Operations Research Letters 42 (2014) 186–189

Contents lists available at ScienceDirect

Operations Research Letters

journal homepage: www.elsevier.com/locate/orl

Using diversification, communication and parallelism to solve mixed-integer linear programs

R. Carvajal [a,b,*], S. Ahmed [a], G. Nemhauser [a], K. Furman [c], V. Goel [d], Y. Shao [d]

[a] Industrial and Systems Engineering, Georgia Institute of Technology, United States
[b] School of Business, Universidad Adolfo Ibáñez, Chile
[c] ExxonMobil Research and Engineering, United States
[d] ExxonMobil Upstream Research Company, United States

ARTICLE INFO

Article history:
Received 7 May 2013
Received in revised form
13 November 2013
Accepted 31 December 2013
Available online 14 February 2014

Keywords:
Integer programming
Branch-and-bound
Diversification
Communication
Parallelism
Performance variability

ABSTRACT

Performance variability of modern mixed-integer programming solvers and possible ways of exploiting this phenomenon present an interesting opportunity in the development of algorithms to solve mixed-integer linear programs (MILPs). We propose a framework using multiple branch-and-bound trees to solve MILPs while allowing them to share information in a parallel execution. We present computational results on instances from MIPLIB 2010 illustrating the benefits of this framework.

1. Introduction

Mixed-integer linear programming (MILP) problems are commonly solved using a linear programming-based branch-and-cut scheme. The scheme proceeds by partitioning the space of solutions in the form of a search tree obtained by fixing variable bounds (branching). Then it uses lower bounds from linear programming (LP) relaxations, typically tightened by the addition of cutting planes, and upper bounds, from feasible solutions, to prune parts of the search space. Modern implementations of this generic scheme involve a multitude of options, e.g., how to select a partition to further subdivide (node selection), how to select a variable to branch on (variable selection), what kinds of cutting planes to add and how often to add them, what types of heuristics to use and how often to use them and so on. It is well known that different choices of these options can have very significant effect on the performance of the branch-and-cut scheme [1]. In addition to these algorithmic options, it has been noted in [2] that the performance of a specific implementation on a particular problem instance can vary very significantly with less understood factors, such as the computational environment, random seeds used in the inner workings of the implementation, and permutation of the rows and columns of the instance. To alleviate inconsistency of computational results due to performance variability issues, [8] suggest a *performance variability score* to present computational results on the MIPLIB 2010 instances. For more details regarding performance variability, the reader is referred to the recent survey article [10].

One way of exploiting performance variability in solving an instance is processing it with multiple different settings (called *configurations* throughout the rest of this paper) of an MILP solver and then selecting the best of these executions. In [11] commercial solvers such as CPLEX and Gurobi are executed multiple times with different random seeds (5 at the time of this writing) to solve a set of MILP benchmark instances and the best result is reported. The results show improvements in time to optimality relative to the default settings of these solvers (around 40% for CPLEX and 30% for Gurobi). In [4], a *bet-and-run* approach is proposed, where multiple configurations of an MILP solver obtained by different kinds of randomization of the root LP are run simultaneously until a given number of nodes is explored. Then a bet is placed on the "best" configuration, which is then run to optimality. The authors report a reduction in the number of branch-and-bound nodes. A similar approach is reported in the context of implementing branch and

* Corresponding author at: School of Business, Universidad Adolfo Ibáñez, Chile.
E-mail address: rocarvaj@gatech.edu (R. Carvajal).

http://dx.doi.org/10.1016/j.orl.2013.12.012
0167-6377/

bound in a parallel architecture [12,9,13]. A well-known issue here is the "ramp-up" phase of the algorithm in which not many branch-and-bound nodes have been generated, making it hard to balance work among available multiple processors and potentially leaving many idle resources. In [12] and then in [9,13] a technique (that is called *racing ramp-up* in [9]) is proposed, which involves running different configurations of an MILP solver in parallel independently across the available processors until a given criterion is met. Then the best-generated branch-and-bound tree is used to continue the algorithm by sending its nodes to the different processors. All other generated trees are discarded. According to [9], this approach has not proved to be very effective. In [3], the effect of variability in collecting good cutting planes and solutions within the CPLEX MILP solver is investigated. The approach starts by running a *sampling* phase that executes the default root-node cut loop starting from different optimal bases of the initial LP (through the use of the random seed parameter available in CPLEX) multiple times, while collecting cutting planes and feasible solutions. After this, a final run is done where the instance is solved by using the collected pools of cuts and solutions. They report considerable reduction in the variability of the percentage root node LP gap closed on a set of "unstable" instances from MIPLIB 2010. They also note improvements in primal solutions at the root node and reductions in the time of the final run of the algorithm (without taking into account the sampling phase).

In this paper, we study a possible way of exploiting performance variability by considering a diverse set of configurations of an MILP solver and executing these in parallel while allowing them to share information among each other. We test different types of information to be shared and compare the performance with that of the base solver with default settings. Our experimental results confirm previous evidence [4] that by simply selecting the best of multiple runs of an MILP solver with different configurations can yield significant performance improvements. We also show that the addition of communication yields substantial benefits in terms of reaching good feasible solutions or good upper bounds quickly. Although previous work (like [3,4]) has explored ideas that can be potentially used in a parallel setting, in most cases a serial implementation is used. However our approach that allows for communication on-the-fly and parallelism is a core part of our implementation.

In Section 2, we describe our diversification–communication framework. Section 3 describes the parallel implementation. Experimental results are given in Section 4 and conclusions in Section 5.

2. The diversification–communication framework

Our scheme consists of multiple configurations of an MILP solver running in parallel on the same instances. We call the set of different configurations a *diversification*. The configurations can share communication in different modes:

- *No communication*. Configurations run independently until one of them proves optimality or all of them run out of time.
- *Light communication*. Configurations run independently, but the best lower and upper bound obtained among all configurations are recorded. Thus optimality can be proved earlier by having one configuration providing the optimal solution and another the best dual bound.
- *Communication*. Configurations share some information with the other configurations, which then use the information in their own searches.

In the communication mode, the configurations can share one or more of the following types of information:

- *Feasible solutions*. Configurations send feasible solutions they find to the other configurations. Each configuration that receives a solution adds it as a heuristic solution to its search.
- *LP bounds*. Configurations send their best LP bound to other configurations. Each configuration uses the value to add an objective value cut of the form $c^T x \geq \bar{z}$, where $\bar{z}$ is a global lower bound for the problem.
- *Cuts at the root node*. The pool of cutting planes generated during the root node cut loop in each configuration is shared with the other configurations. When a configuration receives cutting planes from other configurations, it adds these as new rows to the problem. More details on how this is implemented are given in Section 3.

3. Parallel implementation and experimental framework

We implemented the diversification–communication framework described in the previous section by using a master–worker scheme, in which the master process is only in charge of managing communication among the workers, which are in turn running the different configurations of the MILP solver in parallel. Our code is written in C++ using OpenMPI [5] to implement the communication and CPLEX 12.4 [6] as the base MILP solver. All communications with the worker processes are done through the use of CPLEX callback functions implemented using its C callable library. All computations are done using the `boyle` cluster in the ISyE High Performance Computing Facility (http://www.isye.gatech.edu/computers/hpc/). The machines in this cluster are identical, each with 8 Intel Xeon 2.66 GHz chips and 8 GB RAM running Linux.

The sharing of cuts obtained at the root node by the different solvers is implemented using a two-phase approach. In phase one, a set of configurations with different emphases in generating cutting planes execute CPLEX's root node cut loop until it terminates or reaches a specified time limit. In phase two, the configurations share their corresponding cut pools and best solution found so far (if any).

After cut sharing is completed, we have a new instance which consists of the original problem plus the cuts just shared, added as new rows. Note that we have chosen to bypass CPLEX's cut filtering, since our intention is to force CPLEX to use the cuts and capture their effect on performance. Then, a different diversification is used, which we discuss below, and the optimization is restarted. Due to technical limitations of CPLEX, cuts cannot be collected in a form that corresponds to the original formulation unless we turn off the presolve feature in CPLEX.

Our experiments are performed over the *Benchmark* class of problems from MIPLIB 2010 [8]. We only use the feasible instances, which leaves 84 instances in total. All runs are given a time limit of 3 h of wall-clock time. When sharing cuts at the root node, a 1 h time limit is given to the root node cut loop and 3 h for the re-optimization phase.

One of our goals is to compare the performance of the proposed diversification–communication framework with default CPLEX using the same number of threads as processors that run in parallel. We limit our experiments to using 8 processes because of machine availability. Note that since our implementation uses a master–worker scheme, actually 9 processes are used, but one of them is just in charge of the communication. Also, due to the use of user callback functions in CPLEX in our implementation, dummy callbacks were used when CPLEX was executed with default settings. We think this is necessary in order to get a fair comparison, since the only presence of a user callback function changes the underlying behavior of CPLEX. In order to test the effect of communication, we fixed a diversification consisting of 8 configurations listed below. Each uses one thread, obtained by changing one parameter of CPLEX 12.4 at a time. The corresponding CPLEX parameter considered is indicated in parentheses.