Parallel Constraint Programming

Ciaran McCreesh and Patrick Prosser

University of Glasgow

Motivation

- This laptop: 4 cores.
- Modern workstation or server: 32 cores per machine.

Qty	Description	Unit Price	Ext. Price
1	Server Node	£22,093.95	£22,093.95
	SYS6018TR-TF SuperSuper - 1U Twin Board, Single PSU		
	(2) SuperMicro X10DRT-LIBF Motherboard (8 x DIMM Slot) 2x RJ45 Gigabit Ethernet LAN ports 1x RJ45 Dedicated IPMI LAN port 1x external QSFP Infiniband connector		
	(4) Intel® Xeon® Processor E5-2697A v4 (40M Cache, 2.60 GHz) FC-LGA14A, Tray		
	(16) 64GB DDR4 ECC REG DIMM		
	(2) Samsung PM863A 240GB SSD		
	(2) WD 6TB RED 64MB 3.5" SATA HDD		

SubTotal		£22,093.95
VAT		£4,418.79
Carriage		£0.00
Total		**£26,512.74**

Parallelism and Concurrency

- Concurrent: lots of stuff happening at once (GUIs, operating systems, networking).
- Parallel: our hardware can do more than one thing at once (multi-core, multi-machine, vector processing, GPUs).

Goals

1. Make slow things run faster.
   - If "today's list of parcels to be delivered" isn't available until 5am, and producing "today's delivery schedule" takes twelve hours, we're in trouble. If it takes one hour, we're OK.
   - If it takes one second, we don't care if we can reduce it to one tenth of a second. (Or maybe we do. What if we're producing results interactively?)

2. Deal with bigger or harder problems in "the amount of time we have".
   - We have a fixed amount of time (say, a week) to produce exam timetables. If the University offers more courses, or more flexibility in course choices, we need to solve a larger and harder problem in the same amount of time.

Unfortunately. . .

- Parallel constraint programming is hard.
- Most of this lecture is about techniques that don't usually work very well in practice. The goal is to understand *why* these techniques fail.
- We'll eventually see some techniques that usually work fairly well, most of the time, if you don't investigate too closely.

Attempt One: Parallel Optimisation?

- An optimisation problem is just a sequence of decision problems.
- What if we run each decision problem on a separate processor?

Attempt One: Parallel Optimisation?

```
% graph colouring optimisation problem

int: n;                        % number of vertices
array[1..n, 1..n] of 0..1: A;  % adjacency

array[1..n] of var 1..n: v;    % v[i] = j means vertex i has colour j

% adjacent vertices must have different colours
constraint forall(i, j in 1..n where i < j /\ A[i, j] = 1)(v[i] != v[j]);

% objective is to minimise chi
var 1..n: chi;
constraint chi = max(v);

% make a copy of v, sorted to have highest degree first
array[1..n] of var 1..n: sorted_v = sort_by(v, [-sum(row(A, w)) | w in 1..n]);

% smallest domain first, tie-breaking on highest degree, lowest colour first
solve ::int_search(sorted_v, first_fail, indomain, complete) minimize chi;

% symmetry: colours are equivalent. make colours appear in order in sorted_v.
include "value_precede_chain.mzn";
constraint value_precede_chain([i | i in 1..n], sorted_v);
```

Attempt One: Parallel Optimisation?

```
$ mzn-gecode colOpt.mzn g80.dzn -a -s
v = array1d(1..80 ,[10, 12, 2, 13, 1, 4, 10, 2, 11, 2, 7, 7, 3, 14, 6, 13, 3, 4, 11, 5,
chi = 16;
----------
v = array1d(1..80 ,[10, 12, 2, 13, 1, 4, 10, 2, 11, 2, 7, 14, 3, 15, 6, 13, 3, 4, 11, 5,
chi = 15;
----------
v = array1d(1..80 ,[12, 2, 2, 14, 1, 4, 10, 12, 10, 7, 7, 7, 3, 5, 6, 12, 3, 4, 12, 5, 1
chi = 14;
----------
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,
chi = 13;
----------
==========
%%  runtime:       6:32.621 (392621.621 ms)
%%  solvetime:     6:32.595 (392595.935 ms)
%%  solutions:     4
%%  variables:     81
%%  propagators:   1601
%%  propagations:  1437165314
%%  nodes:         13502114
%%  failures:      6851645
%%  restarts:      0
%%  peak depth:    72
```

Attempt One: Parallel Optimisation?

```
% graph colouring decision problem

int: n;                          % number of vertices
array[1..n, 1..n] of 0..1: A;    % adjacency
int: k;                          % number of colours allowed

array[1..n] of var 1..k: v;      % v[i] = j means vertex i has colour j

% adjacent vertices must have different colours
constraint forall(i, j in 1..n where i < j /\ A[i, j] = 1)(v[i] != v[j]);

% make a copy of v, sorted to have highest degree first
array[1..n] of var 1..n: sorted_v = sort_by(v, [-sum(row(A, w)) | w in 1..n]);

% smallest domain first, tie-breaking on highest degree, lowest colour first
solve ::int_search(sorted_v, first_fail, indomain, complete) satisfy;

% symmetry: colours are equivalent. make colours appear in order in sorted_v.
include "value_precede_chain.mzn";
constraint value_precede_chain([i | i in 1..k], sorted_v);
```

Attempt One: Parallel Optimisation?

```
$ mzn-gecode colDec.mzn g80.dzn -s -Dk=2
=====UNSATISFIABLE=====
%%  runtime:          0.013 (13.011 ms)
%%  solvetime:        0.000 (0.082 ms)
%%  solutions:        0
%%  variables:        80
%%  propagators:      0
%%  propagations:     0
%%  nodes:            0
%%  failures:         1
%%  restarts:         0
%%  peak depth:       0
```

Attempt One: Parallel Optimisation?

```
$ mzn-gecode colDec.mzn g80.dzn -s -Dk=3
=====UNSATISFIABLE=====
%%  runtime:       0.023 (23.387 ms)
%%  solvetime:     0.000 (0.192 ms)
%%  solutions:     0
%%  variables:     80
%%  propagators:   0
%%  propagations:  70
%%  nodes:         0
%%  failures:      1
%%  restarts:      0
%%  peak depth:    0
```

Attempt One: Parallel Optimisation?

```
$ mzn-gecode colDec.mzn g80.dzn -s -Dk=4
=====UNSATISFIABLE=====
%%  runtime:        0.014 (14.702 ms)
%%  solvetime:      0.000 (0.527 ms)
%%  solutions:      0
%%  variables:      80
%%  propagators:    1525
%%  propagations:   186
%%  nodes:          3
%%  failures:       2
%%  restarts:       0
%%  peak depth:     1
```

Attempt One: Parallel Optimisation?

```
$ mzn-gecode colDec.mzn g80.dzn -s -Dk=80
v = array1d(1..80 ,[10, 12, 2, 13, 1, 4, 10, 2, 11, 2, 7, 7, 3, 14, 6, 13, 3, 4, 11, 5,
----------
%%  runtime:        0.025 (25.464 ms)
%%  solvetime:      0.009 (9.443 ms)
%%  solutions:      1
%%  variables:      80
%%  propagators:    1600
%%  propagations:   6867
%%  nodes:          73
%%  failures:       0
%%  restarts:       0
%%  peak depth:     72
```

Attempt One: Parallel Optimisation?

```
$ 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
```

Attempt One: Parallel Optimisation?

```
$ 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,
----------
%%  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
```

Attempt One: Parallel Optimisation?
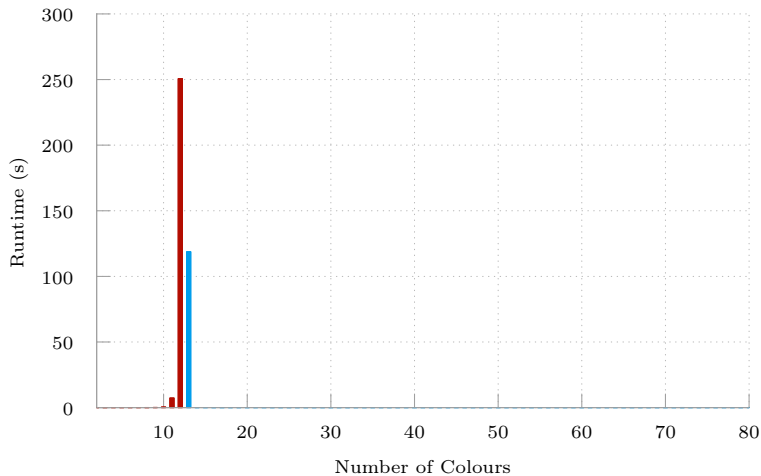
Parallel In Theory

- *Speedup* is sequential runtime divided by parallel runtime.
  - Ideally, over a good sequential algorithm, not a parallel algorithm run with one thread. This is sometimes called *absolute* speedup.
  - This may not be practical if using special hardware.
- A *linear* speedup is a speedup of n using n processors.
  - This is not a realistic expectation on modern hardware. Of particular concern for CP is that more cores does not mean more memory bandwidth.

Parallel In Theory

- *Balance* is whether every compute unit is kept busy doing useful work.
- A *regular* problem is one which can easily be split into equally sized units of work. Irregular problems are hard to balance.
- Often only a small number of the decision problems are "really hard", so we get poor balance.

Parallel In Theory

- Most parallel algorithms contain a "sequential" part which cannot be parallelised, and a "parallel" part.
- *Amdahl's law* says that if the sequential portion is fixed and we divide the parallel portion perfectly among n processors, and if k is the fraction of the work we cannot parallelise, then

$$best\ speedup = \frac{1}{k + \frac{1}{n}(1 - k)}$$

- For CP algorithms, things get much more complicated, so it is important to understand where the formula comes from (using primary school maths), rather than memorising it.
- *Gustafson's law* deals with using more processors to tackle larger problems.

Ciaran McCreesh and Patrick Prosser

Parallel Constraint Programming

Parallel In Theory

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

Parallel Consistency?

- Partition the variables between processors.
- Run AC3 independently on each processor, but when deleting a value, also send a message to other processors telling them to re-add the relevant variables to their stack.
- But maybe only a few variables are involved, and we spend all our time bouncing around between a small number of processors. . .
- And what about slow-running global constraints?

Parallel Consistency?

A distributed arc-consistency algorithm

T. Nguyen, Y. Deville*

*Université catholique de Louvain, Département d'Ingénierie Informatique,
Place Ste Barbe 2, B-1348 Louvain-la-Neuve, Belgium*

A Preliminary Review of Literature on Parallel Constraint Solving

Ian P. Gent, Chris Jefferson, Ian Miguel, Neil C.A. Moore, Peter Nightingale,
Patrick Prosser, Chris Unsworth

Computing Science,
Glasgow and St. Andrews Universities, Scotland
pat@dcs.gla.ac.uk

Experimental results show a linear speedup.

The DisAC-4 algorithm is not intended to be massively parallel.

In 1995 Nguyen and Deville presented a distributed AC-4 algorithm DisAC-4 [24]. A journal version of this work was published in 1998 [25]. The algorithm is based on message passing. The variables are partitioned among the workers, and each worker essentially maintains the AC4 data structures for its set of variables. When a worker deduces a domain deletion, this is broadcast to all other workers. Each worker maintains a list of domain deletions to process (some generated locally and others received from another worker). The worker reaches a fixpoint itself before broadcasting any domain deletions, and waiting for new messages from other workers. The whole system reaches a fixpoint when every worker has processed every domain deletion.

It may be a difficult problem to partition the variables such that the work is evenly distributed. The experimental results are mixed, with some experiments showing close to linear speedup, while others show only 1.5 times speedup with 8 processors.

Parallel Consistency?

Experimental results show a linear speedup.

It may be a difficult problem to partition the variables such that the work is evenly distributed. The experimental results are mixed, with some experiments showing close to linear speedup, while others show only 1.5 times speedup with 8 processors.

Parallel Consistency?

The DisAC-4 algorithm is not intended to be massively parallel.

Parallel Consistency?

On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks

Simon Kasif

Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, USA

Our analysis suggests that a parallel solution is unlikely to improve the known sequential solutions by much. Specifically, we prove that the problem solved by discrete relaxation (arc consistency) is log-space complete for P (the class of polynomial-time deterministic sequential algorithms). Intuitively, this implies that discrete relaxation is inherently sequential and it is unlikely that we can solve the polynomial-time version of the consistent labeling problem in logarithmic time by using only a polynomial number of processors. Some practical implications of our result are discussed.

This negative worst-case result needs to be quantified. Essentially, it suggests that the application of massive parallelism will not change significantly the worst-case complexity of discrete relaxation (unless one has an exponential number of processors). However, this result does not preclude research in the direction of applying parallelism in a more controlled fashion. For instance, we can easily obtain speedups when the constraint graph is very dense (the number of edges is large).

Parallel Consistency?

Our analysis suggests that a parallel solution is unlikely to improve the known sequential solutions by much. Specifically, we prove that the problem solved by discrete relaxation (arc consistency) is log-space complete for P (the class of polynomial-time deterministic sequential algorithms). Intuitively, this implies that discrete relaxation is inherently sequential and it is unlikely that we can solve the polynomial-time version of the consistent labeling problem in logarithmic time by using only a polynomial number of processors. Some practical implications of our result are discussed.

Parallel Consistency?

This negative worst-case result needs to be quantified. Essentially, it suggests that the application of massive parallelism will not change significantly the worst-case complexity of discrete relaxation (unless one has an exponential number of processors). However, this result does not preclude research in the direction of applying parallelism in a more controlled fashion. For instance, we can easily obtain speedups when the constraint graph is very dense (the number of edges is large).

Parallel Consistency?

a polynomial number of processors.

Bitsets and Bit Parallelism

- Domains are sometimes small and compact.
- If domains have no more than 64 values, we can store them in (long unsigned) integers. We have one bit per value. 0 means "not in the set" and 1 means "in the set".
- We can use arrays of integers for larger domains. (And we can go up to 512 bit integers on some Intel CPUs.)

Bitsets and Bit Parallelism

- This is hardware-friendly: the entire model might fit in cache.
- Setting a domain to take exactly one value:

$$d \leftarrow 1 \ll v$$

- Testing whether or not a value is present in a domain:

$$d \ \& \ (1 \ll v) \neq 0$$

- Turning a single bit off:

$$d \leftarrow d \ \& \ \tilde{}(1 \ll v)$$

- There are dedicated hardware instructions for all of these in recent CPUs. We can also count the number of set bits (how many values are left in our domain?), and find the first set bit (pick a value from the domain) in hardware.

Bitsets and Bit Parallelism

- Some constraints are similarly bitset friendly.
- Extensional constraints (a list of all "allowed pairs") can be represented as a "compatibility" bitset for each value in each variable's domain. Now forward checking is just a bitwise "and" operation.
  - Uses a lot of memory, but if our model is reasonably small and dense that's fine.
- Less than, greater than, and certain arithmetic constraints work nicely with bitsets.
  - Fun exercise: figure this out.
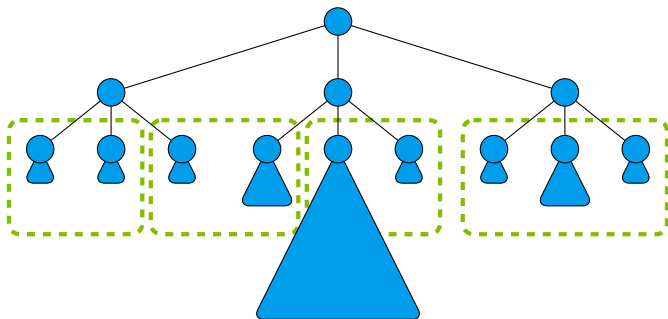- Some constraints are probably not bitset friendly.

Fixed Parallel Tree Search
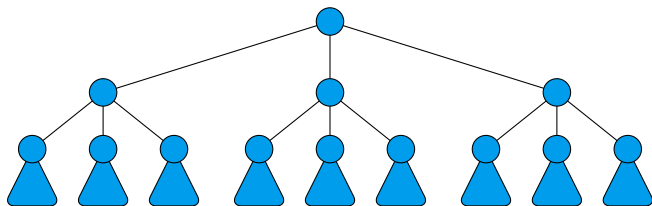
Fixed Parallel Tree Search
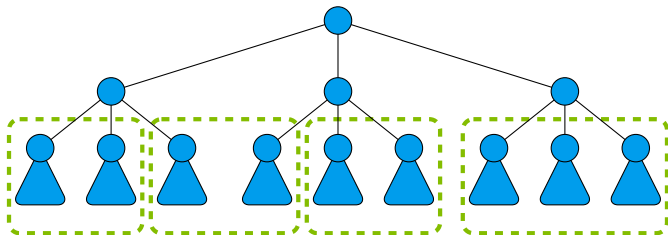
Fixed Parallel Tree Search

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 in Constraint Programming

Arnaud Malapert   ARNAUD.MALAPERT@UNICE.FR
Jean-Charles Régin   JEAN-CHARLES.REGIN@UNICE.FR
Mohamed Rezgui   REZGUI@I3S.UNICE.FR
Université Côte d'Azur, CNRS, I3S, France

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.

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.

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.

Random Work Stealing

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

Random 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.

Random 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.

Random 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.

Ciaran McCreesh and Patrick Prosser

Random 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 (and very often excellent use for large problems with large search trees) of the additional processing power your computer has anyway.

Random Work Stealing

```
$ 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)
```

Random Work Stealing

```
$ 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)
```

Random Work Stealing

```
$ 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,
----------
%%  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)
```

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*.
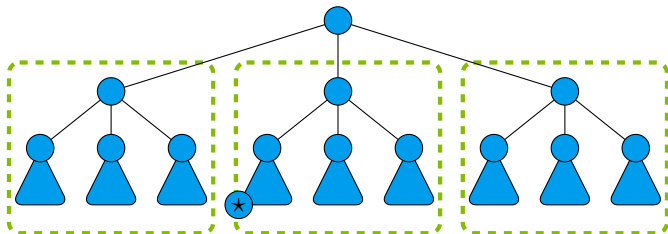
Work Splitting Affects Search
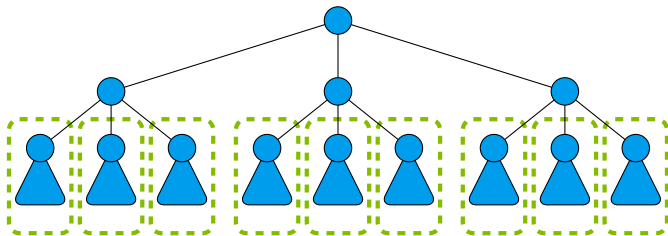
Work Splitting Affects Search
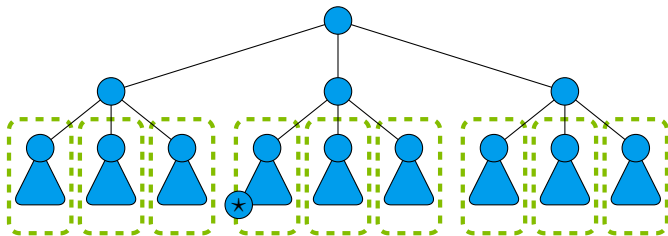
Work Splitting Affects Search

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.

Parallel Discrepancy Search

UNIVERSITÉ LAVAL

Minimisation des perturbations et parallélisation pour la planification et l'ordonnancement
Thèse
Thierry Moisan
Doctorat en informatique
Philosophiæ doctor (Ph.D.)
Québec, Canada
© Thierry Moisan, 2016

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 sub-trees 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

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

- Unfortunately, you can't. . .

Parallel Portfolios?

- We might have several models, several variable- and value-ordering heuristics, tiebreaking, etc.
- Just run them all, and pick whichever finishes first?
- We can share incumbents and bounds between models.

Parallel Portfolios?

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS   FRAMES NO FRAMES   ALL CLASSES
SUMMARY: NESTED | FIELD | CONSTR | METHOD   DETAIL: FIELD | CONSTR | METHOD

org.chocosolver.solver

Class ParallelPortfolio

java.lang.Object
  org.chocosolver.solver.ParallelPortfolio

```
public class ParallelPortfolio
extends Object
```

A Portfolio helper.

The ParallelPortfolio resolution of a problem is made of four steps:

1. adding models to be run in parallel,
2. running resolution in parallel,
3. getting the model which finds a solution (or the best one), if any.

Parallel Portfolios?

Since there is no condition on the similarity of the models, once the resolution ends, the model which finds the (best) solution is internally stored.

Example of use.

```
ParallelPortfolio pares = new ParallelPortfolio();
int n = 4; // number of models to use
for (int i = 0; i < n; i++) {
    pares.addModel(modeller());
}
pares.solve();
IOutputFactory.printSolutions(pares.getBestModel());
```

Parallel Restarts and Random Value-Ordering

- Use random or biased-random search with restarts and nogood recording.
- Give each thread its own random seed.
- Share nogoods using a barrier on restarts.

Parallel Restarts and Random Value-Ordering
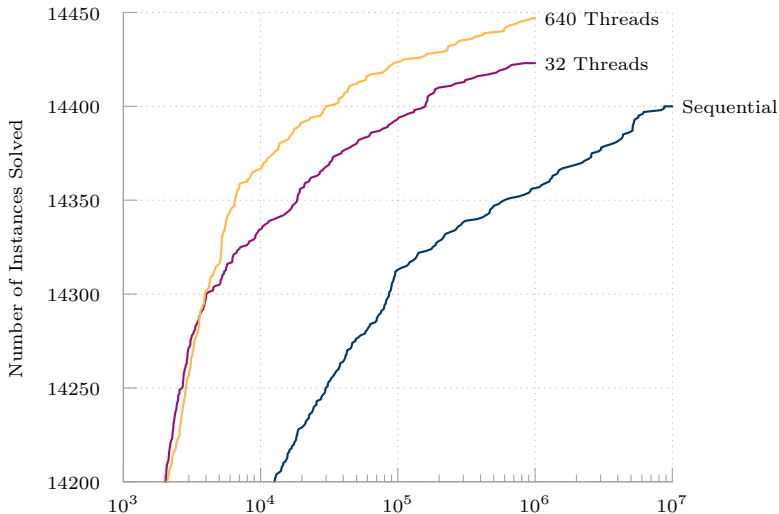
Parallel Restarts and Random Value-Ordering

- Coming soon to Choco!

This is Not The Exam Question

A constraint model takes 10 seconds to solve using one processor. Suppose 80% of that time is spent doing propagation. What is the best possible speedup that could be obtained if 4 processors are used to do parallel propagation, and the rest of the program remains unchanged? What about if we had an unlimited number of processors?

What about if we used the four processors for a portfolio of different solvers?

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.

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?

Design a parallel constraint programming approach that always gives linear speedups.