Top 10 CPLEX Parameters

to Turbocharge Your AMPL Model

A High-level Guide for Technical Users

Mastering Optimization: A Guide to Solver Performance Tuning (CPLEX)

High-performance mathematical optimization has two fundamental parts: building a robust formulation and efficiently tuning the engine that solves it. While modern solvers are highly sophisticated, their default settings are designed for general performance. Tuning allows the engine to be tailored to the specific mathematical structure of a unique problem.

1. The Mechanics of MIP Tuning

1.1 The Purpose of Tuning

Tuning is the strategic adjustment of a solver's internal behavior to improve its performance on a particular problem, which can be viewed as reducing the time required to find an optimal solution or a high-quality feasible point. It provides value by maximizing hardware efficiency, meeting strict operational time limits, and handling numerically difficult models that defaults might struggle to solve. It's important to note that the solution quality and time requirements are defined by the problem context and as such, they are different for every case.

1.2 The General Structure of a MIP Solver

Before diving in the tuning process, it is crucial to understand how most modern Mixed-Integer Programming (MIP) solvers work. They share a common architectural foundation and follow similar patterns to solve the problems:

- **Presolve:** The solver simplifies the model before the main search. It identifies redundant constraints, fixes variables, and tightens bounds to create a smaller, more "tractable" problem.
- **Root Relaxation (First Solution):** usually, the solver starts by ignoring integrality constraints (treating integers as continuous) to solve the **LP Relaxation**.
    - **LP Algorithms:** Solvers use **Simplex** (moving along the edges of the feasible region) or **Barrier** (moving through the interior) methods to find this initial bound.
    - **Heuristics:** These are special algorithms designed to find a valid integer solution early, even if it isn't the best one. They tend to use the structure of the problem that's being solved.
- **Branch & Bound (B&B) and Branch & Cut (B&C):**
    - These are the main algorithms used to solve a MIP problem. Essentially, they work by partitioning the feasible region into smaller subproblems (branching) and using bounds to prune the search tree. It maintains a global "best" solution found so far and compares it against the estimated potential (the bound) of each branch, closing the gap between the two with each iteration.
    - **The Search Tree:** If the root solution isn't integer, the solver "branches" by creating sub-problems. **Cuts** (additional constraints) are added to prune the search space without losing feasible solutions.

- **Primal and Dual Bounds:** The **Primal Bound** is the best integer solution found so far (incumbent). The **Dual Bound** is the theoretical limit of how good the solution could be (LP relaxation of the problem).
- **The Balance:** Performance tuning is a tug-of-war between finding better feasible solutions (improving the Primal Bound via **Heuristics**) and proving that no better solution exists (improving the Dual Bound via **Cuts**, branching and effective pruning).

1.3 How Parameters Influence the Engine

Solver parameters are the entry point to control these components. They can dictate the frequency of heuristics, the aggressiveness of cuts, or manage **physical resources** (e.g., allocating CPU threads and memory). They also define **termination criteria**, such as the *mipgap* (how close the primal and dual bounds must be to stop) or *timelimit*.

1.4 The Importance of Solver Logs

The solver log is a key part of performance tuning. It is the blueprint of the optimization process and works as the primary tool for diagnosing performance bottlenecks.

- **LP Logs:** Detail the number of iterations, objective value progress, and method used (Simplex/Barrier).
- **MILP Logs:** Provide a table showing the progress of the Primal Bound, Dual Bound, and the resulting Gap over time. Understanding these transitions is essential; for instance, if the Dual Bound hasn't moved for several nodes, the branching strategy or cut aggressiveness likely benefits from adjustment.

1.5 Portability of Concepts

While names and specific functionalities may vary (e.g., Gurobi's *MIPFocus* vs. CPLEX's *MIPEmphasis*), the underlying main algorithmic controls are largely consistent across major solvers.

2. Top CPLEX Parameters

This article focuses on tuning the CPLEX solver. AMPL provides an interface to change any solver parameter. It even allows to name common parameters with aliases so you don’t have to remember the exact name of a parameter for every solver. Additionally, it includes a naming convention (*category:name*) to easily look for a parameter. A complete list of CPLEX parameters in AMPL and their description can be found here. However, here are the top 10 parameters that would probably help you tune your model performance:

CPLEX Name	AMPL aliases	Explanation
mipemphasis	mip:emphasis (mipemphasis, mip:focus, mipfocus)	MIP solution strategy: ● 0 - Balance finding good feasible solutions and proving optimality (default) ● 1 - Favor finding feasible solutions ● 2 - Favor providing optimality ● 3 - Focus on improving the best objective bound ● 4 - Focus on finding hidden feasible solutions ● 5 - Focus on finding high quality solutions earlier (heuristic). This value is not valid for CPLEX version 22.1.2.
threads	tech:threads (threads)	How many threads to use when using the barrier algorithm or solving MIP problems; default 0 ==> automatic choice.
presolve	pre:solve (presolve)	Whether to use CPLEX's presolve: ● 0 - No ● 1 - Yes (default)
mipgap	mip:gap (mipgap)	Max. relative MIP optimality gap (default 1e-4).
scale	pre:scale (scale)	How to scale the problem: ● -1 - No scaling ● 0 - Equilibration (default) ● 1 - More aggressive.
mipcuts	cut:cuts (cuts, mipcuts)	Global cut generation control, valid unless overridden by individual cut-type controls: ● -1 - Disallow ● 0 - Automatic (default) ● 1 - Enable moderate cuts generation ● 2 - Enable aggressive cuts generation. ● 3 - Enable very aggressive cuts generation. Note: this option is only available through AMPL. In CPLEX there's no global cuts control but the setting must be specified for each type of cut (cliques, covers, disjcuts, flowcuts, etc).
varbranch	mip:varbranch (varbranch, varsel, varselect)	MIP branch variable selection strategy: ● -1 - Branch on variable with smallest infeasibility ● 0 - Algorithm decides (default) ● 1 - Branch on variable with largest infeasibility ● 2 - Branch based on pseudo costs ● 3 - Strong branching

		• 4 - Branch based on pseudo reduced costs
method	alg:method (method, lpmethod, qpmethod, simplex, mip:method, mipstartalg)	Which algorithm to use for the root node of MIP problems: • 0 - Automatic (default) • 1 - Primal simplex • 2 - Dual simplex • 3 - Network simplex • 4 - Barrier • 5 - Sifting • 6 - Concurrent
nodemethod	mip:nodemethod (nodemethod, mipalg, mipalgorithm)	Algorithm used to solve relaxed MIP node problems (after the initial relaxation) • 0 - Automatic (default) • 1 - Primal simplex • 2 - Dual simplex • 3 - Network simplex • 4 - Barrier • 5 - Sifting
heuristicfreq	heurfreq, heuristicfreq	How often to apply "node heuristics" for MIPS: • -1 = never • 0 = automatic choice (default) • n > 0 = every n nodes. For this one, you have to set the solver to 'cplexasl'. See here for more details.

Using **amplpy**, these parameters can be easily set with a single line of code:

```
from amplpy import AMPL
ampl = AMPL()
# Passing multiple parameters through the option string
ampl.set_option('cplex_options', 'mipemphasis=1 threads=4 mipgap=0.01')
```

3. General Recommendations

1. **Let Logs Guide You:** If the log shows the Dual Bound is flat, increase cut intensity. If no integer solution is found for a long time, increase heuristic frequency.
2. **The Baseline Rule:** Solvers improve significantly with every version. Always re-test your tuning when upgrading the solver version.
3. **Cross-Validation:** Tune over multiple data sets. A parameter set that speeds up "Instance A" might cause "Instance B" to fail.
4. **Avoid Overfitting:** Don't change ten parameters at once. Usually, adjusting 1 or 2 key parameters provides 80% of the potential gain.
5. **Pareto Frontier:** Be aware that faster solution times often come at the cost of slightly worse optimality or higher memory usage.

6. **Reformulation First:** A more efficient mathematical representation of the problem (e.g., breaking symmetry) if possible, is almost always more effective than parameter tuning.

4. Automated Tuning Tools

Some solvers like CPLEX include a built-in **Tuning Tool** that automates the trial-and-error tuning process. This tool will perform multiple runs to find the optimal parameter set for your specific model. Please note that this is **computationally expensive** as it requires solving the model (or segments of it) many times. It should only be used during the development phase to fine tune the solver and never in a production real-time environment.

Finally, results from the tuning tool should be taken only as suggestions, and must be tested furtherly. It is often the case where some of the auto-tuning results provide little difference, especially when applied to multiple instances of the problem.

4.1 Implementation in AMPL

Using the CPLEX tuning tool via **amplpy** is as simple as defining the related options. The most basic command would be to define the tuning results file, indicating the solver to print which parameter is it trying, and setting a time limit for the whole tuning process:

```
from amplpy import AMPL
ampl = AMPL()

ampl.solve(solver="cplex", cplex_options='outlev=1 tunebase=tuning_results.txt
tunedisplay=2 tunetime=300')
```

End of the Top 10 CPLEX Parameters

For additional support, [contact AMPL Optimization]