Parallel Search, Ordering, Reproducibility, and Scalability

Blair Archibald Ruth Hoffmann **Ciaran McCreesh**
Patrick Prosser Phil Trinder

University of Glasgow

Motivation

- Everyone has at least a few cores.
- Multicore with a few tens of cores is cheap.
- Thousands of cores of distributed memory through cloud or HPC is affordable.
- GPUs are a whole other problem…

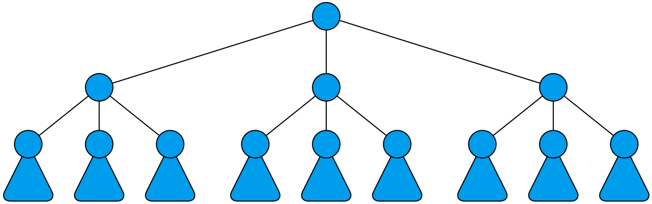
Demotivation

- Parallelising good combinatorial search algorithms is *hard*.
- There is no expectation of a linear speedup, and there is a real risk of introducing occasional (or frequent…) exponential slowdowns.
- We might also lose reproducibility: running the same search on the same machine twice could take vastly different amounts of time.

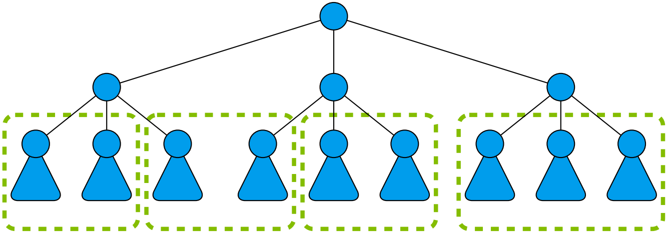
Remotivation

- Under certain circumstances, we can guarantee we will not introduce anomalies.
- We can also abstract out a lot of the work.

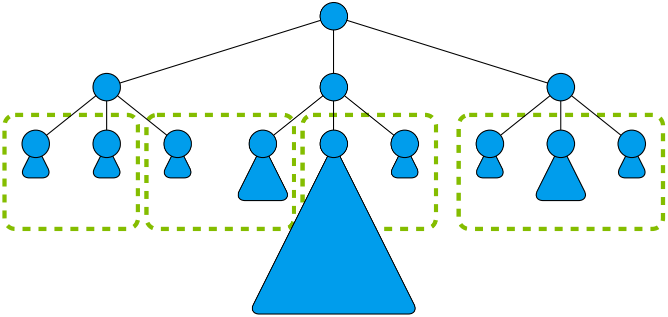
Parallel Tree Search



Parallel Tree Search



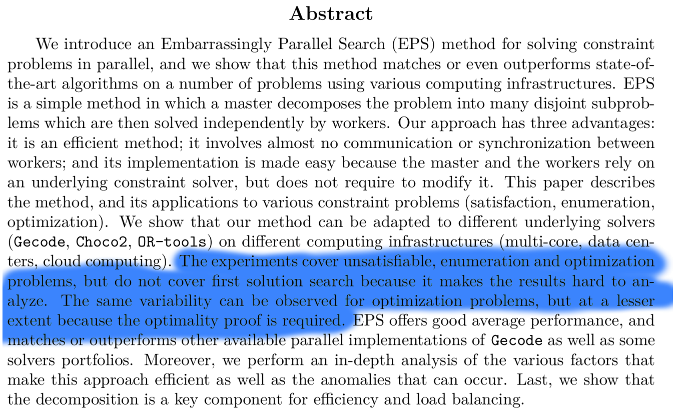
Work Balance is Tricky



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



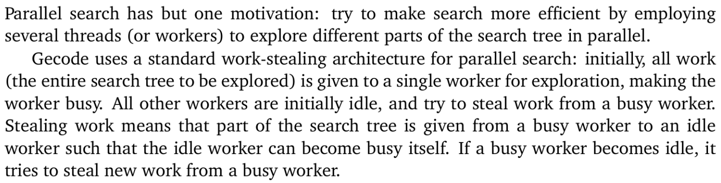
Abstract

We introduce an Embarrassingly Parallel Search (EPS) method for solving constraint problems in parallel, and we show that this method matches or even outperforms state-of-the-art algorithms on a number of problems using various computing infrastructures. EPS is a simple method in which a master decomposes the problem into many disjoint subproblems which are then solved independently by workers. Our approach has three advantages: it is an efficient method; it involves almost no communication or synchronization between workers; and its implementation is made easy because the master and the workers rely on an underlying constraint solver, but does not require to modify it. This paper describes the method, and its applications to various constraint problems (satisfaction, enumeration, optimization). We show that our method can be adapted to different underlying solvers (`Gecode`, `Choco2`, `OR-tools`) on different computing infrastructures (multi-core, data centers, cloud computing). The experiments cover unsatisfiable, enumeration and optimization problems, but do not cover first solution search because it makes the results hard to analyze. The same variability can be observed for optimization problems, but at a lesser extent because the optimality proof is required. EPS offers good average performance, and matches or outperforms other available parallel implementations of `Gecode` as well as some solvers portfolios. Moreover, we perform an in-depth analysis of the various factors that make this approach efficient as well as the anomalies that can occur. Last, we show that the decomposition is a key component for efficiency and load balancing.

Work Stealing

- Start a backtracking search, as normal.
- Have additional threads "steal" work from each other when they are idle, according to some selection policy.
  - Random victim selection is common.
  - Also have to decide which part of the victim's work we steal.
- Gecode has parallel search using randomised work-stealing, but it is not on by default.

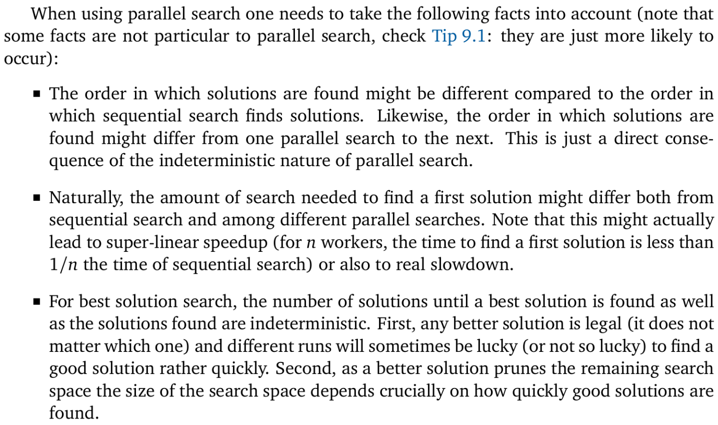
Gecode's Long List of Caveats



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.

Gecode's Long List of Caveats



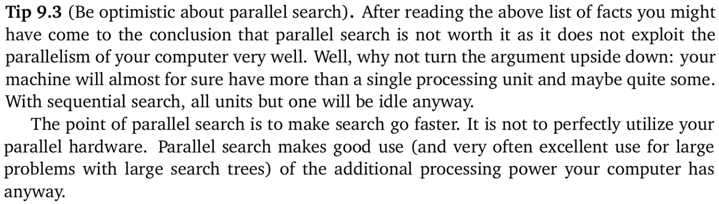
When using parallel search one needs to take the following facts into account (note that some facts are not particular to parallel search, check Tip 9.1: they are just more likely to occur):

- 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.

Gecode's Long List of Caveats

- 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.

Gecode's Long List of Caveats



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.

Gecode’s Long List of Caveats

```
$ mzn-gecode colOpt.mzn g80.dzn -a -s
%%  runtime:       6:32.621 (392621.621 ms)
%%  runtime:       6:31.311 (391311.168 ms)
%%  runtime:       6:31.314 (391314.705 ms)
%%  runtime:       6:30.360 (390360.443 ms)
%%  runtime:       6:31.217 (391217.210 ms)
%%  runtime:       6:33.723 (393723.672 ms)
%%  runtime:       6:31.279 (391279.313 ms)
%%  runtime:       6:30.765 (390765.244 ms)
%%  runtime:       6:31.057 (391057.970 ms)
%%  runtime:       6:30.460 (390460.464 ms)

$ mzn-gecode colOpt.mzn g80.dzn -a -s -p32
%%  runtime:       1:31.237 (91237.601 ms)
%%  runtime:       22.783 (22783.639 ms)
%%  runtime:       1:33.024 (93024.102 ms)
%%  runtime:       23.844 (23844.334 ms)
%%  runtime:       1:32.932 (92932.198 ms)
%%  runtime:       24.415 (24415.674 ms)
%%  runtime:       26.329 (26329.784 ms)
%%  runtime:       1:31.005 (91005.068 ms)
%%  runtime:       1:17.512 (77512.379 ms)
%%  runtime:       1:33.766 (93766.558 ms)
```

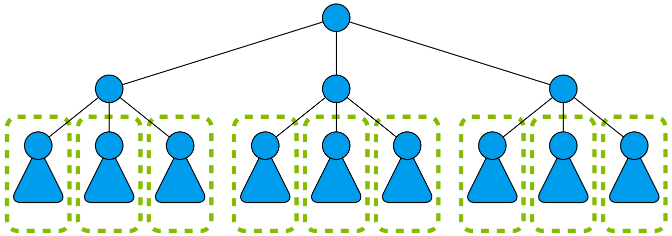
Gecode's Long List of Caveats

```
$ mzn-gecode colDec.mzn g80.dzn -s -Dk=12
=====UNSATISFIABLE=====
%%  runtime:       4:10.639 (250639.167 ms)
%%  solvetime:     4:10.625 (250625.001 ms)
%%  solutions:     0
%%  variables:     80
%%  propagators:   1533
%%  propagations:  909705614
%%  nodes:         8390437
%%  failures:      4275231
%%  restarts:      0
%%  peak depth:    32

$ mzn-gecode colDec.mzn g80.dzn -s -Dk=12 -p32
%%  runtime:       20.642 (20642.164 ms)
%%  runtime:       21.168 (21168.189 ms)
%%  runtime:       21.896 (21896.198 ms)
%%  runtime:       20.773 (20773.895 ms)
%%  runtime:       21.291 (21291.617 ms)
%%  runtime:       21.229 (21229.889 ms)
%%  runtime:       21.994 (21994.265 ms)
%%  runtime:       21.929 (21929.667 ms)
%%  runtime:       20.992 (20992.017 ms)
%%  runtime:       21.269 (21269.629 ms)
```

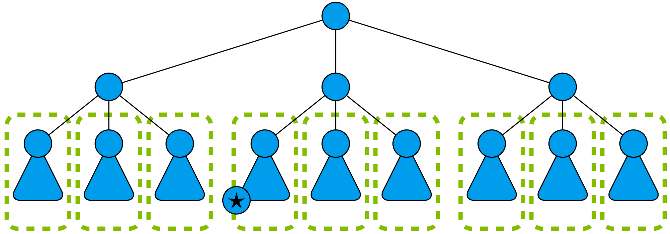
Gecode's Long List of Caveats

```
$ mzn-gecode colDec.mzn g80.dzn -s -Dk=13
v = array1d(1..80 ,[4, 7, 2, 8, 1, 11, 12, 6, 13, 10, 8, 4, 3, 13, 4, 4, 3, 4, 13, 5, 1, 12,
----------
%%  runtime:       1:58.791 (118791.370 ms)
%%  solvetime:     1:58.777 (118777.305 ms)
%%  solutions:     1
%%  variables:     80
%%  propagators:   1534
%%  propagations:  518688552
%%  nodes:         5165932
%%  failures:      2604376
%%  restarts:      0
%%  peak depth:    41

$ mzn-gecode colDec.mzn g80.dzn -s -Dk=13 -p32
%%  runtime:       28.291 (28291.176 ms)
%%  runtime:       38.170 (38170.591 ms)
%%  runtime:       24.390 (24390.286 ms)
%%  runtime:       10.547 (10547.847 ms)
%%  runtime:       39.040 (39040.280 ms)
%%  runtime:       28.988 (28988.032 ms)
%%  runtime:       1:08.091 (68091.332 ms)
%%  runtime:       1:13.061 (73061.990 ms)
%%  runtime:       11.618 (11618.722 ms)
%%  runtime:       1:05.073 (65073.290 ms)
```

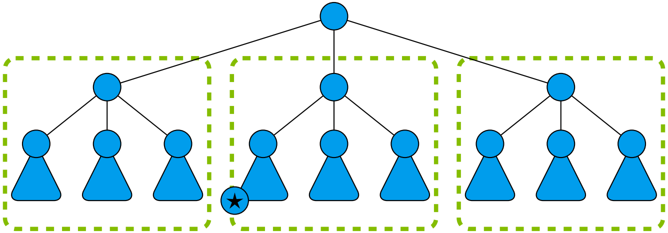
Value-Ordering Heuristics Matter



Value-Ordering Heuristics Matter



Value-Ordering Heuristics Matter



Value-Ordering Heuristics Matter

- Value-ordering heuristics are right most of the time.
- They are most likely to be wrong early on in the search.
- Mistakes made early in the search tree are the most costly.

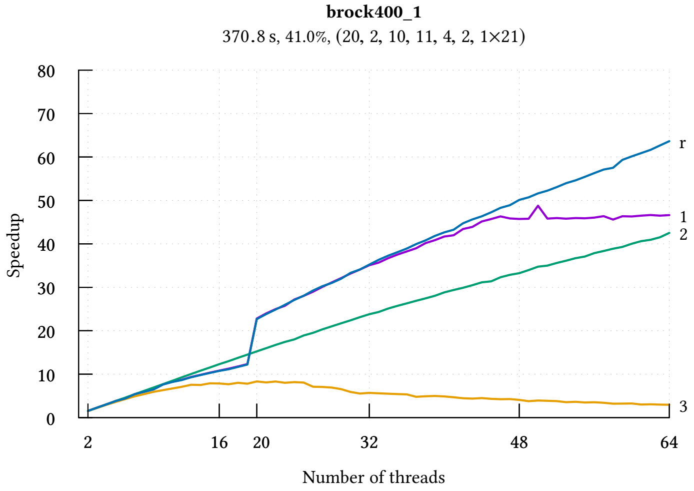
Performance Guarantees and “Anomalies”

- There is literature from the 1980s and early 1990s on “anomalies” in a particular kind of branch and bound algorithm.
- In particular, we know sufficient conditions to guarantee:
    - Parallel cannot be worse than sequential.
    - Increasing the number of processors cannot increase runtimes.
    - Running the same instance on the same hardware twice will take the same amount of time.
- However, value-ordering heuristics were not widely understood in those days.

Descending Resplitting

- Split the search tree at depth one, and place the subtrees in order in a queue.
- Have each worker pull from this queue.
- When the queue is empty and a worker needs work, suspend all workers, and requeue remaining work (branches to the right) at depth two of the search tree.
- Then again at depth three, and so on.
- This gives us all the performance guarantees, and could be particularly good if value-ordering heuristics are weakest at the top of search or if mistakes are most costly at the top of search.
- It is, however, moderately unpleasant to implement…

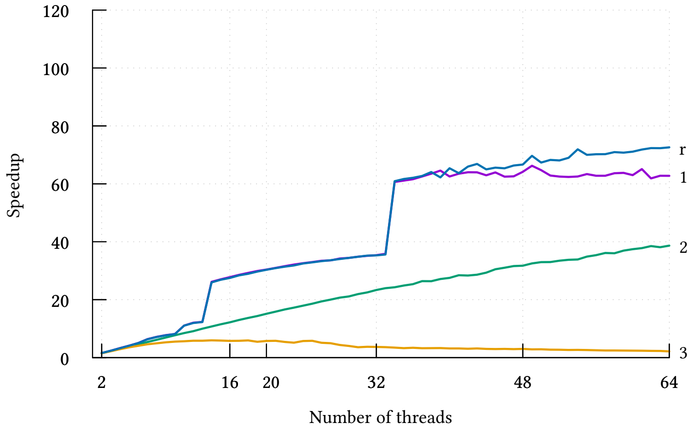
In a Maximum Clique Algorithm



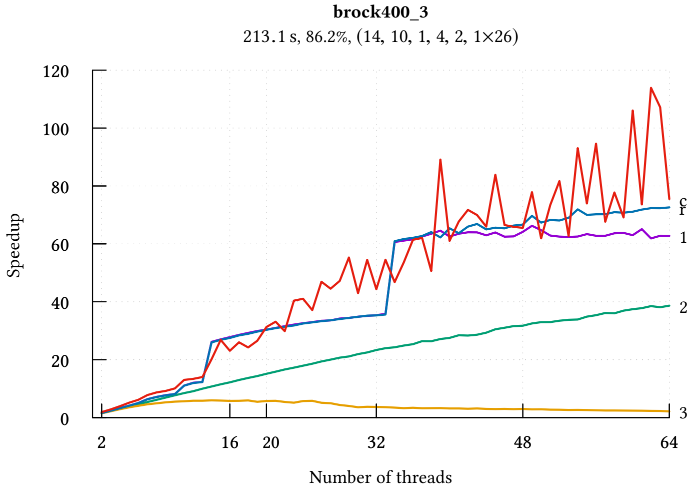
In a Maximum Clique Algorithm

brock400_3

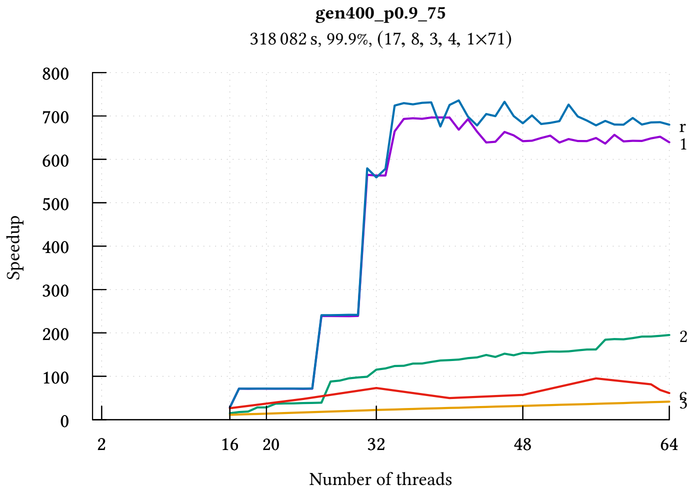
213.1 s, 86.2%, (14, 10, 1, 4, 2, 1×26)



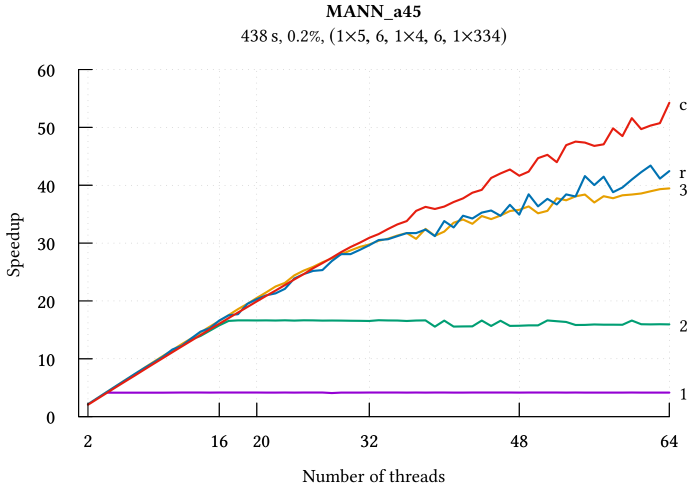
In a Maximum Clique Algorithm



In a Maximum Clique Algorithm

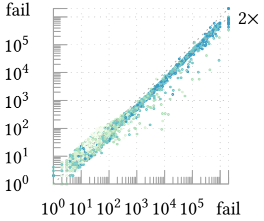


In a Maximum Clique Algorithm

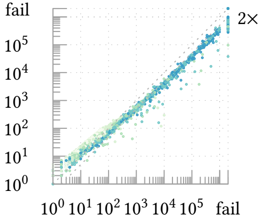


The Theoretical Guarantees Hold in Practice!

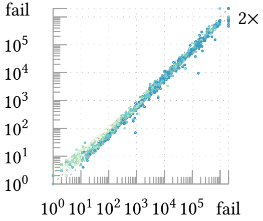
1 vs 2



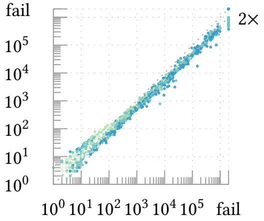
2 vs 4



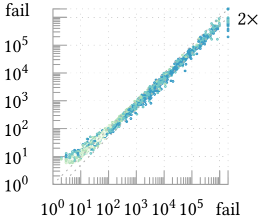
4 vs 8



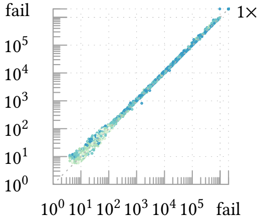
8 vs 16



16 vs 32



32 vs 32



Abstracting to Skeletons

- This work-stealing turns a twenty line recursive search algorithm into three hundred lines, or nearly a thousand lines to also support distributed memory parallelism.
- Algorithmic skeletons provide an alternative:
  - Write your code once, to fit the skeleton.
  - Compile many times, for sequential, shared memory, cloud, or HPC parallelism.
  - Performance close to hand-tuned (even sequentially).
  - Can swap in different search schedulers.

Modern Algorithmic Features that Cause Problems

- Non-monotonic propagators and misleading bound functions break the guarantees.
- No guarantees with adaptive variable-ordering heuristics like wdeg, although in practice anomalies seem much rarer.
- Learning ruins everything.

A Quick Preview of a Possible Alternative

- Introduce just a tiny bit of randomness into value-ordering.
- Use aggressive restarts and decision nogood recording.
- All gather nogoods on restarts.
- Advantages:
  - Extremely non-intrusive to implement.
  - Scales to many hundreds of cores.
- Problems:
  - Need to implement 2WL for nogood recording.
  - It changes (improves?) the sequential search algorithm.
  - No performance guarantees, although if we could come up with some notion of “statistical” guarantees they would probably hold.

http://www.dcs.gla.ac.uk/~ciaran
ciaran.mccreesh@glasgow.ac.uk

University of Glasgow