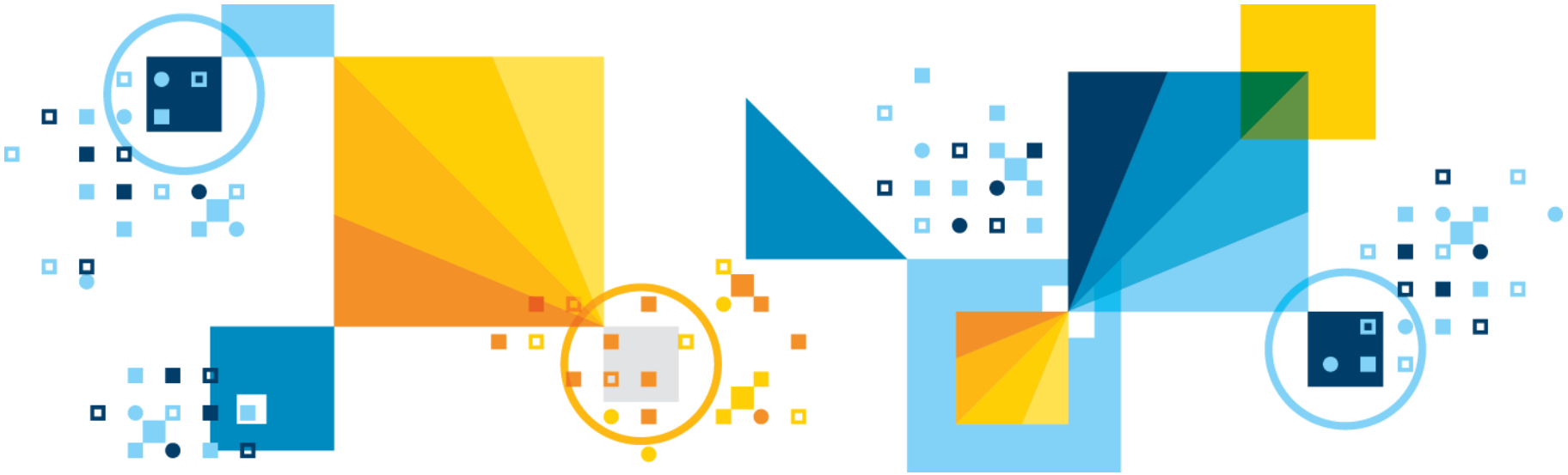
CPLEX Callbacks

@CPLEX School 2017 Montreal



Opening the black box

- CPLEX delivers excellent performance on a wide range of problems straight out of the box
- Sometimes this is not enough, and some customization is needed
- Easiest way to open the black box: parameters
  - Programmatically trivial
  - Can have a tremendous effect on performance (but pay attention on how to evaluate the changes…more on that later)
  - Sometimes still not enough
- Next step: callbacks!

What are callbacks?

Callbacks are user-defined functions (*function pointers in C*) that are executed by CPLEX when certain conditions are met during the optimization process.

```
int mycounter = 0;
int mycallback(CPXCENVptr env, void *cbdata,
               int wherefrom, void *cbhandle)
{
   int* cnt = (int*)cbhandle;
   printf("Infocb called %i times so far\n", *cnt);
   *cnt += 1;
   return 0;
}

…

CPXXsetinfocallbackfunc(env, &mycallback, (void*)&mycounter);
```

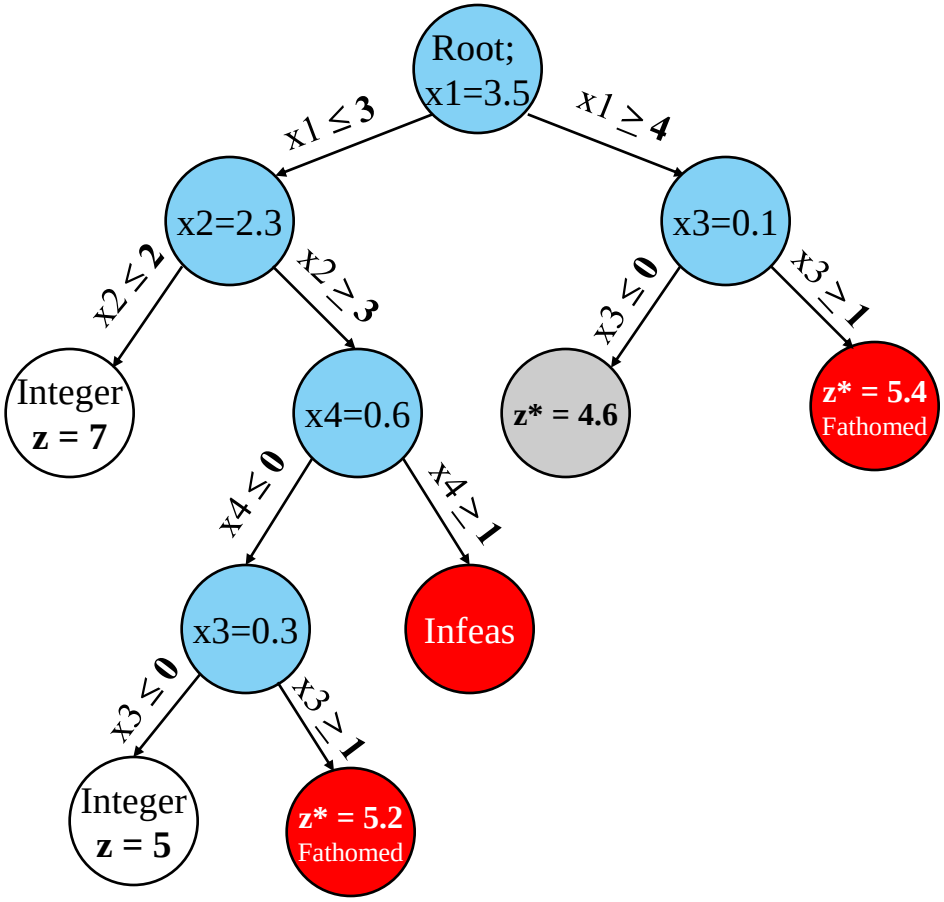
What are they used for?

- Monitor the optimization process without affecting it (except for possibly early termination)
- Influence the solution process by overriding CPLEX strategies with custom strategies, based on problem-specific knowledge
- Implement advanced solution techniques like solving models with an exponential numbers of constraints

Which callbacks are available in CPLEX?

- Info Callback:
  - Compatible will all search strategies/algorithms (even distmip!)
  - Access only to global statistics about the current optimization
  - Mostly to monitor progress or trigger early termination
- Control Callbacks:
  - Used to implement custom strategies within the B&C algorithm
  - Not compatible with defaults settings, for mathematical, historical and technical reasons
  - Have access to presolved space and node LPs

LP-based Branch and Cut (B&C)

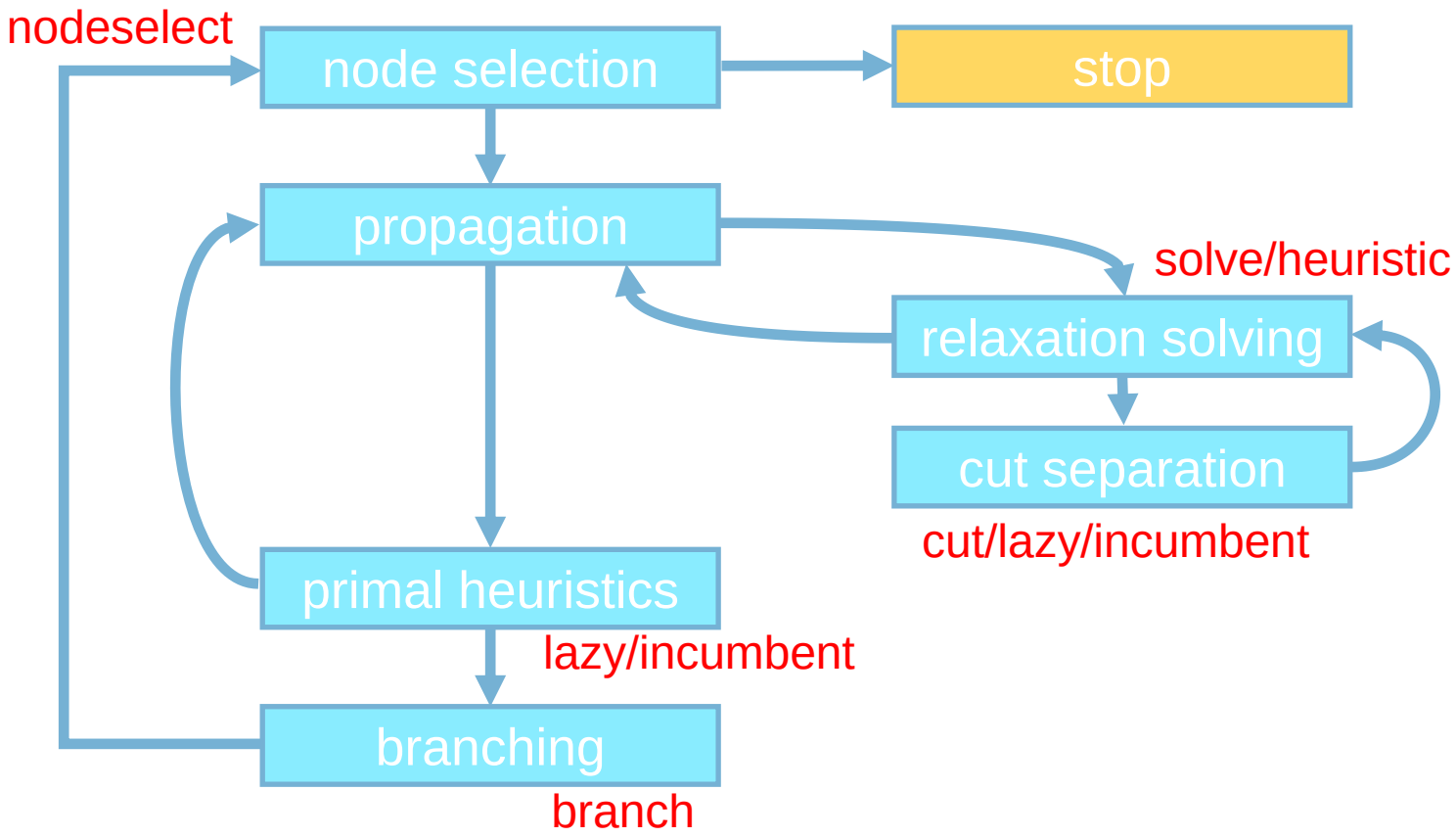


- B&C algorithm:
  - Enumerative scheme based on the LP relaxation of MIP
- **Bounding**:
  - Nodes with **z* ≥ UB** can be **fathomed** without further ramification.
- Key points to avoid exponential explosion of B&C tree:
  - Strong LP relaxation
  - Effective branching rules
  - Primal heuristics

Lazy vs. User cuts

- User cuts are conventionally defined to mean constraints which can change the feasible space of the continuous relaxation but do not rule out any feasible integer solution.
- Lazy constraints represent one portion of the constraint set, and the model would be incomplete (and possibly deliver incorrect answers!) in their absence.
- CPLEX has a lot of freedom with user cuts: if and when to add them, how to purge, etc.
- Any integer solution **must** be checked against lazy constraints

Branch& Cut + Control Callbacks Flow Overview



Info Callback

- Compatible will all search strategies/algorithms (even distmip!)
- Access only to global statistics about the current optimization
- Mostly used to:
  - monitor progress
  - trigger early termination
  - implement complex termination criteria not ready available with parameters (e.g., stop after 30 minutes but only if gap is below 5%)
- Never called concurrently

Tips & Tricks: do not expect it to be called on a fixed regular basis

Branch Callback

- CPLEX already picks its branching candidate, which is passed as input to the callback
- User can:
  - Accept the default and do nothing
  - Branch as CPLEX, but attaching user data to newly created nodes
  - Override branching decision (again, optionally attaching user data)
- User can branch on arbitrary linear constraints
- Can create at most two child nodes
- Need to disable dual/nonlinear reductions to work in the original space

Tips/Pitfalls

- *if always overriding, switch CPLEX to a very cheap branching rule*
- *Delete user data attached to nodes via the* CPXXsetdeletenodecallbackfunc

Heuristic Callback

- Given fractional solution x* (and its objective value) in input
- User can use the given x* to feed his own heuristics
- If a new solution is found, the user can use the very same array to report back to CPLEX (together with the new objective value)
- Can optionally ask CPLEX to double-check feasibility of the new solution
- Can add only one solution at a time

Tips/Pitfalls

- *Beware of* checkfeas=FALSE*! CPLEX will make no checks at all not even on the objective value!*
- *Not called when a new incumbent is found (not really good for improvement heuristics)*
- *Called even when heuristics are switched off.*

Lazy/User cut Callback

- Ask CPLEX the current integer (resp. fractional) solution to cut via a lazy (resp. user) cut via CPXXgetcallbacknodex
- Can return both global and local cuts
- Can tell CPLEX when and how to purge cuts added via a callback
- Within the usercuts callback, the user can:
  - Tell whether we are in the middle of a cutloop or at the end
  - Instruct CPLEX to abort the cutloop and move to branching
- Need to disable nonlinear reductions to work in the original space
- Need to disable dual reductions if adding lazy constraints

Tips/Pitfalls:

- *do not assume your cuts are added to the nodeLP immediately (or at all)!*
- *Because of numerics, always impose a limit on the number of times you return cuts on a given node.*

Example of lazy and dual reductions

$$\begin{cases} \min 3x_1 + 2x_2 \\ x_1 + 2x_2 \leq 3 \\ x_1, x_2 \in \{0, 1\} \end{cases}$$

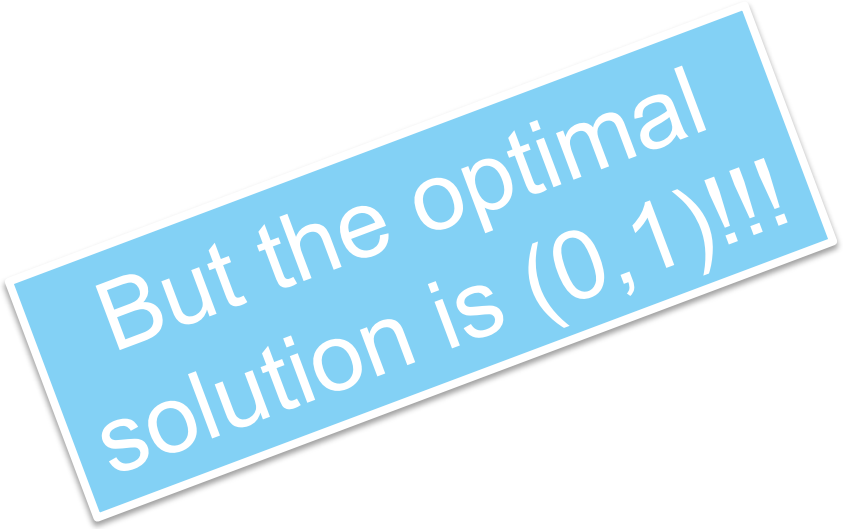
dual reductions

$$x_1 = x_2 = 0$$

violated by lazy constraint

$$x_1 + x_2 \geq 1$$

Model Infeasible



Incumbent Callback

- Pending incumbent is passed in input
- Can reject it without providing any reason/cut
- No need to disable any reduction

Tips/Pitfalls: do not use it to track the incumbent, as it disables some algorithm components. Use an info callback instead!

Solve Callback

- Used to get control of which algorithm to use to solve the node LP
- Can only use CPLEX algorithms to solve the node LP!

Tips/Pitfalls: hardly ever used, so think twice!

How to use control callbacks

- Ideally rather simple:
  - just implement your custom strategies
  - pass the corresponding function pointers to CPLEX
  - start the optimization
- In practice, using control callbacks is more involved, mostly because of:
  - Presolve
  - Parallel
  - Side-effects

Tips/Pitfalls

- *Adding a control callback will automatically switch CPLEX to sequential B&C*
- *Control callbacks default to presolved space*

Presolve

Presolve space vs. Original space

- CPLEX usually solves a very different problem than the one you submitted to it…
- MIP Presolve is quite sophisticated:
  - Lots of reductions currently implemented that use a wide array of arguments (and the list is growing…)
  - Presolved model might look completely different from the original model (i.e., it is not just the original model with some variables fixed)
- Usually you don't have to care:
  - CPLEX does all the book-keeping to be able to crush/uncrush solution vectors
  - In the end it always reports solutions in the original space!
- **BUT…**

Presolve space vs. Original space (II)

- You want to use callbacks, and thus need to interact with CPLEX in the middle of the solution process!
- Two options (**both** available):
  - **Work in the original space:** CPLEX needs to map back and forth (crush/uncrush) any piece of information you ask/provide.
  - **Work in the presolved space:** no crushing/uncrushing needed, but original structure is potentially lost.

Working in the presolved space

- No need to crush/uncrush anything
- No restrictions: solver at (almost) full speed
- Access to node LP

- Original structure is gone
- Applicable only if doing research on general-purpose methods
- Presolved model can change anytime (e.g. because of root represolve and other magic)

Working in the original space

- Most common in applications
- User can exploit his knowledge about the model
- Need to crush/uncrush objects

- It is not always possible to crush/uncrush linear forms (cuts, branching constraints).
- Might need to disable some reductions in order to work correctly.
- No access to the node LP

What about those nonlinear reductions???

- Some reductions do not allow to map linear functions back and forth
- Example: binary expansion*

$$y \in \{0, \ldots, 7\} \Rightarrow y = 4x_2 + 2x_1 + x_0$$

branch in presolved space

$$x_1 \leq 0 \vee x_1 \geq 1$$

uncrush branch dichotomy

$$y \in \{0, 1, 4, 5\} \vee y \in \{2, 3, 6, 7\}$$

Non-linear: cannot be applied!!!

*purely hyphotetical reduction

Primal vs. dual reductions

- Primal reductions change the problem only based on feasibility arguments.
- Dual reductions change the problem based on optimality arguments: in particular, they might remove feasible (even optimal!) solutions provided at least one optimal solution is left.
- Dual reductions **must** be switched off when adding lazy constraints!!! (CPLEX does it automatically)

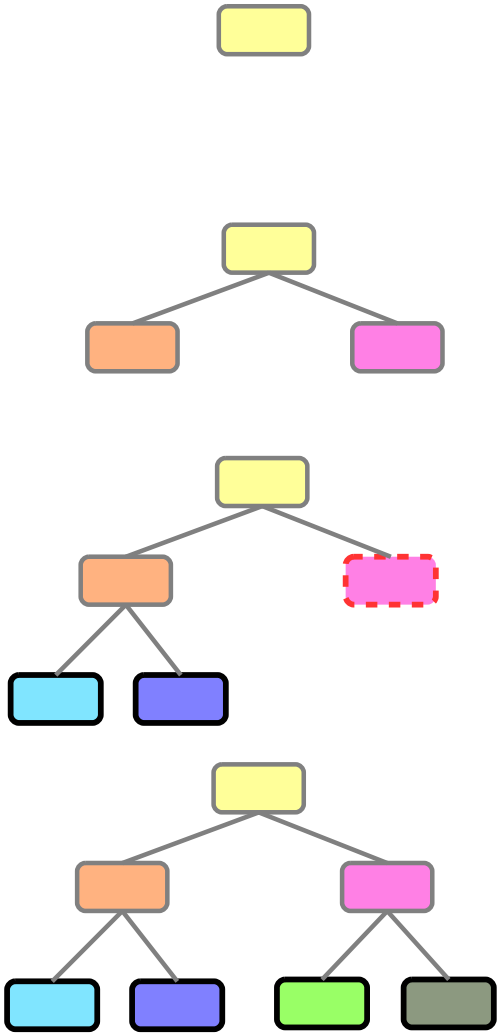
Question: what about reduced cost fixing?

Presolve recap

- With control callbacks, some care must be taken with certain presolve settings.
- Depending on the callback in use, you might need to disable nonlinear reductions, dual reductions or both!
- This applies in particular to:
  - Usercut/lazycut callbacks
  - Branch callback
- ***Tips & Tricks***: what about switching off presolve entirely? Or do the presolve once and for all?

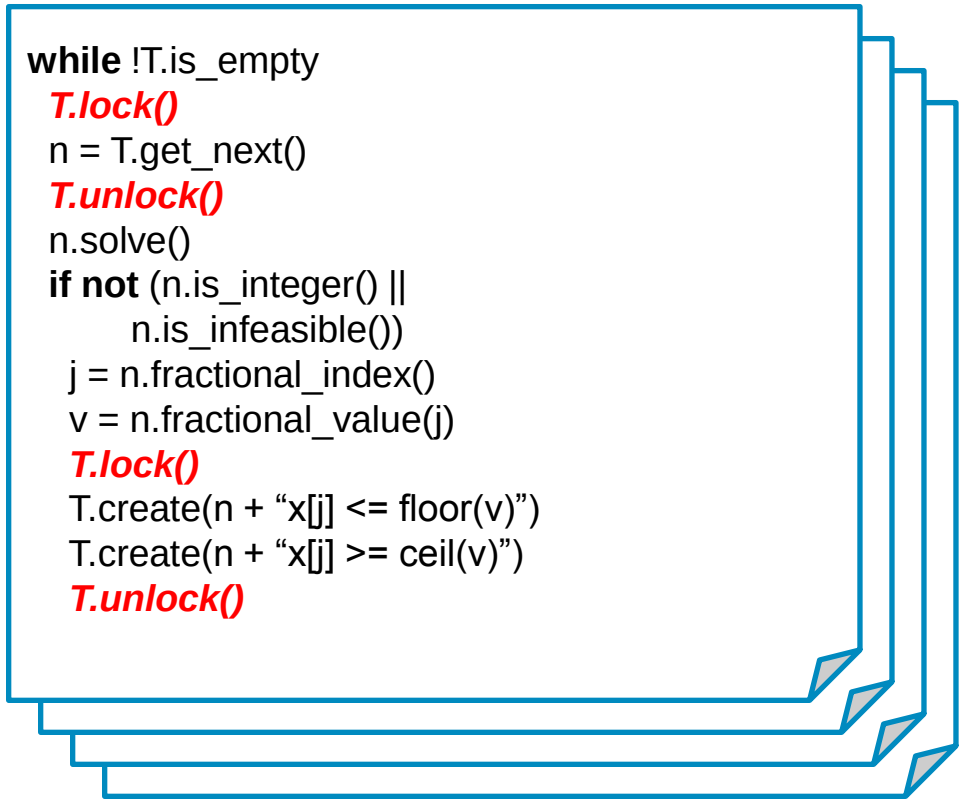
Parallel

Sequential Branch-and-Bound



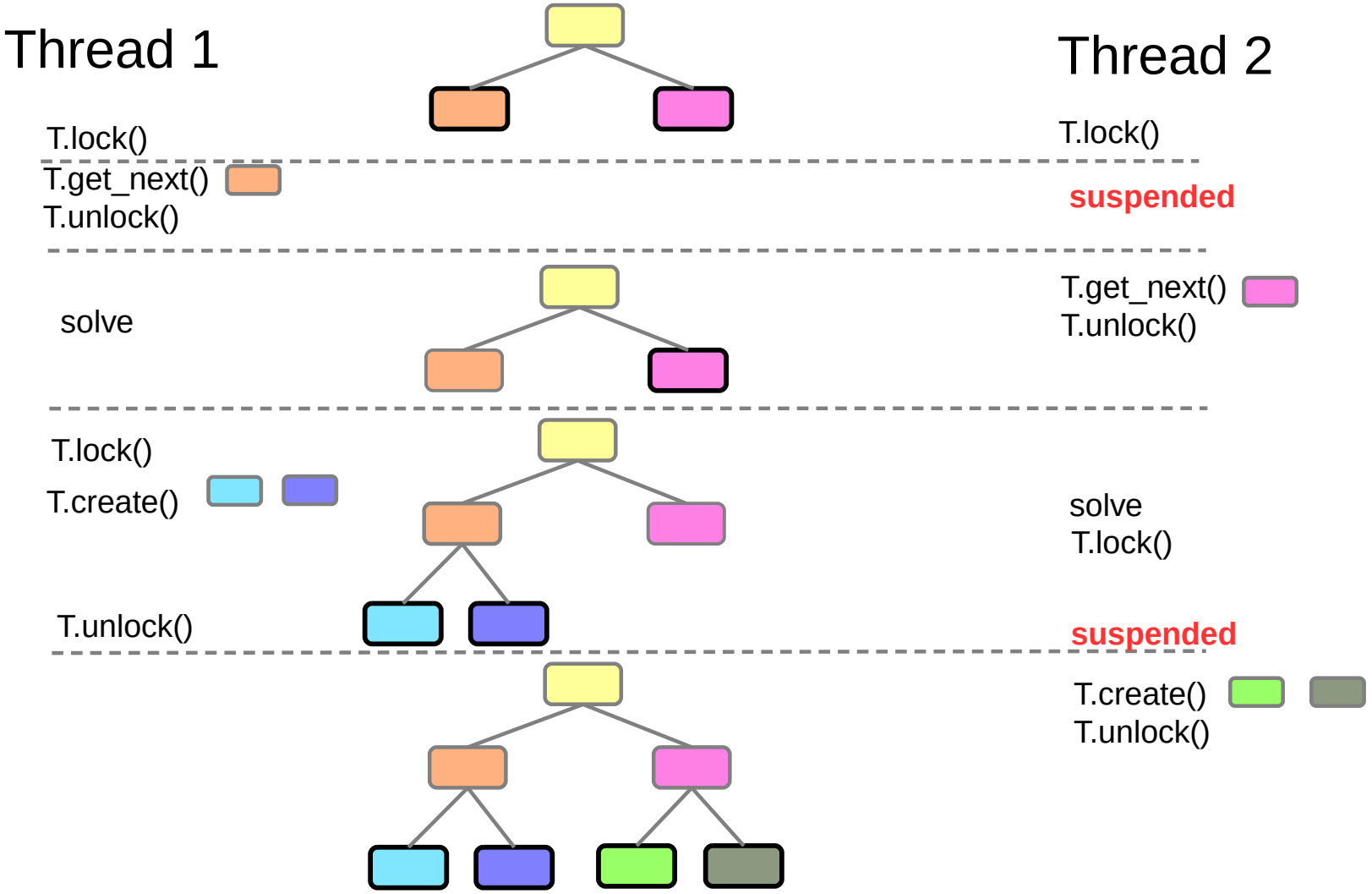
```
while !T.is_empty
  n = T.get_next()
  n.solve()
  if not (n.is_integer() ||
          n.is_infeasible())
    j = n.fractional_index()
    v = n.fractional_value(j)
    T.create(n + “x[j] <= floor(v)”)
    T.create(n + “x[j] >= ceil(v)”)
```

Parallel Branch-and-Bound

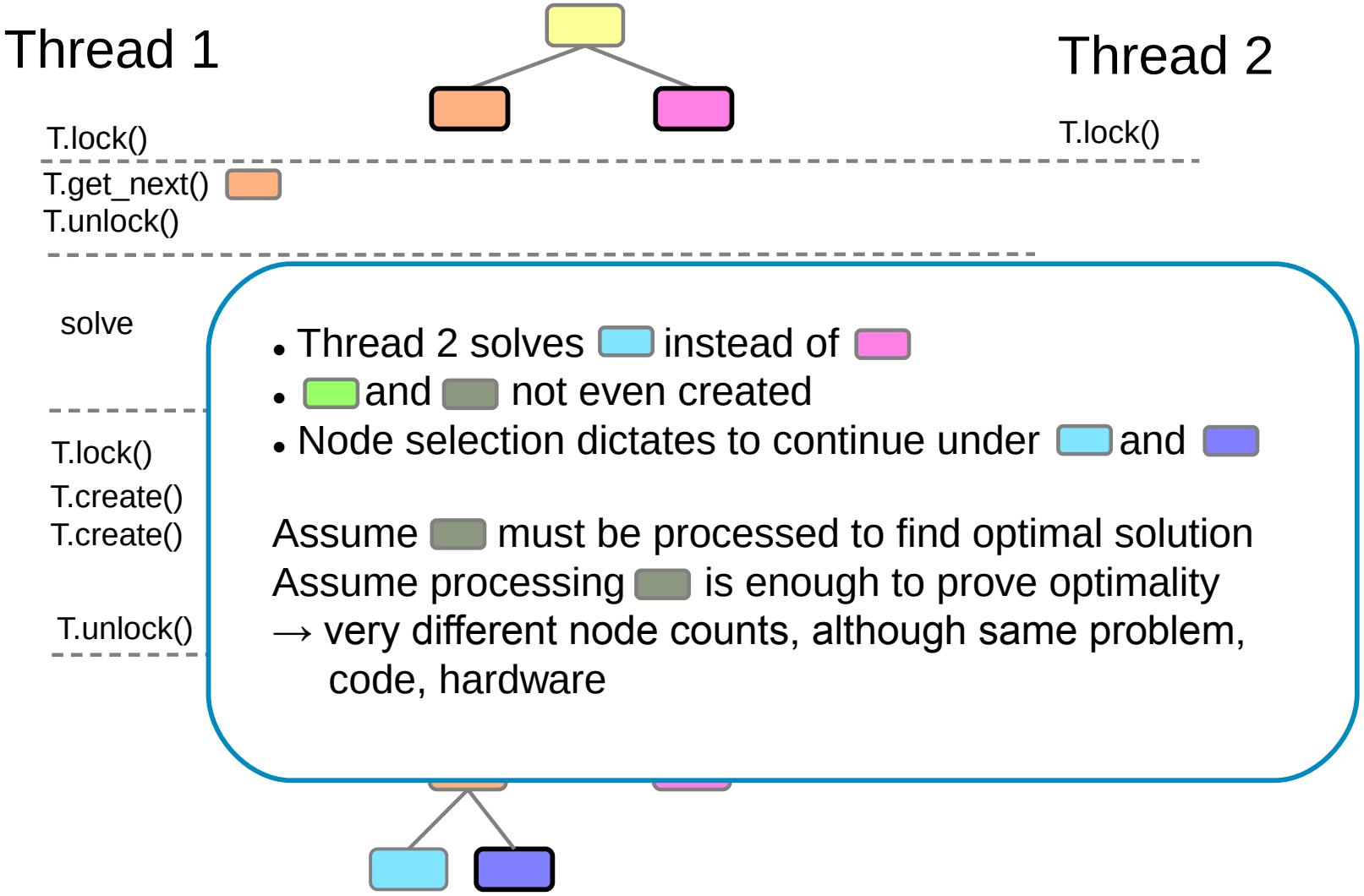


- All threads execute the same code
- Needs locking when accessing the tree

Parallel Branch-and-Bound: example



Parallel Branch-and-Bound: not deterministic!



Issues with non-determinism

- opportunistic behavior of OS scheduler
- very minor side effects (cache misses, page swaps, …)

  can change order of threads

gmu-35-40			*iis-pima-cov*			*glass4*		
Time	Iterations	Nodes	Time	Iterations	Nodes	Time	Iterations	Nodes
21.31	2236199	444838	**62.92**	**1678290**	**17744**	41.60	6577338	340192
18.46	2060789	322282	139.72	5475373	47428	70.57	11800428	637471
4.62	**253871**	**51778**	84.39	2075243	26460	**29.30**	**4075955**	**304913**
31.55	3300767	509641	80.37	2766163	22016	34.31	6026804	208466
11.89	1238162	177236	71.25	1753092	17513	160.12	29207354	1082067
79.50	**7212334**	**1307340**	69.18	1615592	21636	**244.62**	**35189917**	**1893985**
21.10	2955104	595939	117.43	3002680	40299	54.40	8700758	512053
16.27	1953450	401507	72.41	1704471	21301	74.64	11307243	698508
13.61	1410170	261508	124.01	4863607	42298	47.19	8600157	389436
17.10	1948228	410949	**191.31**	**7898071**	**71043**	54.42	8275014	544172

Issues with non-determinism

- opportunistic behavior of OS scheduler
- very minor side effects (cache misses, page swaps, …)

  can change order of threads

gmu-35-40			*iis-pima-cov*			*glass4*		
Time	Iterations	Nodes	Time	Iterations	Nodes	Time	Iterations	Nodes
4.62	**253871**	**51778**	**62.92**	**1678290**	**17744**	**29.30**	**4075955**	**304913**
79.50	**7212334**	**1307340**	**191.31**	**7898071**	**71043**	**244.62**	**35189917**	**1893985**

This is a problem in practice:

- results are not repeatable
- debugging becomes painful
- benchmarking is complicated
- assessment of algorithmic changes is difficult

We need a solver that operates in a deterministic way!

Deterministic Multi-threading

opportunistic	• OS scheduler, cache misses, swaps, ... • order in which lock is granted to threads is not deterministic
deterministic	Make this order deterministic: • use *deterministic* time (to be defined) • grant lock to first thread to arrive in *deterministic* time
Kendo	"Kendo: Efficient Deterministic Multithreading in Software" http://people.csail.mit.edu/mareko/asplos073-olszewski.pdf → use number of retired store instructions as time (x86)
CPLEX	deterministic time = number of array accesses • works the same on all hardware • explicit instrumentation of source code

```
for (int i = 0; i < n; ++i)
 x[i] = y[i] * 0.5;
DETTIME_INCREMENT (2 * i);
```

Deterministic multi-threading

gmu-35-40		
Time	Iterations	Nodes
4.62	253871	51778
79.50	7212334	1307340
38.48	4124364	737994

iis-pima-cov		
Time	Iterations	Nodes
62.92	1678290	17744
191.31	7898071	71043
72.84	1382605	20449

glass4		
Time	Iterations	Nodes
29.30	4075955	304913
244.62	35189917	1893985
91.68	11737409	528160

Deterministic threading is not for free:

- Settles for a particular path through the search tree
- Increases wait/idle times in locks

Multi-threading and callbacks

- How does parallel affect callbacks?
- The overall code with callbacks must stay correct (of course) and, if needed, deterministic.
- Not trivial:
  - Callback writers must take all measures to avoid:
    - Race conditions
    - Non-deterministic behaviour
  - Writing parallel code is hard in general! ☹

Parallel gotchas and pitfalls

- Let's start with correctness first…
- By default, setting a control callback will force CPLEX to sequential B&C
  - Correctness is easy ☺
  - Performance might suffer greatly ☹
- Parallel can be forced by setting the number of threads explicitly
- Depending on the parameters, CPLEX might or might not serialize callback calls from different threads. To avoid race conditions:

Always lock access to global non-constant data structures

Parallel gotchas and pitfalls (II)

- And now on determinism!
- Even if CPLEX serializes callback calls, the order of callback invocations is **not** deterministic (even for deterministic parallel B&C!).
- In order to keep determinism:

Only write to thread-local data, if any!

Side Effects

Side effects of control callbacks

- Some control callbacks are incompatible with some CPLEX features, most notably dynamic search.
- For technical and historical reasons, other (optional) components are switched off in case control callbacks are installed.
- Preparing the arguments to be passed to callback functions requires some (small) work, that however affects tick counts, and thus the path.
- **Empty callback != No callback**
- For benchmarking your callback-based algorithm, always compare against empty callback! Then also against CPLEX defaults to see if it was all worth it…

Legal Disclaimer