work.
- If we do not communicate bounds, or if we do not preserve sequential ordering, we could get an *absolute slowdown*.

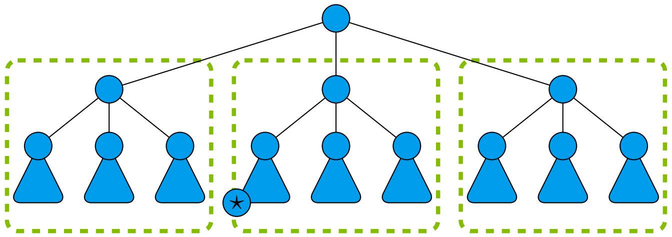
# Work Splitting Affects Search



## Work Splitting Affects Search



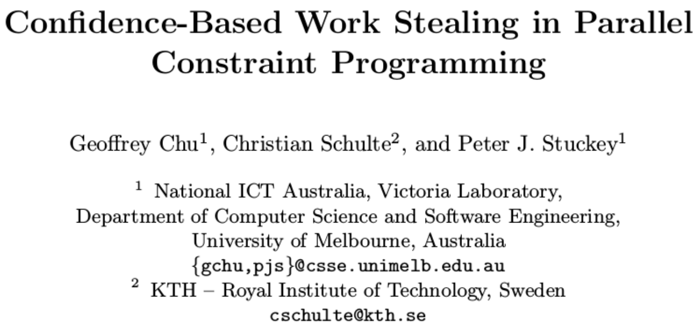
## Work Splitting Affects Search



## Work Splitting Affects Search

- For satisfiable instances and optimisation problems, where you split the work doesn't just affect balance. It also affects the amount of work to do. We just saw an example where *better* work balance gave *worse* performance, because it took longer to find a solution.
- Remember Harvey and Ginsberg's Limited Discrepancy Search?
  2. Value-ordering heuristics are most likely to wrong higher up in the tree (there is least information available when no or few choices have been made).
- Stealing early or splitting high introduces diversity against early heuristic choices. Stealing late gives a close-to-sequential ordering.

## Confidence-Based Work Stealing

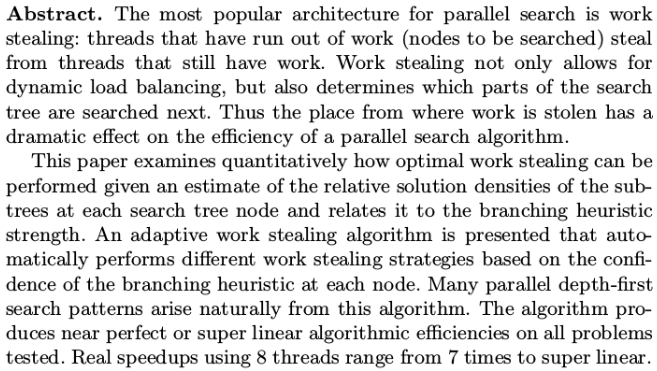


# Confidence-Based Work Stealing in Parallel Constraint Programming

Geoffrey Chu[1], Christian Schulte[2], and Peter J. Stuckey[1]

[1] National ICT Australia, Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au
[2] KTH – Royal Institute of Technology, Sweden
cschulte@kth.se

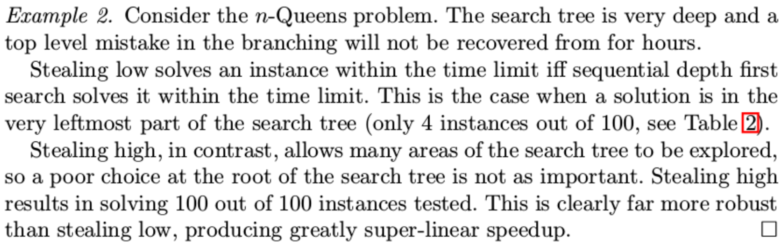
## Confidence-Based Work Stealing



**Abstract.** The most popular architecture for parallel search is work stealing: threads that have run out of work (nodes to be searched) steal from threads that still have work. Work stealing not only allows for dynamic load balancing, but also determines which parts of the search tree are searched next. Thus the place from where work is stolen has a dramatic effect on the efficiency of a parallel search algorithm.

This paper examines quantitatively how optimal work stealing can be performed given an estimate of the relative solution densities of the subtrees at each search tree node and relates it to the branching heuristic strength. An adaptive work stealing algorithm is presented that automatically performs different work stealing strategies based on the confidence of the branching heuristic at each node. Many parallel depth-first search patterns arise naturally from this algorithm. The algorithm produces near perfect or super linear algorithmic efficiencies on all problems tested. Real speedups using 8 threads range from 7 times to super linear.

## Confidence-Based Work Stealing

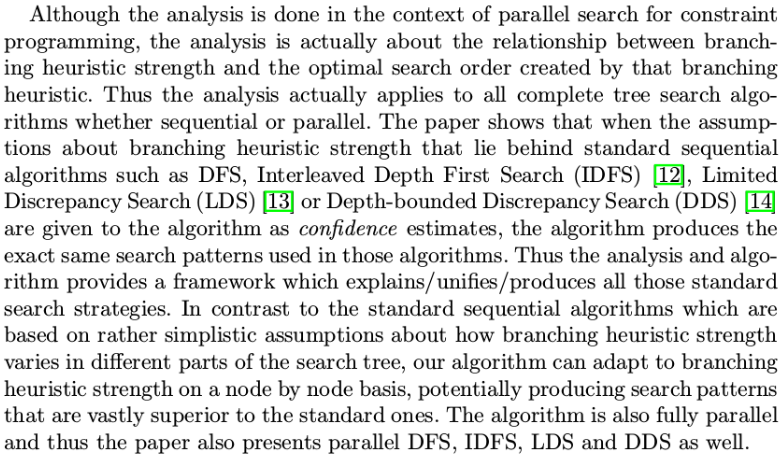


*Example 2.* Consider the $n$-Queens problem. The search tree is very deep and a top level mistake in the branching will not be recovered from for hours.

Stealing low solves an instance within the time limit iff sequential depth first search solves it within the time limit. This is the case when a solution is in the very leftmost part of the search tree (only 4 instances out of 100, see Table 2).

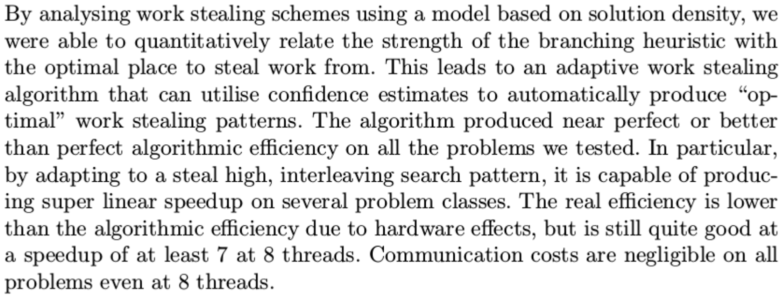
Stealing high, in contrast, allows many areas of the search tree to be explored, so a poor choice at the root of the search tree is not as important. Stealing high results in solving 100 out of 100 instances tested. This is clearly far more robust than stealing low, producing greatly super-linear speedup. □

## Confidence-Based Work Stealing



Although the analysis is done in the context of parallel search for constraint programming, the analysis is actually about the relationship between branching heuristic strength and the optimal search order created by that branching heuristic. Thus the analysis actually applies to all complete tree search algorithms whether sequential or parallel. The paper shows that when the assumptions about branching heuristic strength that lie behind standard sequential algorithms such as DFS, Interleaved Depth First Search (IDFS) [12], Limited Discrepancy Search (LDS) [13] or Depth-bounded Discrepancy Search (DDS) [14] are given to the algorithm as *confidence* estimates, the algorithm produces the exact same search patterns used in those algorithms. Thus the analysis and algorithm provides a framework which explains/unifies/produces all those standard search strategies. In contrast to the standard sequential algorithms which are based on rather simplistic assumptions about how branching heuristic strength varies in different parts of the search tree, our algorithm can adapt to branching heuristic strength on a node by node basis, potentially producing search patterns that are vastly superior to the standard ones. The algorithm is also fully parallel and thus the paper also presents parallel DFS, IDFS, LDS and DDS as well.

## Confidence-Based Work Stealing

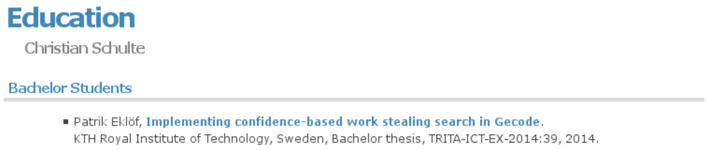


By analysing work stealing schemes using a model based on solution density, we were able to quantitatively relate the strength of the branching heuristic with the optimal place to steal work from. This leads to an adaptive work stealing algorithm that can utilise confidence estimates to automatically produce "optimal" work stealing patterns. The algorithm produced near perfect or better than perfect algorithmic efficiency on all the problems we tested. In particular, by adapting to a steal high, interleaving search pattern, it is capable of producing super linear speedup on several problem classes. The real efficiency is lower than the algorithmic efficiency due to hardware effects, but is still quite good at a speedup of at least 7 at 8 threads. Communication costs are negligible on all problems even at 8 threads.

## Using Confidence-Based Work Stealing

- I would like to be able to tell you that all you need to do to get excellent speedups from parallelism, and all of the benefits of discrepancy searches with none of the costs, is to annotate your model and your heuristics a little bit and then use Confidence-Based Work Stealing.
- But the implementations aren't quite at that stage yet.

## Using Confidence-Based Work Stealing



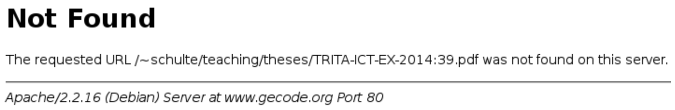
Education

Christian Schulte

Bachelor Students

- Patrik Eklöf, **Implementing confidence-based work stealing search in Gecode**.
  KTH Royal Institute of Technology, Sweden, Bachelor thesis, TRITA-ICT-EX-2014:39, 2014.

## Using Confidence-Based Work Stealing



**Not Found**

The requested URL /~schulte/teaching/theses/TRITA-ICT-EX-2014:39.pdf was not found on this server.

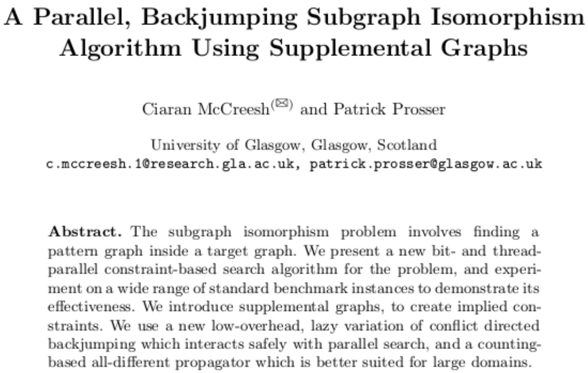
*Apache/2.2.16 (Debian) Server at www.gecode.org Port 80*

## Using Confidence-Based Work Stealing

## Early Diversity Work Stealing

- Confidence seems hard. What if we just steal as early as possible, and subject to that, as far left as possible?
- We'll get reproducible runtimes, can guarantee we'll never get a substantial slowdown, and adding more processors will never make things worse.
- Added bonus: we can parallelise conflict-directed backjumping this way too (although we no longer expect a linear speedup for unsatisfiable instances).

## Early Diversity Work Stealing



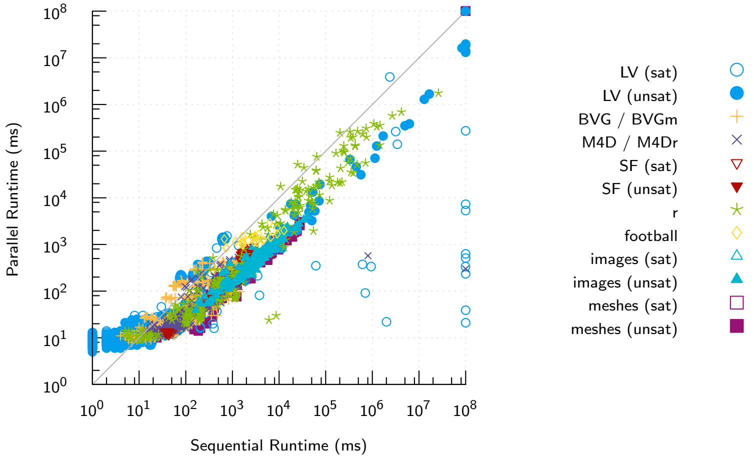
A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs

Ciaran McCreesh[✉] and Patrick Prosser

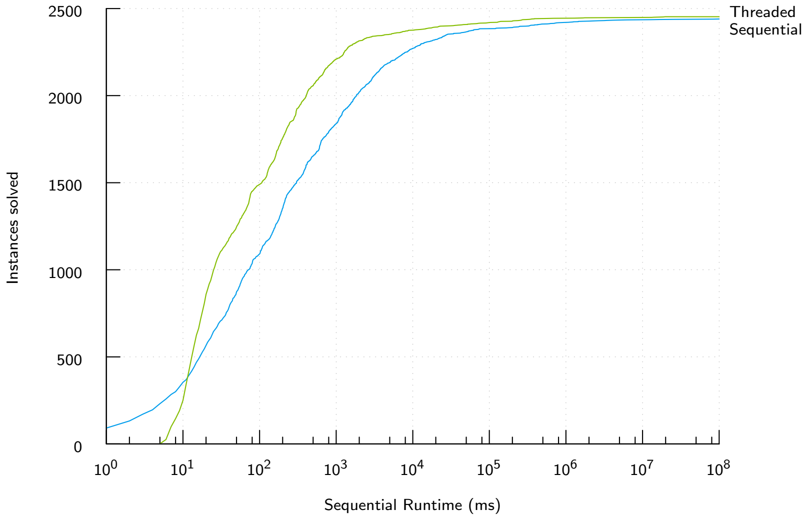
University of Glasgow, Glasgow, Scotland
c.mccreesh.1@research.gla.ac.uk, patrick.prosser@glasgow.ac.uk

**Abstract.** The subgraph isomorphism problem involves finding a pattern graph inside a target graph. We present a new bit- and thread-parallel constraint-based search algorithm for the problem, and experiment on a wide range of standard benchmark instances to demonstrate its effectiveness. We introduce supplemental graphs, to create implied constraints. We use a new low-overhead, lazy variation of conflict directed backjumping which interacts safely with parallel search, and a counting-based all-different propagator which is better suited for large domains.

# Early Diversity Work Stealing



# Early Diversity Work Stealing



## This is Not The Exam Question

What is *balance*, and why is it a problem if we try to parallelise a tree-search by creating *n* sub-trees for *n* processors? Suggest two potential remedies.

Why does Amdahl's law not apply to parallel tree-search? Why are *super-linear* speedups possible?

Suppose we are solving a decision problem which has a sequential part taking one second to run, and a parallelisable part which takes twenty seconds to run on one processor. What is the best possible runtime we might see when using ten processors to solve this problem with a parallel tree-search, if the instance is satisfiable? What if it is unsatisfiable?