# Parallel Constraint Programming

(and why it is hard...)

**Ciaran McCreesh** and Patrick Prosser

University of Glasgow

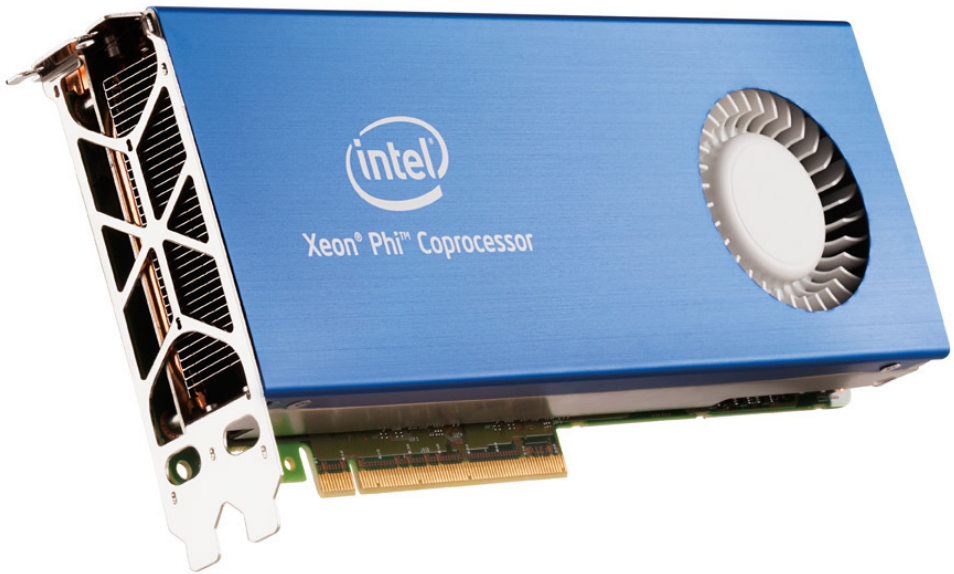
# This Week's Lectures

- Search and Discrepancies
- Parallel Constraint Programming
  - Why?
  - Some failed attempts
  - A little bit of theory and some very simple maths
  - Some partial successes
- Parallel Search

# 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).

# Why Care?




- Make your programs run 100 times faster overnight! All you need is this simple £2000 card. Doctors (of Philosophy) are astonished!

# 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.

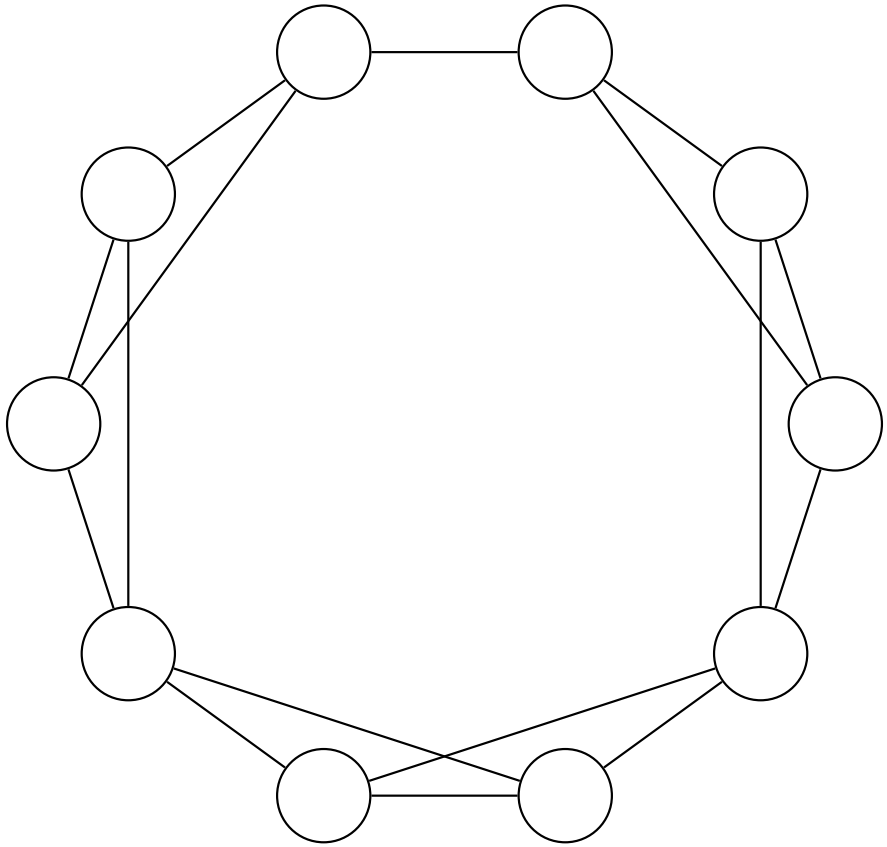
# Unfortunately...

- 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.
- Tomorrow we'll see some techniques that usually work fairly well, most of the time, if you don't investigate too closely.

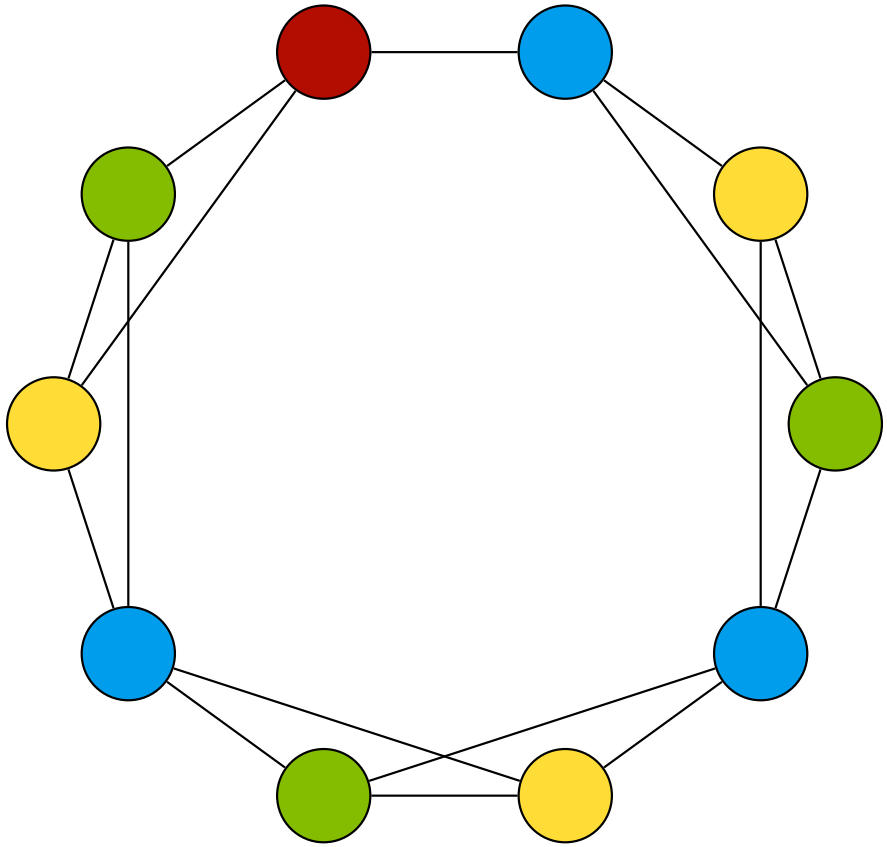
# Parallel Optimisation via Decision Problems?

- An optimisation problem can be solved as a sequence of decision problems.
- What happens if we solve each decision problem in parallel?

# Parallel Optimisation via Decision Problems?



# Parallel Optimisation via Decision Problems?

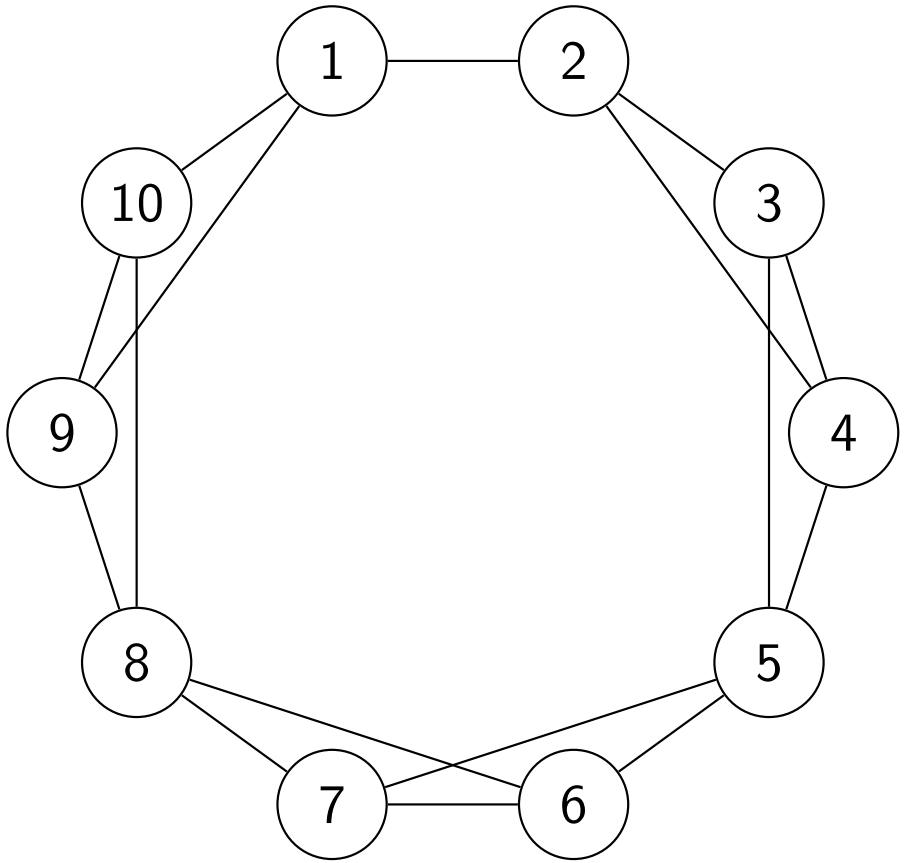


# Parallel Optimisation via Decision Problems?

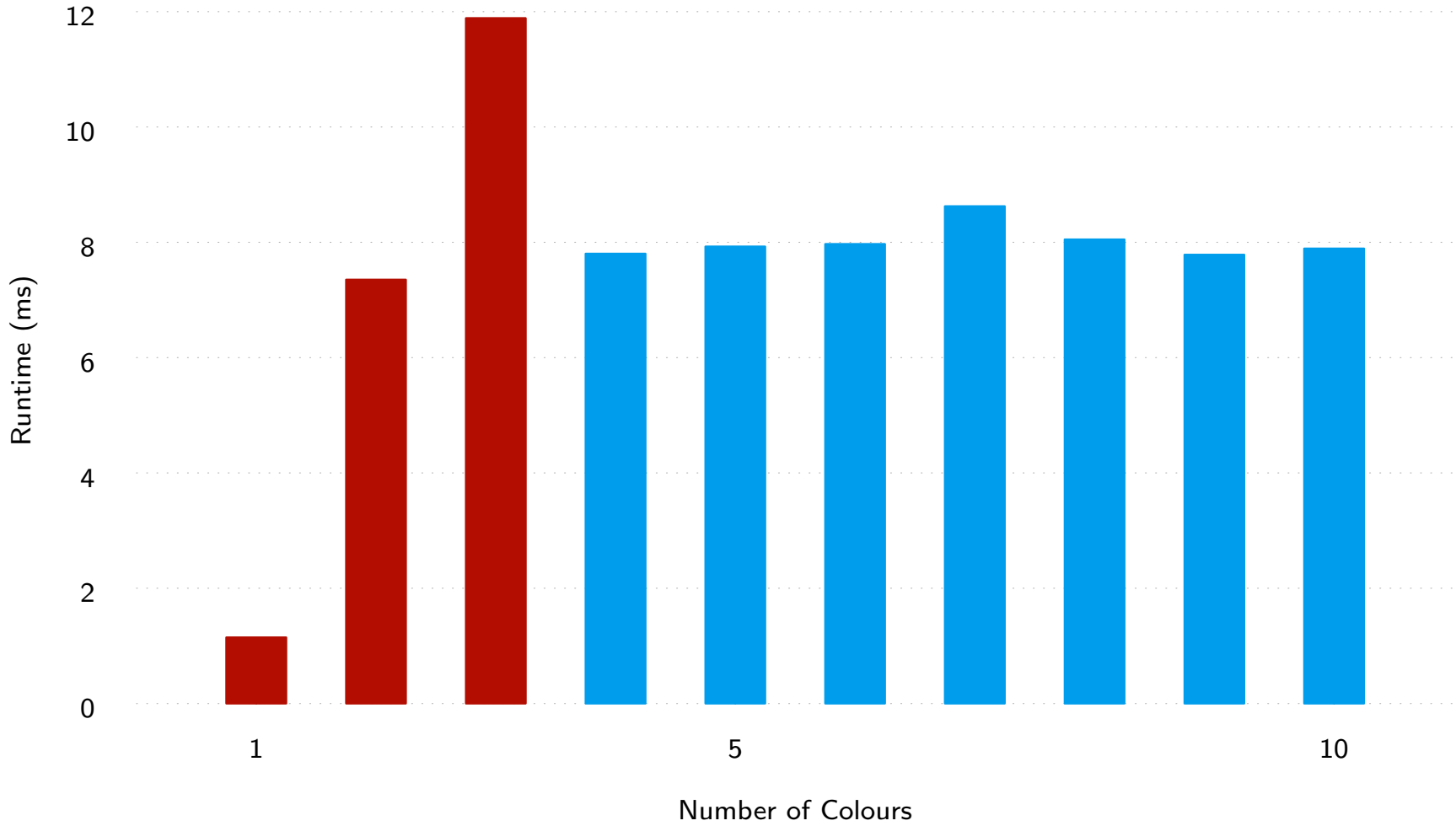
```
Solver solver = new Solver("colouring");

try (Scanner sc = new Scanner(new File(args[0]))) {
    int nVertices = sc.nextInt();
    int nColours  = Integer.parseInt(args[1]);

    IntVar[] v = enumeratedArray(
            "vertex", nVertices, 0, nColours - 1, solver);

    while (sc.hasNext()) {
        int from = sc.nextInt();
        int to = sc.nextInt();
        solver.post(arithm(v[from - 1], "!=", v[to - 1]));
    }
}

System.out.println(solver.findSolution());
System.out.println(solver.getMeasures().getTimeCount());
```

# Parallel Optimisation via Decision Problems?

```
10
1  2
2  3
2  4
3  4
3  5
4  5
5  6
5  7
6  7
6  8
7  8
8  9
8  10
9  1
9  10
10  1
```

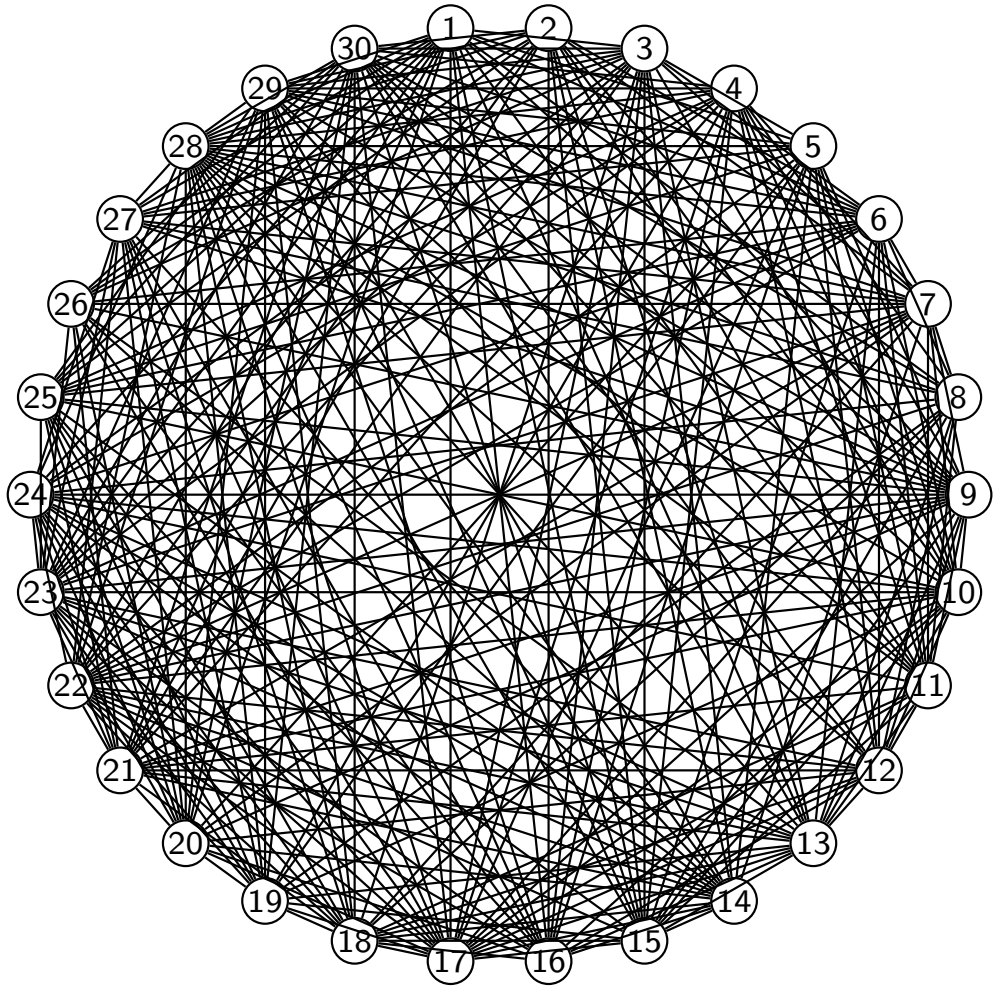


# Parallel Optimisation via Decision Problems?

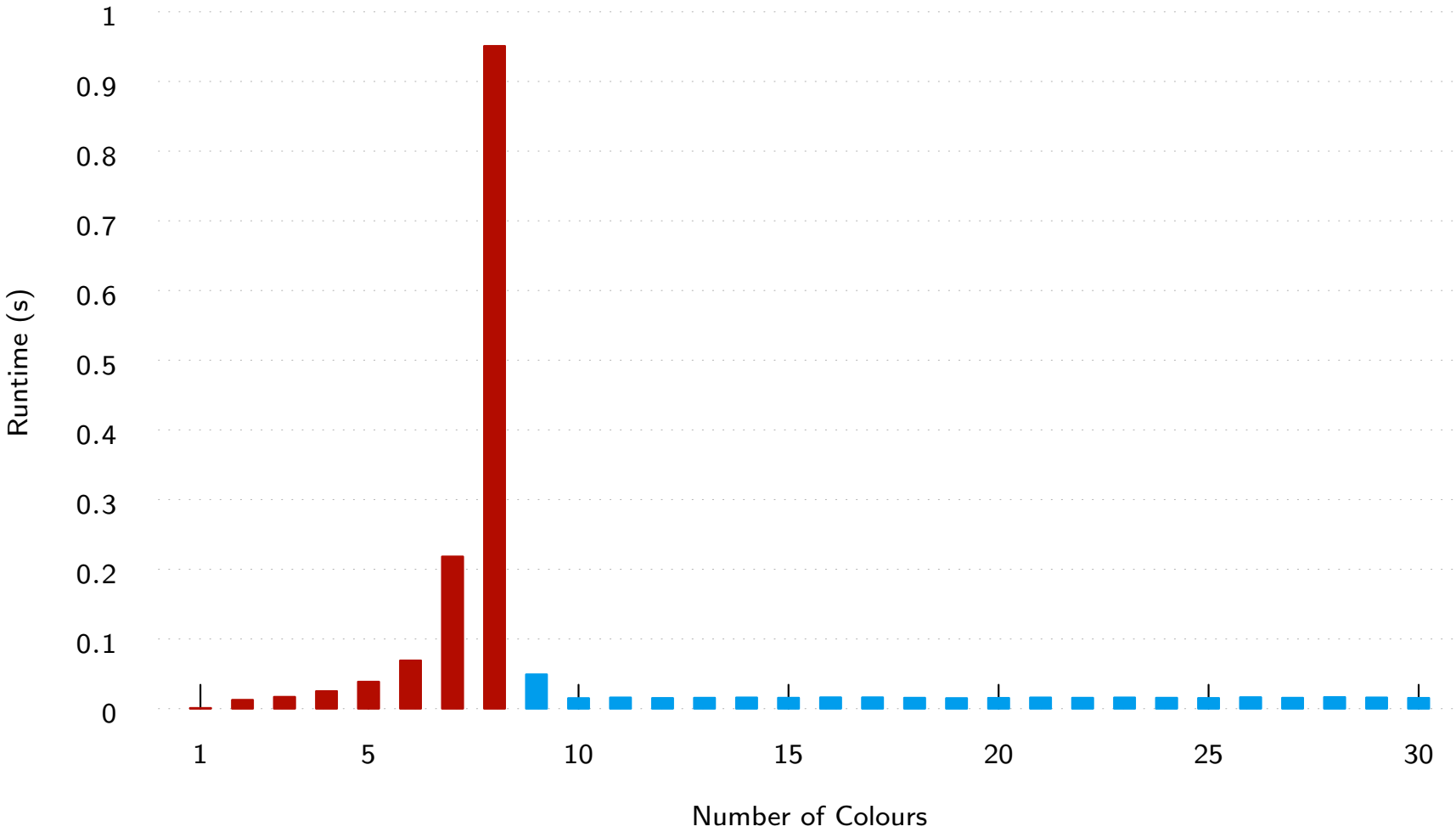


# Parallel Optimisation via Decision Problems?

```
30
1  3
1  5
1  6
1  8
1  9
1  11
1  13
1  15
1  16
1  17
1  20
1  21
1  22
1  23
1  24
1  25
1  26
1  27
1  28
1  29
```

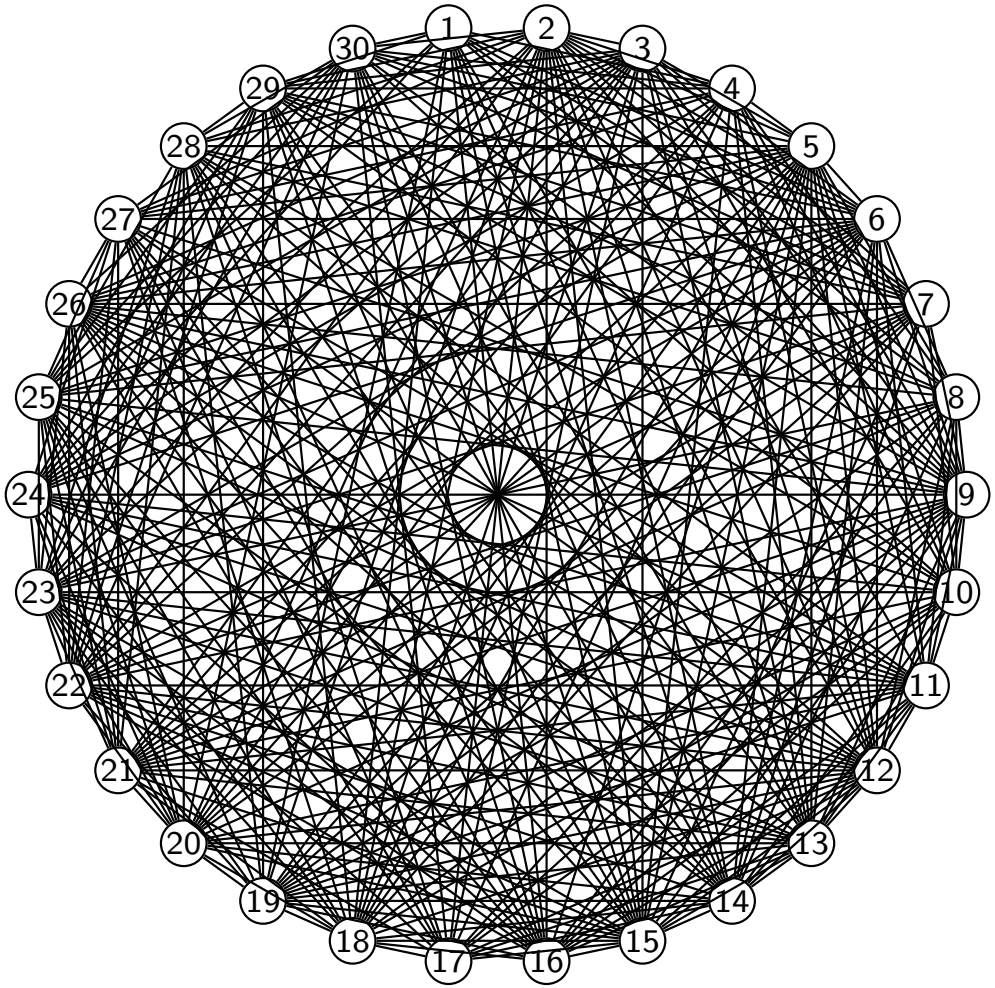


# Parallel Optimisation via Decision Problems?

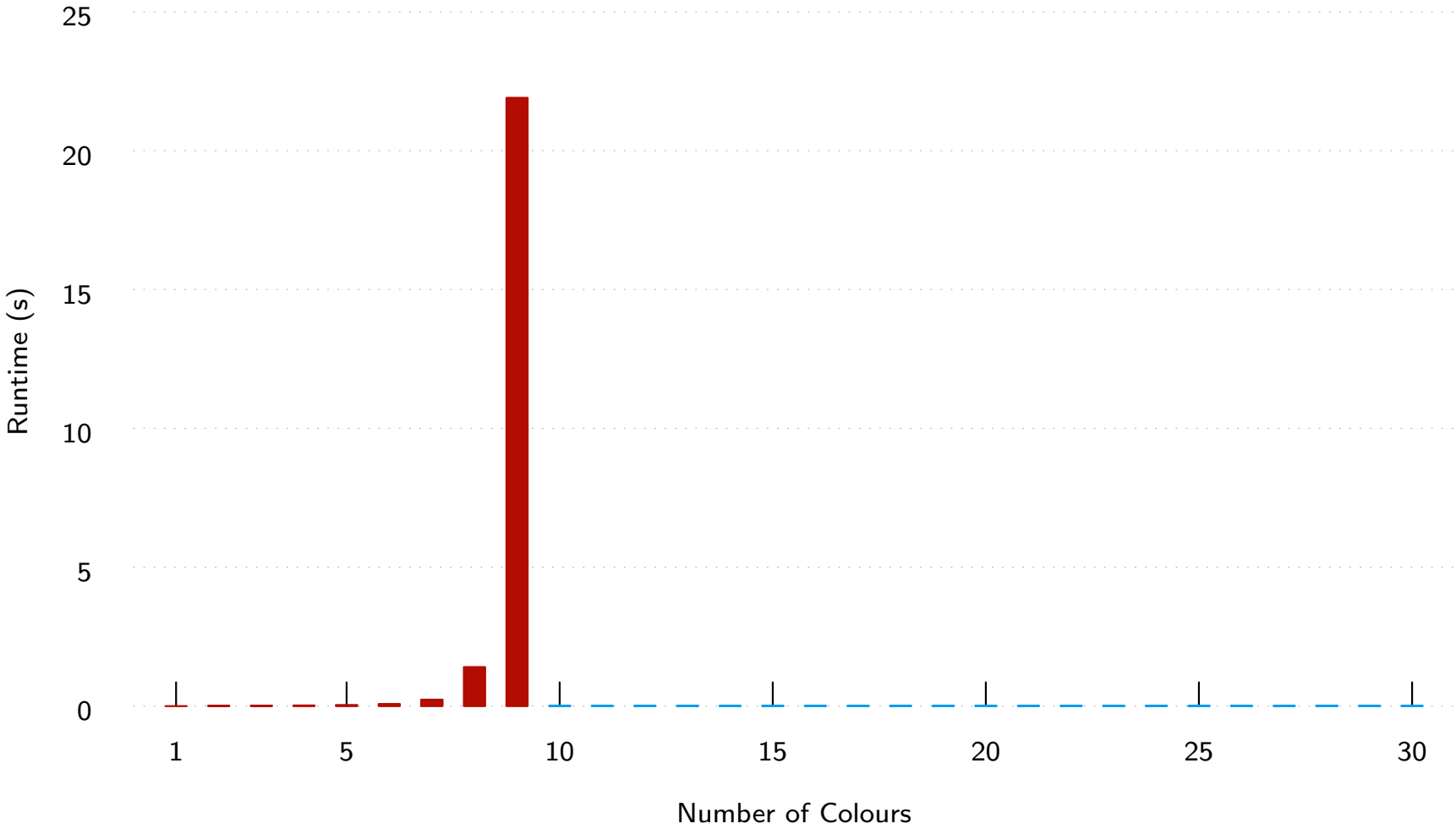


# Parallel Optimisation via Decision Problems?

```
30
1  4
1  6
1  7
1  8
1  9
1  12
1  13
1  14
1  15
1  16
1  18
1  19
1  24
1  25
1  29
1  30
2  3
2  4
2  5
2  6
```



# Parallel Optimisation via Decision Problems?



# Measuring Parallel Improvements

- *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.

# Measuring Parallel Improvements

- *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.

# Measuring Parallel Improvements

- 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

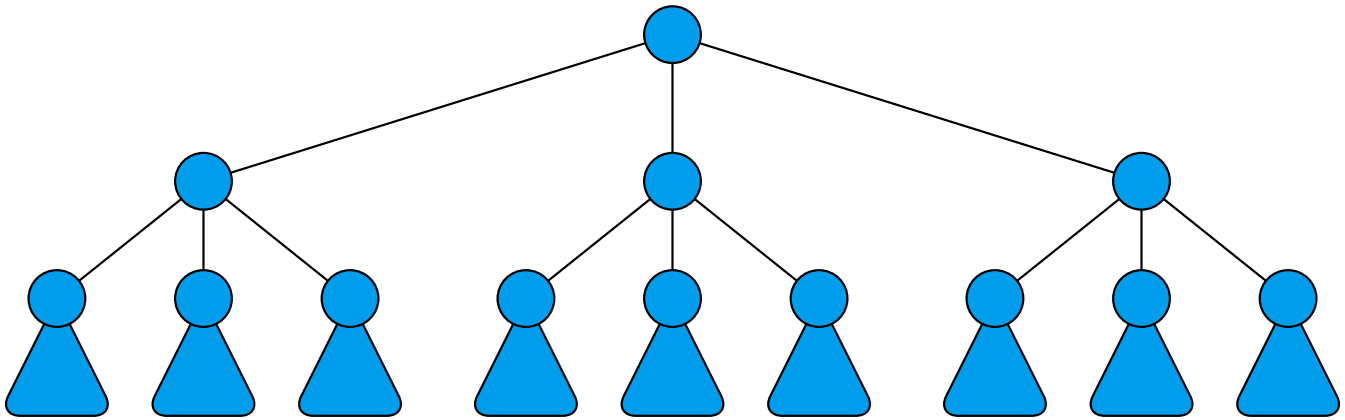
$$best\ speedup = \frac{1}{k + \frac{1}{n}(1-k)}$$

- 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.

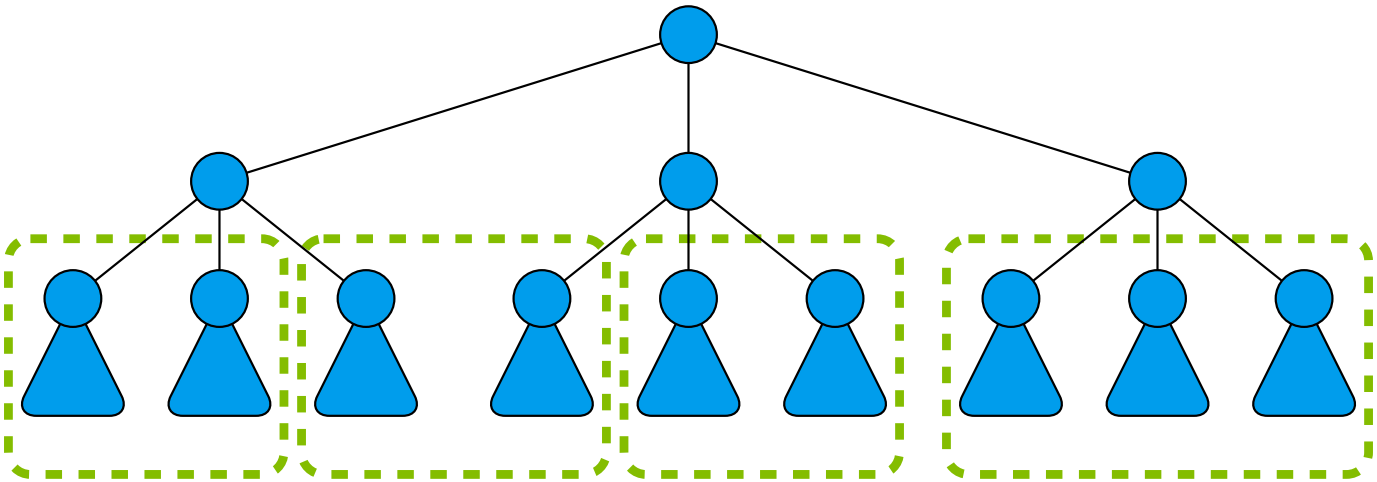
# Measuring Parallel Improvements

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

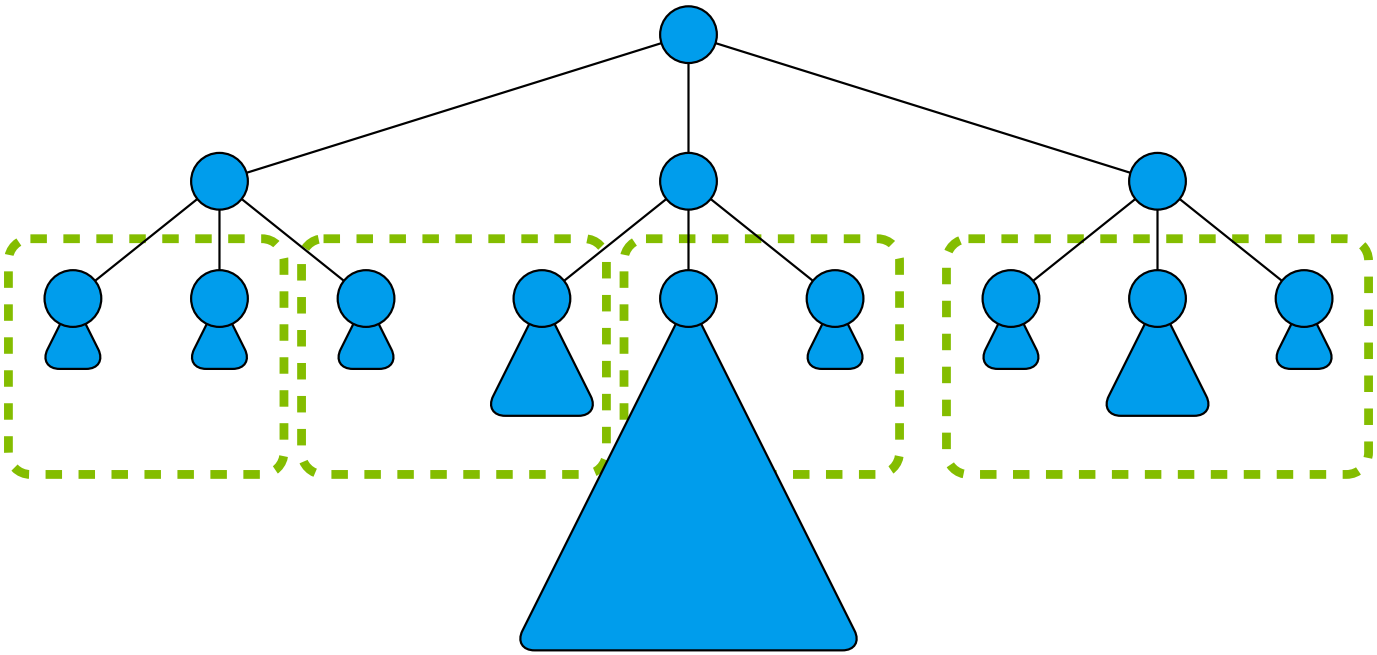
# Fixed Parallel Tree Search?



# Fixed Parallel Tree Search?



# Fixed Parallel Tree Search?



# Parallel Portfolios?

- Irregularity seems to be a problem. . . How about a different approach? We may have many choices available:
  - Models
  - Heuristics
  - Search algorithms
  - Levels of consistency
  - Solvers
- What if we try lots of different combinations in parallel?
- Now our runtimes are determined by whoever finishes first, not whoever finishes last.

# 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.

# 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.
  - Maybe only a few variables are involved, and we spend all our time bouncing around between a small number of processors...

# 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*

Experimental results show a linear speedup.

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

## 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

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).

# A Little Bit of Heresy

*a polynomial number of processors.*

# A Little Bit of Heresy

"I want to buy a polynomial number of processors."

# A Little Bit of Heresy

Intel® Xeon® Processor E7-8890 v2
(37.5M Cache, 2.80 GHz)

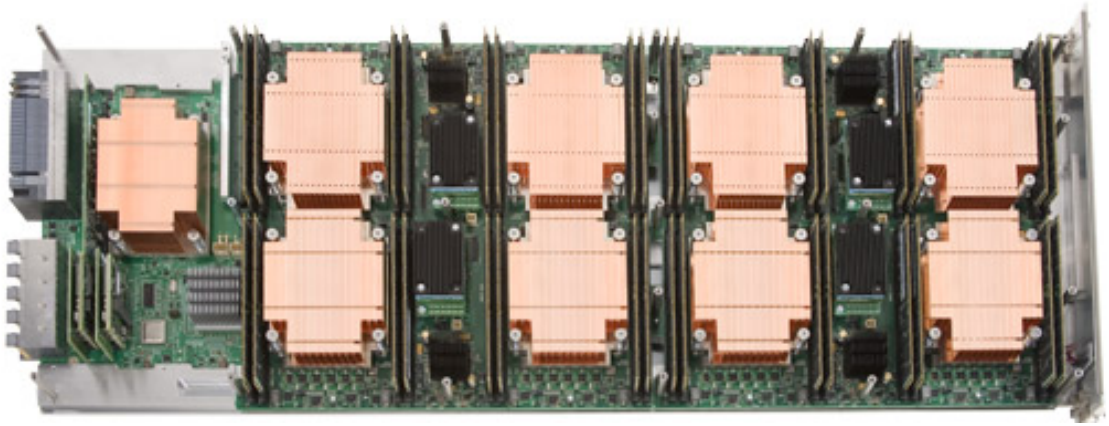
| Performance | |
|---|---|
| # of Cores | 15 |
| # of Threads | 30 |
| Processor Base Frequency | 2.8 GHz |
| Max Turbo Frequency | 3.4 GHz |
| TDP | 155 W |

# A Little Bit of Heresy

## ARCHER Hardware

The ARCHER hardware consists of the Cray XC30 MPP supercomputer, external login nodes and postprocessing nodes, and the associated filesystems. There are 3008 compute nodes in ARCHER phase 1 and each compute node has two 12-core Intel Ivy Bridge series processors giving a total of 72,192 processing cores. Each node has a total of 64 GB of memory with a subset of large memory nodes having 128 GB.

A high-performance Lustre storage system is available to all compute nodes. There is no local disk on the compute nodes as they are housed in 4-node blades (the image below shows an XC30 blade with 4 compute nodes).



# A Little Bit of Heresy

**Constants Matter: Implementing Minion, a fast Constraint Solver**

Chris Jefferson

Uversity of Oxford, Oford, UK, Chris.Jefferson@comlab.ox.ac.uk

This talk will deal with many of the practical matters of implementing am efficient constraint solver using existing algorithms and methods. SAT solvers have historically been able to solve much larger problems than CSP solvers and search thousand of times more nodes per second. This talk will discuss the implementation of constraint solver Minion, which is one of the fastest constraint solvers available and has gone some way to reducing this gap. Most of Minion's speed come from better data structures and careful use templates in C++.

# A Little Bit of Heresy

# Representing Domains

- A variable's domain contains a set of values.
- How should we implement this this?

# Representing Domains

- A variable's domain contains a set of values.
- How should we implement this this?
  - A list.
  - An array.
  - A tree.
  - A purely functional tree.
  - A hash set.
  - Just a pair of values, if we're doing bounds consistency.
  - ...

# 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.

# 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?

Give three reasons that the achieved speedup is likely to be worse than this in practice.

What if we had an unlimited number of processors?

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