Distributed Asynchronous Column Generation

S.Basso and A.Ceselli

Università degli Studi di Milano, Dipartimento di Informatica
{saverio.basso@unimi.it, alberto.ceselli@unimi.it}

Pre-Print Pre-Review Version for Institutional Use

Abstract

We propose a revision of the classical column generation algorithm for solving Dantzig-Wolfe decompositions of mixed integer programs. It is meant to fully exploit the availability of distributed computing resources, making optimization algorithms in general purpose solvers to scale better.

The main idea is to trigger massive parallelism by fully decoupling the computing flow of each component, including the resolution of the master problem, thus allowing different pricing algorithms to concurrently work on different sets of dual variables, and the master algorithm to asynchronously update dual information as soon as new columns are available.

Our algorithms ensure the same optimality convergency properties of the classical method. Experiments on mixed integer programs for three benchmark problems from the combinatorial optimization literature prove our approach to be one order of magnitude faster than state-of-the-art general purpose solvers in computing high quality root node dual bounds. Even if devised to exploit clusters of machines which do not share memory space, our algorithms show to be faster than earlier attempts from the literature also when run on virtual machines hosted on a single physical one, proving this improvement to derive from our algorithmic methodology rather than technological factors.

1 Introduction

Information systems are invaluable assets for a company. They currently provide the only practical mean of managing flows of data in business processes. A recent trend is that of virtualization, offloading information systems as cloud services. This yields major advantages, among which the option of performing on-demand provisioning of hardware resources like CPU cores, keeping them reactive also under load peaks [1].

Information systems represent also the entry point for diverse enterprise tools for the analysis of data, among which those performing quantitative decision support. Their complexity, and the need for flexibility, makes it often impractical to develop and deploy fully ad-hoc code. That is why general purpose solvers for mathematical programming problems [2, 3, 4] keep on increasing their penetration in industrial contexts.

However, the current architecture of general purpose solvers makes them hard to be adapted and scale well in an offloading environment. In fact, relying on branch-and-cut algorithms, they ask for centralized computations in a single powerful unit, while offloading grants computing power by the concurrent availability of several smaller units.

Decomposition methods provide a clear architectural advantage in this context. Recent attempts show that decomposition techniques like Dantzig-Wolfe ones can be applied automatically to some extent [5] [6] [8], thereby opening the road to general purpose solvers based on column generation and branch-and-price [7] instead of cutting planes and branch-and-cut.

Dantzig-Wolfe decomposition methods are expected to actually unfold potential when highly decomposable problems need to be optimized, and therefore massive parallelism can be potentially triggered. However, this intuition clashes with experimental evidence: standard parallelization methods typically fail. In fact, attempts from the literature often consider using limited parallelization, restricting such approach to the resolution of individual subproblems [9]. More effective methods have been proposed, which are however problem-specific [10] [11]. Finally, more elaborate implementations appear in the literature, which however are leveraging parallelism by dropping optimality guarantees [12] [13], actually becoming heuristics. At the same time, additional parallelization strategies have been recently studied for other decomposition methods, like bundle methods [14] [15] [16], although they focus mainly on distributed computation for cutting plane approaches.

In [17] we have introduced instead an Asynchronous Column Generation (ACG) mechanism. It is a concurrent version of column generation, that does not exploit the combinatorial structure of any specific problem, and guarantees the same optimality convergence properties of the classical method. It allows, thanks to asynchronous computing, almost linear speed-ups with respect to the available resources and much better performance than state-of-the-art solvers, on large scale instances. To the best of our knowledge, no further attempt to devise concurrent approaches to column generation for generic mathematical programs has been carried out in the literature.

In this paper we propose instead a distributed computing framework for column generation. It exploits massive concurrency when a *cluster* of machines is available by distributing calculations on multiple nodes in a smart way.

The main idea is not only to solve pricing problems in parallel, but to fully decouple the computing flow of each component, including the resolution of the master problem, thus allowing different pricing algorithms to concurrently work on different sets of dual variables, and the master algorithm to asynchronously update dual information as soon as new columns are available. We also discuss terminating conditions under which no optimality convergency property of the classical method is lost.

We remind the reader to Appendix A.1 for a review on the main concepts concerning parallel and distributed computing. Unfortunately, existing techniques are not easy to adapt in a distributed framework for two main reasons: (a) cluster machines do not share memory space, and therefore synchronization mechanisms among computing threads in different machines need to be specifically designed, and (b) cluster machines are often connected one another by network links which are orders of magnitude slower than CPU to RAM buses. On the contrary, techniques devised in a distributed setting can always

be adapted in shared memory (or even sequential) ones.

To achieve our goals, we focus on maximizing concurrent computation through the design of dedicated data structures, clean yet effective communication strategies. Part of our research is also a set of techniques to filter, rebalance and manage column pools, allowing concurrent column generation algorithm run faster; although we propose them in a distributed computing setting, these can be adapted in a sequential one as well.

We focus on the standard implementation, that is, we do not use all the machinery to speedup column generation, such as stabilization techniques [18], that have been developed during the last decades. Our approach, instead, allows to analyse in detail the impact of parallelization. However, we remark that these additional techniques can always be implemented on top of our version. Likewise, we focus on the column generation process for producing root node dual bounds, expecting reoptimization in inner nodes to be faster, and existing branch-and-bound techniques (even parallel ones) to be compatible with our approach.

We report a thorough experimental analysis where we compare our distributed algorithm against the asynchronous single machine version (ACG) proposed in [17] and state-of-the-art solvers.

The paper is organized as follows. In section 2 we review the main concepts of Dantzig-Wolfe decomposition for Integer Programs dual bounding, we introduce our notation, and we compare the standard column generation method with some parallelization options, highlighting the corresponding weaknesses and bottlenecks. In section 3 we introduce our asynchronous distributed computing framework for column generation and we compare its design to other parallel ones from the literature. Then, in section 4 we detail our implementation and parameters tuning. We then report its full experimental analysis. Finally, in section 5 we collect some conclusions.

2 Notation and Background

Dantzig-Wolfe decomposition was introduced in the sixties to cope with Linear Programming models not fitting into the very limited memory of computers. In the last two decades, however, its applicability range strongly expanded as researchers understood how to use it to obtain strong dual bounds for Mixed Integer Programs (MIPs), and in particular Integer Linear Programming (ILP) ones. In fact, let us assume to be given an ILP in the following form:

$$\min \sum_{j \in J} c^j x^j \tag{1}$$

$$\text{s.t.} \sum_{j \in J} B^j x^j \geq b \tag{2}$$

$$A^j x^j \geq a^j \qquad \forall j \in J \tag{3}$$

$$x^j \in X^j \qquad \forall j \in J \tag{4}$$

where x is the vector of variables, J is a partitioning of this vector, c^j are the objective function coefficients for each class in the partition, (2) is a set of constraints that potentially link different vectors x^j, (3) are $|J|$ sets of block

constraints, each $j \in J$ involving only variables x^j, and (4) define the domain of each variable (e.g. contain integrality constraints). That is, the constraint matrix has a single bordered block-diagonal form and constraints (2) correspond to the horizontal border. An additional vertical border can be defined as well, with a slight generalization, by considering an additional set of variables potentially belonging to all constraints but, for the sake of brevity, we disregard its notation. Similarly, minor additional notation (but no additional concept) is needed if the region described by (3) and (4) is unbounded. To ease exposition, we simply assume it to be a polytope.

Following the interpretation of [5] in terms of partial convexification, let θ^j be the convex hull of the extreme points of (3) and (4), that is, for each $j \in J$:

$$\theta^j = conv\{x | A^j x \geq a^j, x \in X^j\} \tag{5}$$

that is, when X^j encodes integrality conditions, θ^j represents the convex hull of extreme *integer* points of (3).

Let Ω^j be the set of indices of extreme points in θ^j and, for each p in Ω^j, let $\bar{x}^j_p$ be one of these extreme points. According to the Dantzig-Wolfe principle, we find a relaxation by replacing $x^j \in X^j$ with $x^j \in \mathbb{R}^{n_j}$ and by forcing for each $j \in J$:

$$x^j = \sum_{p \in \Omega^j} \bar{x}^j_p z^j_p$$

where z^j_p are new variables that act as multipliers to enforce that each x^j is a convex linear combination of the extreme points of θ^j.

DWD allows to disaggregate these problems in a master entity, involving only constraints (and variables) in the border, and one or more subproblems, one for each diagonal block, according to the following extended formulation:

$$\min \sum_{j \in J} \sum_{p \in \Omega^j} c^j \bar{x}^j_p z^j_p \tag{6}$$

$$\text{s.t.} \sum_{j \in J} \sum_{p \in \Omega^j} B^j \bar{x}^j_p z^j_p \geq b \tag{7}$$

$$\sum_{p \in \Omega^j} z^j_p = 1 \qquad \forall j \in J \tag{8}$$

$$0 \leq z^j_p \leq 1 \qquad \forall j \in J, \forall p \in \Omega^j \tag{9}$$

We report that an alternative interpretation of the Dantzig-Wolfe principle, leading to a similar (but not fully equivalent) integer counterpart of this extended formulation is exploited in research streams like [20].

2.1 Sequential and Parallel Column generation algorithms

To solve an extended formulation efficiently, state-of-the-art approaches adopt *column generation*. It consists in the following: (i) each set Ω^j is replaced by a restricted subset $\bar{\Omega}^j$ (ii) the restricted *master* program (RMP) that remains is optimized (iii) being a linear program, the optimization produces both a primal and a dual solution: the set of dual variables is collected, and for each $j \in J$, a search for the element of Ω^j corresponding to the column of *minimum reduced*

cost is carried out (iv) if any *negative* reduced cost column is found for some $j \in J$, the corresponding $\bar{\Omega}^j$ is updated, and the process iterates from (ii). On the contrary, if no new column is found, column generations is stopped, and the RMP solution is retained, being optimal for the full problem (that considers whole Ω^j) as well. The optimization problem that needs to be solved for finding minimum reduced cost columns (step (iii)) is called the *pricing* problem. Since only extreme integer points need to be considered, the pricing problem involves in general optimization over integer variables.

For each $j \in J$, we denote as π_j the following pricing subproblem

$$\min \{(c^j - \lambda)x_k^j - \mu_j, \text{ s.t. } x_k^j \in \Omega^j\}.$$

It needs to be solved at each column generation iteration. Vector λ refers to the dual variables of constraints (7), and vector μ to those of constraints (8).

Such a process can be performed for any ILP, but unfolds more potential if the original problem has a large value of $|J|$ and few constraints (7), showing to be a composition of smaller subproblems which are easier to be solved than a single large one during pricing.

Sequential column generation (CG). In a standard setting, like the one described in [7], pricing subproblems are solved in turn, sequentially, for each $j \in J$. Scholars have devised many techniques for speeding up this type of column generation algorithms. One of them is partial pricing: as soon as a negative reduced cost column is found while solving a certain pricing subproblem j, the control returns to the restricted master (step (ii)). We also mention the popular use of heuristic pricing procedures, in which each pricing subproblem is tentatively solved first by means of fast heuristics. Exact pricing is performed only when no negative reduced cost columns are found by heuristics. Both techniques share the intuition that solving all pricing problems to optimality might be needed only at the last column generation iteration, to prove optimality. The earlier one is in the column generation process, the more is worth to quickly populate the RMP with columns which are just good enough to lead the RMP dual solution into the right direction. Both techniques share also the same weakness: columns yielding only marginal changes to the RMP dual solution might be generated in a certain column generation iteration, while much better ones would have been found by either pricing subproblems which are skipped (in case of partial pricing) or by exactly solving some pricing subproblems (in case of heuristic pricing). Both techniques must therefore be fine tuned carefully, and such a tuning is in general problem dependent.

Synchronous parallel column generation (SCG). As fully detailed in [17], step (iii) can be easily run in parallel: the search for new columns for each $j \in J$ can be performed independently. In terms of parallel computing, a set of *disjoint* tasks is obtained by assigning a different execution thread to each subproblem π^j, where disjoint means that, working on different data, tasks do not need to communicate. They only need a single synchronization point, waiting all these tasks to terminate before joining execution back in a single thread. Therefore, as long as a column generation algorithm waits such a join before proceeding to step (iv) and possibly iterating, no additional synchronization structures are needed, and there are no changes to the standard algorithm (i) – (iv). Pricing

algorithms for different subproblems are even allowed to be run on *different machines*. This is indeed the common technique exploited in the literature (see e.g. [10]). Heuristic pricing is still a viable option. Partial pricing is instead a clear example about how even simple techniques which are standard in CG could partially clash in a concurrent setting: asking a subproblem to stop the others when a negative reduced cost is found requires tasks to communicate one another, at least with shared termination flag variables. That, however, makes tasks not disjoint anymore, thereby requiring to introduce synchronization mechanisms, allowing to simultaneously read and write these flag variables from a single task at a time. This in turn reduces parallelism potential. Any policy introducing a partial ordering between subproblems would similarly bring the algorithm back towards the sequential setting. In any case, synchronous column generation has a fundamental drawback: the time of the slowest subproblem π^j might strongly impact the time required for completing step (iii), that is, one subproblem may act as bottleneck when significantly slower than the others.

Asynchronous concurrent column generation (ACG). In [17] we have proposed to remove such a drawback and exploit massive concurrency. All subproblems $j \in J$ are initially run in parallel. As soon as a negative reduced cost column is found by *any* $j \in J$, and the corresponding set $\bar{\Omega}^j$ is updated, we start computing step (ii) and update the set of dual variables, as it would happen with a partial pricing approach. At the same time, the other subproblems are not stopped; on the contrary, *they are let free to complete their computation*. Therefore, this mitigates also the weakness of partial pricing: no column is lost because of early interruptions in the pricing sequence. Furthermore, such a setting allows the additional option of running heuristic and exact pricing algorithms in parallel, no one becoming bottleneck of the other, and still requiring no synchronization, thereby mitigating also the weakness of heuristic pricing.

3 Distributed algorithms

We remark that ACG allows to remove also the join point before step (iv), which is instead needed in SCG, thus further improving concurrency. In terms of computation structure, however, the direct result of our ACG approach is the following: different subproblems π^j, as well as different algorithms for the same subproblem, may be concurrently working on different sets of dual variables.

While intuitively simple, designing a suitable algorithmic framework for managing such a massive concurrent approach turned out to be non trivial in practice. We highlight two issues. The first issue is computational: to choose data structures and synchronization points in such a way that both synchronization overhead and idle time remain limited. The second issue is theoretical: to ensure that the convergence properties of column generation are kept.

An outline of our ACG algorithm is depicted in Figure 1. In our design, RMP and subproblems are assigned to concurrent tasks that update and solve their configuration as soon new information is available, and the only synchronization point is at the end of the algorithm, to check terminating conditions. The tasks communicate through a shared pool that contains the latest dual variables of the RMP and new columns, produced by subproblems. Since different tasks may be working on different versions of dual variables, we also store and use

an integer value, the *version stamp*, to keep track, for each task, of the latest dual variables that have considered for their last configuration. This is again necessary to guarantee optimality when checking terminating conditions. We avoid race conditions with fine tuned exclusive access. More in detail, after the RMP solve step, new dual variables are written in the pool. The RMP then awaits for new columns to be produced by subproblems. As soon one is available, the RMP gets updated and solved. At the same time, when a subproblem has finished computing a solution, it stores a new column in the pool and, if newer dual variables are available, its configuration gets updated and solved. Termination happens only when no new columns are available in the pool and all the version stamps of the tasks match, that is, the lastest dual variables produced by the RMP have been considered and processed by all the subproblems. An example of key steps of our ACG algorithm can be found in A.2, in the Appendix.

We refer to [17] for all the technical details and its experimental validation. In short, testing showed that ACG allowed consistent gains and almost linear speedups with respect to the number of CPU cores used.

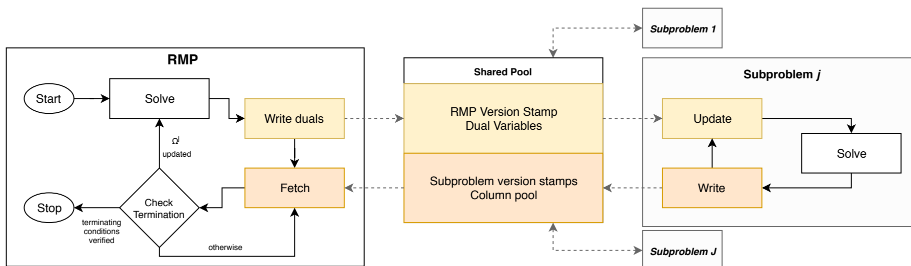


Figure 1: Asynchronous column generation algorithm

However, our ACG has a weakness blocking further scalability: it requires a shared pool for communication, with suitable locks on critical sections. Although this implementation is effective on a single machine, storing global data structures in memory requires a shared address space. Unfortunately, our attempts for directly mapping such a system design on computer clusters, produced poor results. In fact, emulating shared pools in a distributed memory environment introduced too much overhead.

3.1 A distributed asynchronous column generation algorithm

We propose a completely new architecture to exploit distributed computing. A general outline of our algorithm is reported in Figure 2.

We commit to the same asynchronous philosophy, considering however a scenario in which multiple machines are available for computation. We propose a distributed algorithm that greatly improves parallelism by decoupling the concurrent execution of the restricted master problem and the subproblems π^j on different computing units. In particular, we use the term *core* to define a single computing unit, and the term *node* to define pools of computing units: cores of the same node share memory space, while different nodes do not, and communicate instead by means of slower links. We design a master-client architecture

in which the RMP is assigned to a specific master node and the subproblems are split on the cores of the remaining available nodes. For every subproblem we create one task. Precise load optimization of the system is possible and can be fine tuned by manually balancing the number or the overall complexity of tasks assigned to specific nodes.

We devised communication by making use of a message passing paradigm and by following a centralized strategy: in our design, the master node controls the exchange of messages by deciding when to send data to or retrieve data from client nodes.

In the following we detail our Distributed concurrent Column Generation method (DCG).

3.1.1 Data structures

Similarly to the ACG we store, for each task, an integer value, that is the version stamp that represents the overall progress of algorithm. For the RMP, it indicates the overall number of LP optimization problems that have been solved. For each subproblem, instead, it represents the corresponding dual solution that it is being used to solve the current iteration of the specific subproblem task. Unlike ACG, however, we do not employ global data structures. Instead, each node stores its own *local* data. We detail data structures below.

Master node data structures The master node stores locally the following information:

- *RMP version stamp*
  the current version stamp of the RMP;
- *dual variables*
  a vector containing the values of dual variables obtained by the RMP at the end of the last optimization process;
- *subproblem version stamps*
  a collection of every subproblem version stamp.

Since the master node is dedicated for the RMP, no action is required to make data task safe.

Client nodes data structures Client nodes store instead a *local* pool that is shared among the assigned subproblems. On these nodes, we make use of an additional dedicated task, the *handler*, that is relegated to pool management and answering queries of the master node.

The *local pool* contains:

- *RMP version stamp*
  the latest version stamp of the RMP received from the master node;
- *dual variables*
  a vector containing the latest values of dual variables received from the master node;

- *solution pool*
  a collection of solutions found by subproblems on the client; each solution consists of a subproblem version stamp and a column vector.

Competition synchronization mechanisms are used to make the local pools task safe. Additional information about communication and termination are stored and managed by the RMP and each handler.

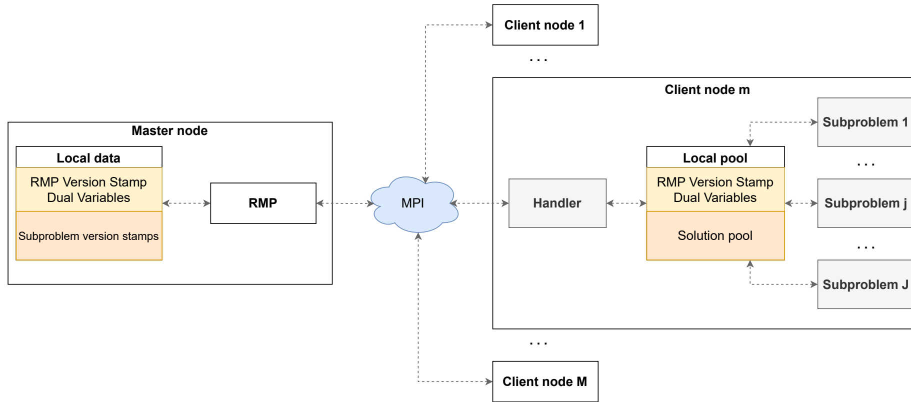


Figure 2: Distributed asynchronous column generation algorithm general outline

3.1.2 Algorithm outline

After initialization, the master node follows the subsequent steps:

Master node (RMP task).

1. `solve:` optimize the current RMP
2. `write duals:` check if the new dual solution is different from the previous one
   - if it is, (i) update the RMP version stamp and dual variables (ii) broadcast the dual solution and the stamp to every client node
   - Otherwise, broadcast an up-to-date signal

   Then, apply `Rebalancing strategies`.
3. `fetch:` check if new solutions can be collected from the clients; if at least one column can be fetched, update the respective local subproblem version stamps and apply `RMP update policy`.
4. `check termination:` if any subset $\bar{\Omega}^j$ is updated during `fetch`, broadcast a continue signal and goto `solve`; else check `RMP terminating conditions`: if they are verified, broadcast a termination signal and then **stop**, as global optimality is reached; otherwise broadcast a request signal and goto `fetch`.

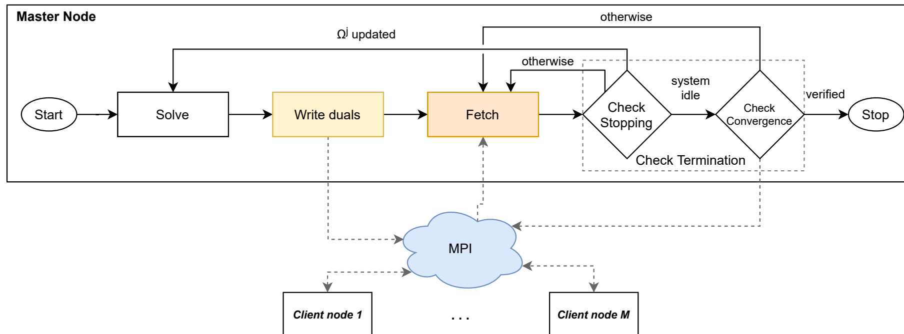


Figure 3: Distributed asynchronous column generation algorithm. Master node outline

A graphical outline for the master node is reported in Figure 3. In the following, we detail `Rebalancing strategies` and `RMP update policy`. We summarize instead `RMP terminating conditions` at the end of this section.

Rebalancing strategies Whenever the overall number of variables in the RMP configuration exceeds an upper limit *maxsols* we rebalance the system by eliminating unpromising columns to lower computing times for the RMP `solve` operation.

We order the RMP variables by non-decreasing reduced cost until a specified position *minsol*. All the variables that follow this position and have a worse reduced cost are discarded, that is, *minsol* acts as a lower limit to the overall number of variables in the RMP. We commit to rebalancing at the beginning of the algorithm.

RMP update policy When a new column is collected from client nodes we can apply one of the following policies:

- `Aggressive:` the corresponding $\bar{\Omega}^j$ set is always updated;
- `Conservative:` we recompute the reduced cost of the column with the latest dual variables. If it is still negative, we update $\bar{\Omega}^j$. Otherwise, we discard the column.

We commit to a strategy at the beginning of the algorithm. We do not change the update policy at runtime.

On the client nodes instead, subproblem tasks and the dedicated handler repeat, at the same time, the following operations:

Client node (Subproblem task).

1. `update:` check if the subproblem version stamp is different from the RMP version stamp stored in the local pool.
   - If they differ, store locally the new duals and set the local subproblem version stamp to be equal to the RMP version stamp in the local pool
   - Otherwise, wait for a suitable signal (from `handle broadcast`) into a sleeping state, and then repeat `update`
2. `solve:` optimize the subproblem with the local set of dual variables
3. `write:` save the solution into the local pool; goto `update`.

We remark that, during the `write` step, solutions are saved in the local pool independently of their reduced cost.

Client node (Handler task).

1. `handle broadcast:` wait for a new dual solution or an up-to-date signal from the master node (`write duals`); if a new solution has been received, (i) update the dual solution and the RMP version stamp in the local pool (ii) send an update signal to subproblem tasks
2. `handle fetch:` wait for a fetch request from the master node; when a request arrives, (i) send every solution in the pool to the master node (ii) clean the pool
3. `handle status:` wait for a status signal from the master node
   - if a continue signal is received, goto `handle broadcast`
   - if a request signal is received, goto `handle fetch`
   - if a termination signal is received, then **stop**

We show the processing flow outline for client nodes in Figure 4. Additional details on concurrency and on the subproblem task `solve` step are reported below.

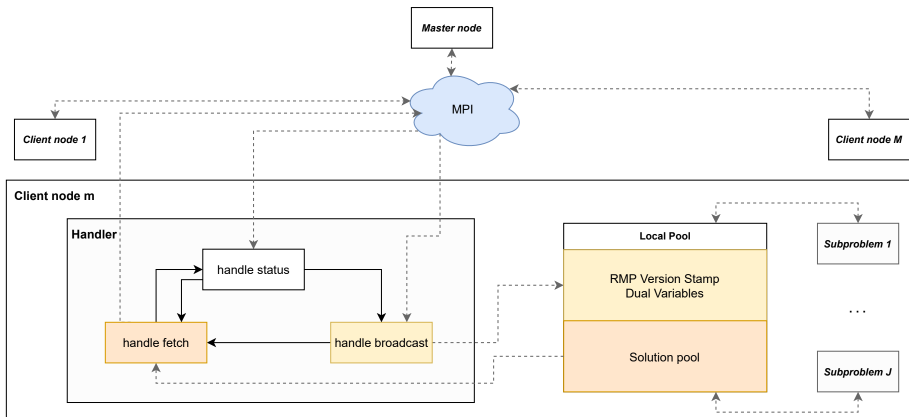


Figure 4: Distributed asynchronous column generation algorithm. Client node outline

Subproblems timelimit policy A global timelimit for the `solve` step can be specified for the subproblems. In this case, subproblems might terminate early their execution and normally proceed to the `write` and `update` steps. When this happens, they write in the local pool a dummy solution, that will be discarded, along with an integer timelimit flag to take note of the early stopping. This is necessary for terminating conditions. The advantage of this policy is twofold. First, we can improve meaningful computing: slower subproblems that are normally stuck on the `solve` step for a very long time might not be able to produce a column that is relevant for the current version of the RMP. Second, early interruption of subproblems allows to grasp the main advantages of asynchronous computation, that is, subproblems are able to fetch new versions of the dual variables from the RMP as fast as possible. This means that the algorithm, as whole, is expected to converge faster to the optimal solution.

We commit to a timelimit policy at the beginning of the algorithm.

Concurrency The local pool we use on each client node requires synchronization to manage race conditions. We use two mutex regions to limit reading and writing. The first one regulates access to the dual variables and the RMP version stamp. This region is locked by the handler during `handle broadcast` and by the subproblems during `update`. The second one is used instead for the solution pool: the handler locks it during `handle fetch`, whilst subproblems do during `write`. Differently from ACG, we do not partition the solution pool, that is, every subproblem task will write columns in the same data structure. Partitioning was quite useful to reduce overhead in ACG but in this version, due to the reduced number of subproblems assigned to each machine, the performance impact is negligible. Priority for updating is always given to the handler task. We also specify that the RMP presents an active wait between the `fetch` and `check termination` steps, like in ACG. However, since this time the task is running on a dedicated machine, there is no notable performance impact.

RMP terminating conditions Termination of the algorithm is managed by the RMP during the `check termination` operation.

1 `Check Stopping`: First, we make sure that no subproblem is running. When the version stamp of the latest columns collected from every subproblem is equal to the latest version stamp of the RMP, the system is idle.

2 `Check Convergence`: Then, we check if any subproblem had reached its timelimit during their last iteration. If they did, their computation is resumed with no timelimit. If at least one column with negative reduced cost is produced, the RMP follows the `solve` and `write duals` operations. If the update is successful, and new dual variables have been generated, `check termination` fails and the algorithm proceeds normally.

3 Otherwise, if the system is idle and every subproblem has fully completed computation, no further solution can be collected from client nodes. When this holds, termination conditions are satisfied.

We designed these conditions to preserve the guarantee of convergence to a global optimum.

Proposition 1. *Upon termination according to conditions 1 - 3, the DCG algorithm is guaranteed to achieve a global optimal solution of* (6) – (9).

Proof. We observe that the correctness of sequential column generation in yielding a global optimum relies on the following invariant: at each iteration either the RMP solution is optimal, or a possibly improving direction, that is a column, is produced. In other terms, the RMP may be seen as an oracle, producing a dual solution: a certificate of its optimality is given whenever no negative reduced cost column is found by the pricing subproblems, independently on which dual solution is produced by the oracle. The same invariant holds in our DCG. The RMP can be seen as an oracle, producing dual solutions, together with an associated version stamp: if all subproblems are run on that version stamp (condition 1), without producing new columns (as the system is idle), and each of them was solved optimally (condition 3), then the dual solution having that version stamp is optimal. Similarly, when early termination of some subproblems occurs (condition 2) which given the chance of terminating find no new column, we fall back to the previous argument (condition 3). A case remains to be considered, that is after being resumed, some subproblem finds new columns, but no change in the dual solution is observed in the subsequent RMP optimization. In that case, running the subproblems on the same dual solution would produce no new column, falling again back to the previous argument (condition 3). This last case might happen only in presence of degeneracy and aggressive RMP update policy.

At the same time, as in sequential column generation, if any improving direction exists with respect to a dual solution with a certain version stamp, it must be possible for some subproblem to find it when running on that version stamp.

□

4 Implementation and experimental analysis

We developed our Distributed Asynchronous Column Generation (DCG) algorithm in C++11, by using the Open MPI 3.0.1 library for distributed communi-

cation. Boost 1.66 was used as an MPI interface and for threads management.

We remark that our investigations are mainly driven by the perspective application of decomposition techniques in general purpose solvers. We therefore designed our code with a generic philosophy by using abstract data structures and inheritance to decouple the execution of the algorithm from the problem as much as possible, that is, the core of the algorithm is not built on a specific mathematical programming problem. For the same reason, in the main experiments reported in Section 4.2, each pricing problem was solved as a generic MIP. Indeed, speeding up our column generation algorithms by means of ad-hoc pricing algorithms is possible and promising: we evaluate the relative impact of such a choice, also reporting the full experimental results in Section A.6.2. We used CPLEX 12.6.3 with default settings to solve both the RMP LPs and the pricing problem MIPs. Single thread execution was set for each subproblem.

Architecture We employed the master-client architecture defined in section 3.1. When more than one machine was available, we reserved the dedicated master workstation for the RMP whilst we divided subproblems equally on the remaining client ones. We also considered an additional configuration where the Master machine was split in two virtual slots, and we assigned RMP to one slot and subproblems resolution to the other one. We made use of this scenario when RMP workload was light or for the single machine version.

Handling concurrency We implemented concurrent tasks for the helper and the subproblems as threads and critical sections in the shared pool of each machine were protected by lock objects. The handler of each client machine was given update priority through simple boolean flags and conditional variables were used for managing waiting times for subproblems. The RMP part was implemented as a sequential algorithm.

Communication Our implementation made use of a shared communicator for message exchange and synchronization between the master node and the client nodes. We considered two types of functions: collective functions such as *broadcast* are used to notify all client machines of new dual variables whilst *point to point communication* is used to query each client, in turn, for new columns. Serialization of data was necessary to send messages and was achieved with the boost library. We report that an attempt to employ multiple parallel point to point communication was done during an initial prototype phase. However, at the time of development, this OpenMPI feature was not fully tested and the system was not stable. We therefore discarded it.

Computing environment The experimental environment was setup as a beowulf cluster [22] up to 4 identical workstations running on Ubuntu 16.04 os, equipped with a quad-core Intel(R) Core(TM) i7-6700K 4.00GHz CPU and 32GB RAM. Each CPU is able to manage up to 8 physical threads, therefore the overall system grants resources for 32 cores and 128 GB or RAM. NFS directories were used for exchanging data.

Initial parameter tuning The starting configuration of parameters followed the setup used with ACG. That is, we stop the subproblems `solve` step at the

first solution with negative reduced cost and we use depth first exploration for the search trees to reduce memory footprint. After preliminary experiments rebalancing policies were fine tuned as follows. We chose 10000 columns for *minsol* and 20000 columns for *maxsol*. Initial timelimit for each subproblem was set to 10 minutes. Additional parameter tuning is detailed in subsection 4.2 in which we discuss the impact of the subproblem timeouts, rebalancing and RMP updating policies.

4.1 Test-bed instances

Designing a suitable test-bed needs special care. On one hand, our framework currently applies to problems having MIP formulations for which effective decomposition patterns are known, or can be detected by methods such as those of [5] or [8]. On the other hand, improving the state-of-the-art algorithms in specific combinatorial optimization problems is well beyond the scope of this paper.

To this end, we care of selecting diverse MIP conditions: both conditions in which formulations are known, making a standard general purpose solver perform well and conditions for which only weak compact formulations are known. The second key factor we take into account is how much MIPs are amenable for a massive decomposition approach in terms of number and hardness of pricing subproblems.

We therefore consider MIPs for three representative optimization problems for testing. The first one is the multi-dimensional variable size and cost bin-packing problem (MDVCSBP). It requires to pack optimally a set of items into different bin types with different capacities and costs. Its details, including the mathematical programming formulation, are reported in the Appendix, in section A.3. It represents a compromise benchmark: it allows for an effective decomposition approach but, at the same time, small and medium size instances can generally be solved easily by general purpose solvers through branch-and-cut. For testing we used 30 instances from the ones that we generated in [17], in particular, all the large ones with either 500 or 1000 objects and 250, 500 or 750 subproblems. The second problem we consider is a heterogeneous Vehicle Routing Problem with Time Windows (VRPTW). In this case, we have a fleet of vehicles, with different capacities and costs, whose routes must be designed to satisfy known demands of customers that can be visited only in a specified time interval at minimum cost. In section A.4 of the Appendix, we report the full mathematical model and its Dantzig-Wolfe reformulation. We tested our algorithm against 21 instances from the literature [21], with 30 customers and a fleet of maximum 25 vehicles of 5 possible types. The instances are split in three different classes (A, B and C) defined by different costs assignment strategies for the vehicles. The VRPTW is more challenging for pure generic approaches, as compact formulations are typically weak. The third problem is a Multi-Depot variant of our VRPTW (MDVRPTW), whose full mathematical model and Dantzig-Wolfe reformulation are reported in section A.5. It combines features of MDVCSBP and VRPTW: hard heterogeneous MIP subproblems, but high decomposition potential.

That is, these three problems represent three completely different working scenarios. The MDVCSBP is characterized by a massive number of subproblems with limited complexity. In fact, full tuning of parameters has been done on

MDVCSBP. The VRPTW, instead, has very few hard routing subproblems, one for each type of vehicle. Intuitively, our framework is better suited for MDVCSBP instances because of the sheer number of parallelism opportunities. In the VRPTW, instead, asynchronous computation is limited. Therefore it does not represent the normal working conditions for employing a distributed system. MDVRPTW is an intermediate case.

4.2 Experiments

In the following, we report the results of the experimental analysis of DCG. For comparison, we also included results for the state of the art general purpose solver CPLEX, asynchronous column generation (ACG) and the naive, synchronous implementation of classical column generation (SCG). CPLEX is used mainly as a reference term indicating the behaviour of branch-and-cut when large scale data is considered and more computing resources are available. SCG and ACG are instead the baseline and direct benchmark from the literature, respectively, for parallel column generation. To obtain a meaningful comparison, we always gave to CPLEX the same MIP which was used for the Dantzig-Wolfe decomposition considered in the corresponding SCG, ACG and DCG tests. Since pricing is also solved as a MIP, that simulates the working condition in which a MIP is submitted to a solver, and the solver decides to solve it either by parallel branch-and-cut or by parallel column generation. This experimental setting also weakens as much as possible the link between the underlying combinatorial optimization problem test and the results: what is mostly relevant is the structure of the starting MIP; from a qualitative point of view, both the branch-and-cut of CPLEX and the pricing problems of the CG would benefit (resp. suffer) better (resp. worse) formulations.

Benchmark algorithms were developed in C++11. We also analyzed the performance impact of the different policies that we have designed and, in the reminder, we refer to our algorithm as DCG_{opt} when we adopted the best policies. Otherwise, the default implementation with the same settings of ACG (that is, no rebalancing, aggressive RMP update policy and global subproblem timelimit policy) is called DCG_{plain}. Finally, we provide scalability tests as the number of nodes increases. In the experiments, we used a timelimit of 30 hours for solving each MDVCSBP instance and 10 hours for solving each VRPTW one. Computation of DCG and CPLEX was stopped at the root node of the branching tree in every experiment. For VRPTW instances, we split the master node in two virtual nodes. After preliminary experiments, we found these settings to give a good compromise between interpretability of results and overall simulation time.

4.2.1 Optimizing DCG performance

In the first round of experiments we estimated the impact of our column management, rebalancing and subproblems timelimit policies.
In Table 1 we report the relative speedup of DCG over 2, 3 and 4 nodes with respect to a single workstation. We present the results when solving MDVCSBP instances under aggressive and conservative update policies as the average wall time ratio for each number of objects n and each number of bin types $|J|$. Higher values mean better performance.

n	$\lvert J\rvert$	Aggressive			Conservative		
		2	3	4	2	3	4
500	250	3.28	2.95	2.34	3.83	4.09	3.97
	500	3.63	3.02	2.41	3.45	4.96	5.34
	750	2.54	2.58	1.90	2.67	3.73	3.81
1000	250	3.20	2.22	2.00	3.61	3.69	3.63
	500	4.11	2.68	1.96	4.85	4.67	4.53
	750	6.14	4.25	2.75	5.50	6.79	6.67
Overall		3.82	2.95	2.23	3.99	4.66	4.66

Table 1: DCG speed-up profiling over 2, 3 and 4 machines when solving MDVCSBP under aggressive and conservative update policies

Results show that DCG almost attains 4 times the speedup when two workstations are used instead of one. However, under the aggressive strategy performance deteriorates when increasing the size of the cluster. This behavior does not happen with a conservative policy as limited speedups can be observed up to 3 machines and further increasing the cluster size does not have a negative impact. Generally, when more workstations are available, subproblems are spread among multiple nodes and more columns can be generated at the same time, possibly based on the same set of dual variables. In particular, when using an aggressive policy, the size of $\bar{\Omega}^j$ grows faster, the `solve` step for RMP takes longer and subproblems are updated less frequently to new dual variables. These results show that a conservative management is beneficial to the algorithm as any kind of negative impact on the frequency of generation of new dual variables greatly undermines the advantages of asynchronous computation.

In Table 2 we report results when solving MDVCSBP on 2 machines under a conservative policy, active rebalancing and using different subproblem timelimits. We consider the following configurations: 600 seconds (like ACG), 30 seconds and 1 second. We present the relative average wall clock time ratio for each n and each $|J|$ when compared to DCG on 2 machines with 600 seconds subproblem timelimit and no rebalancing. Higher values mean better performance.

n	$\lvert J\rvert$	600 [s]	30 [s]	1 [s]
500	250	1.12	1.86	6.19
	500	1.03	1.47	1.55
	750	0.97	1.54	2.22
1000	250	1.07	1.33	2.53
	500	1.22	1.41	4.39
	750	1.18	1.39	1.34
Overall		1.10	1.50	3.04

Table 2: DCG speed-up profiling on 2 machines over subproblem timelimits when solving MDVCSBP with conservative updates and rebalancing

When facing identical subproblem timelimits, our rebalancing strategy provides around a 10% boost in performance. Higher improvements can be expected over longer runs when more subproblems or more machines are present. However, the largest impact on solution time is provided by the timelimit policy for subproblems. In fact, when we move from a 600 seconds to a 1 second timelimit the algorithm finishes computation 3 times faster. This is due to asynchronous computation. When a very short timelimit is enforced, the algorithm saves only the columns that are generated in that specific time frame. This means, that slower subproblems stop computation immediately and start from scratch with the newest dual variables. Unpromising columns are discarded and we move to the generation of more favorable ones. This behavior speeds up the convergence of the algorithm. However, further experiments are needed to fully evaluate the impact of this pricing strategy on parallelization.
To summarize, when DCG is tuned with the best policies (DCG_{opt}) we can observe a big performance improvement. This includes conservative solution management, active rebalancing and short subproblem timelimits. For MDVCSBP we used a 1 second timelimit. For VRPTW, after preliminary experiments, we chose a 60 seconds subproblem timelimit.

4.2.2 Comparison against Asynchronous Column Generation

In our second round of experiments we studied how our DCG algorithm performs with respect to the asynchronous column generation. To this end, we compared CPLEX, DCG with default parameters (DCG_{plain}) and DCG with optimized policies (DCG_{opt}) with respect to ACG when solving MDVCSBP. We omit a direct comparison with SCG, since the superiority of ACG over SCG has been extensively discussed in [17].

To make this comparison fair, distributed algorithms run on a single machine through virtualization, that is, we create a scenario in which each algorithm can exploit the same amount of computing resources. We report the relative average wall clock time ratio (Spd-up) and the total number of timeouts for each n and each $|J|$ in Table 3. The speed up averages take into account also the tests that have reached timeout. For speed up, higher values mean better behavior.

		CPLEX		DCG_{plain}		DCG_{opt}	
	$\|J\|$	Spd-up	Timeouts	Spd-up	Timeouts	Spd-up	Timeouts
500	250	0.10	0	0.76	0	1.86	0
	500	0.12	1	0.91	0	1.78	0
	750	0.04	1	0.91	0	1.55	0
1000	250	0.19	0	0.99	0	2.42	0
	500	0.48	3	0.89	1	2.62	0
	750	0.41	5	0.93	0	3.11	0
Overall		0.22	10	0.90	1	2.22	0

Table 3: CPLEX and DCG pre optimization (DCG_{plain}) and post optimization (DCG_{opt}) compared to ACG, on a single machine, when solving MDVCSBP

When facing such big instances CPLEX is simply not competitive and, on the largest ones, systematically hits timeouts on all of them ($|J| = 750$, n = 1000). DCG_{plain} retains most of the speed-up of asynchronous column generation,

achieving about 90% of its efficiency. We speculate that the missing 10% is lost because of the communication and virtualization overhead that we meet when using a single machine. However, when we employ DCG with optimized policies we obtain a noticeable performance boost even on a single node. In fact, results show that DCG_{opt} is on average more than twice as fast as ACG, providing up to three times the performance on the largest problems. Furthermore, DCG_{opt} is the only algorithm that finishes all computations before hitting a single timeout.

From these results a question arises, if the improvement of DCG is given entirely by the additional local columns pool management policies. To investigate this, in Table 4 we compare the total computing time for solving MDVCSBP, when using the same number of CPU cores, over 1 or 2 machines (Configuration), with ACG, DCG_{plain} and DCG_{opt}. That is, all configurations share the same amount of CPU resources.

Algorithm	Configuration	Total Time
ACG	1 node, 8 cores	522387.15
DCG_{plain}	1 node, 8 cores	600764.10
	2 nodes, 4 cores each	124740.00
DCG_{opt}	1 node, 8 cores	209852.18
	2 nodes, 4 cores each	68522.97

Table 4: ACG, DCG pre optimization (DCG_{plain}) and post optimization (DCG_{opt}) total solving time, for MDVCSBP instances, when using the same number of cores on 1 and 2 nodes.

The comparison between the two configurations of DCG_{plain} allows to understand the contribution given by the distributed architecture alone. It can be noticed that the total time decreases sharply when moving from 1 node to 2 nodes, even if the total number of available cores remains the same. This is also true for DCG_{opt}. The comparison between the rows of DCG_{plain} and DCG_{opt} on identical configurations allows instead to evaluate the contribution given by the new policies alone, which is significant as well. We also notice that even the distributed architecture alone allows DCG_{plain} to greatly outperform ACG when moving to 2 nodes, while keeping the same number of cores. The same happens comparing DCG_{opt} and ACG on a single node, due to the new policies.

Finally, in Table 5, we study how ACG and DCG_{opt} perform from a convergence point of view. For each algorithm and each n and $|J|$, we report the average number of iterations (IT) of the RMP and the average total number of generated variables (Var.). For DCG, we report results when run on 2 nodes.

This Table shows how DCG_{opt} converges to the solution in about the same number of iterations of ACG. It would be easy to assume that a performance boost could be obtained anyway by brute forcing parallel computing in a distributed system. However, our architecture works more efficiently. In fact, the average total number of variables that we generate in our runs are one order of magnitude less than ACG. We argue that, thanks to our design and policy changes, this new algorithm is capable of converging with less but more meaningful columns that are generated faster. This holds true even when using a single machine.

	$\|J\|$	ACG		DCG_{opt}	
		IT	Var.	IT	Var.
	250	1751.20	81408.80	1826.80	22070.40
500	500	2829.00	143325.00	4294.80	32112.60
	750	1918.40	145395.80	2427.60	31155.60
	250	3530.80	152800.80	3589.80	49585.60
1000	500	4154.40	248396.60	4685.80	72420.60
	750	4466.60	350803.20	4261.20	81364.20
Overall		3108.40	187021.70	3514.33	48118.17

Table 5: Convergence comparison between ACG and DCG_{opt} on two machines when solving MDVCSBP

4.2.3 DCG performance profiling

In our third round of experiments, we study how DCG performs when the available resources increase, that is when there are more machines for subproblems computation.

MDVCSBP. In Table 6 we report the average total wall clock time (Time) of DCG_{opt} when solving MDVCSBP over 1, 2, 3 and 4 machines. We also present the speed-up (Spd-up) as the average ratio between the wall clock time of 2, 3 and 4 workstations and the wall clock time of one workstation. For completeness, the rows marked as 'Overall' report the average wall clock times and the average speedup over the instances of the corresponding block.. For time lower scores are better whilst for the speed-up higher values mean better performance.

		1	2		3		4	
n	$\|J\|$	Time	Spd-up	Time	Spd-up	Time	Spd-up	Time
	250	923.29	3.52	262.86	3.80	242.94	3.79	242.62
500	500	4015.91	3.05	1377.78	4.57	843.63	4.73	806.38
	750	1961.90	3.24	611.18	4.80	408.33	4.66	418.20
Overall		2300.36	3.27	750.61	4.39	498.30	4.39	489.07
	250	5167.02	2.13	2488.58	2.28	2327.75	2.10	2492.41
1000	500	15998.30	2.73	5602.79	2.57	6113.38	2.38	7912.23
	750	13904.03	2.79	5032.92	2.77	5132.72	2.72	5309.86
Overall		11689.78	2.55	4374.76	2.54	4524.62	2.40	5238.17
Overall		6995.07	2.91	2562.69	3.47	2511.46	3.39	2863.62

Table 6: DCG_{opt} speed-up profiling over 1, 2, 3 and 4 machines when solving MDVCSBP

When $n = 500$, DCG_{opt} attains consistent performance boosts up to 3 machines and, in these particular instances, more than linear speed-ups. However, the algorithm hits some limitations when 4 workstations are considered: this is consistent in all the testing. We also observe that DCG has better speed-ups when subproblems are easier. In fact, when we increase the number of objects n from 500 to 1000 we lose, in some instances, a bit of efficiency, even when using

3 machines. We conjecture that this behavior is a consequence of asynchronous computation, that is, to sustain high speed the algorithm needs to produce very quick solutions for both the RMP and the subproblems. When one gets slower asynchronous computation is less effective and the system slows down. Summarizing, when facing problems that are well suited for distributed computing decoupling RMP and subproblems on different machines grants very big improvements. Further scaling with additional workstations might be problem and instance dependent.

VRPTW. In Table 7 we consider DCG_{opt} performance when facing VRPTW instances. As reported before, this is a stress test for the algorithm because of the limited number of subproblems: asynchronous computation can hardly be exploited optimally. We present, for each instance class, the average total wall clock time when using from 1 to 4 workstations and the relative improvement (Spd-up) with respect to one machine. For speed-ups higher scores are better, for time lower values mean better results. Full results can be found in Table 9 in the Appendix.

	1	2		3		4	
Class	Time	Spd-up	Time	Spd-up	Time	Spd-up	Time
A	314.06	0.93	372.85	1.90	266.49	1.05	337.77
B	1104.93	1.10	877.65	1.10	893.59	1.42	727.98
C	1765.78	1.28	1312.32	1.32	1497.56	1.22	1486.20
Overall	1061.59	1.10	854.28	1.16	885.88	1.23	850.65

Table 7: DCG_{opt} speed-up profiling over 1, 2, 3 and 4 machines when solving VRPTW

Scalability is not as strong as with MDVCSBP instances. On average, DCG_{opt} shows small improvements up to 4 machines. Differently from before, having RMP and subproblems running on different nodes is not enough to provide a very noticeable boost. We also notice that scaling is not consistent, that is, there are class of problems in which 3 machines work better than 4 and vice versa.

In an effort for better understanding this behaviour we measured the realtive time spent by the RMP task in actually solving LPs, and that spent in waiting for new columns to be produced by pricing subproblems. For detailed results we refer to [19]. In synthesis, on MDVCSBP instances less than 1% of the time is spent in waiting; on VRPTW instances the opposite happens: less than 1% of the time is spent in LP solving. Furthermore, for every VRPTW instance, the overall number of generated variables is always under 1000. These results are indeed in line with intuition from sequential and even synchronous column generation.

Under these conditions (few, slow subproblems), asynchronous computation is heavily undermined. In general, parallelization techniques have less potential in this setting, as pricing-intensive computations can be saved only by exploiting runtime information from one pricing problem to another, which cannot easily be done if pricing computing threads are disjoint. Even in this worst case scenario

with very few subproblems, our algorithm still provides benefits and a decent boost in performance. We suspect that even better results could be obtained by considering also auxiliary heuristic subproblems to generate columns quickly.

One may argue that these limited speed-ups arise from the use of MIP solvers for pricing in a VRP context, which is not the common practice. We have therefore performed another experiment, implementing a dynamic programming algorithm for the elementary resource constrained shortest path pricing problem. Our results are reported in Appendix (Table 10). As expected, the running time strongly decreases, however the speed-up values remain similar, proving our overall analysis to be robust in this regard. That also suggests that our methods are indeed complementary to existing ones on specific problems, providing similar improvement, and the additional gain of distributed computing.

MDVRPTW. In Table 8 we report our results on the MDVRPTW in terms of average wall clock time (Time) for 1, 2 and 4 machines for each class of problems. We also report the average speed-up (Spd-up). Full results can be found in Table 11 in the Appendix. We remark that our MDVRPRTW differs from the VRPTW of the previous section mainly by the number of pricing subproblems (5 on the VRPTW, 155 on the MDVRPTW).

	1	2		4	
Class	Time	Spd-up	Time	Spd-up	Time
A	1138.89	2.19	523.17	4.76	232.32
B	3964.75	2.11	1952.04	4.63	684.83
C	3531.35	2.56	1085.80	4.67	645.54
Overall	2798.58	2.27	1168.25	4.69	507.12

Table 8: DCG_{opt} speed-up profiling over 1, 2 and 4 machines when solving MDVRPTW

Differently from the VRPTW experiments, which presented limited scalability over a small number of subproblems, DCG_{opt} scales more than linearly with the number of nodes, obtaining results that are similar (but even better) than with MDVCSBP. Indeed, this confirms the potential of our architecture when solving large scale instances of highly decomposable problems: when facing a large number of subproblems, our algorithm is capable of exploiting all the available resources, providing great benefits when moving from one machine to a cluster of nodes. Additionally, these results confirm that our architecture can scale well even when facing formulations with complex pricing problems.

4.2.4 Comparison with cutting planes.

To conclude our analysis, we evaluate the potential of using Dantzig-Wolfe decomposition and our DCG algorithms to solve highly decomposable MIPs, with respect to state-of-the-art branch-and-cut approaches. In particular, since branch-and-cut does not provide a straightforward way to exploit distributed computing, our aim is to understand if we can expect to get any farther by changing the resolution paradigm, using our methods. We therefore compared the overall performance of DCG_{opt} with CPLEX, when both are used to get

dual bounds at the root node of a branch-and-bound tree. Our full results are reported in Table 12 of the Appendix. We remark that these results need to be seen in the light of solving specific MIPs (at the root node) with general purpose algorithms, which is different than choosing best performing methods and formulations for the corresponding combinatorial optimization problems. We also stress that cutting planes (as CPLEX) and Dantzig-Wolfe decomposition (as in our algorithms) produce in general different bounds and are therefore difficult to compare. However, in our test-bed, CPLEX and our algorithms experimentally produced bound values remarkably similar. That is, Dantzig-Wolfe decomposition and generic cutting planes tend to produce, respectively by inner and outer representation, similar partial convexifications.

When massive parallelism can be exploited, like with MDVCSBP instances, the performance gains we get are massive. DCG_{opt} completes the experiments on average 87 times faster than CPLEX and, in some instances, the algorithm is hundreds times better. We also report that CPLEX cannot complete computation before the timelimit in 10 instances. In these cases, computing times are likely much greater. Even when we observe results for VRPTW, DCG_{opt} performs very well, always completing the root node dual bounding phase, whilst CPLEX at the root node goes systematically out of memory in 19 instances and reaches the timelimit in the remaining 2.

A graphical representation of the performance of the algorithms is also reported in Figure 5, where we present the average computing time in seconds on the y-axis and the number of subproblems on the x-axis. We report results for CPLEX, ACG and DCG_{opt} (with 3 machines). As a benchmark, we also report the results for SCG, that is the classical implementation of parallel column generation. SCG needs to be considered more as a reference than as a benchmark, having the structural drawbacks highlighted in section 2.1. We consider only MDVCSBP instances, since for this problem the compact MIP used with CPLEX is known to perform well.

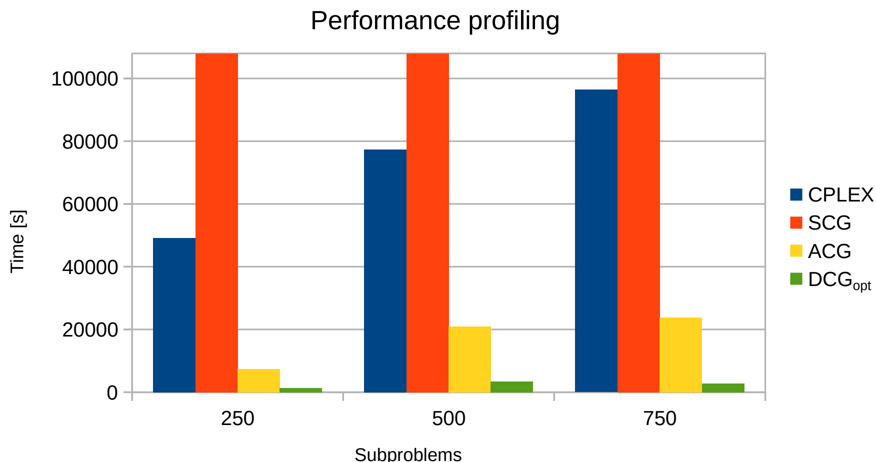


Figure 5: Computing time for MDVCSBP instances over 250, 500 and 750 subproblems

As reported, on large scale instances the performance gains with respect to CPLEX are massive. However, it is also notable that ACG and DCG are the only configurations that scale very well when the number of subproblems increases. Surprisingly, DCG performs even better with 750 subproblems, rather than 500. This is a strong indicator of scalability, and suggests that even in this scenario, we are still far from saturating all the computational capabilities of our architecture. Results also confirm that asynchronous computation has a huge impact on the performance column generation, since the synchronous version hits timeout on all the instances.

5 Conclusions

In this paper we proposed a restructuring of classical column generation algorithms which takes full advantage of distributed computing. Our design and architectural choices allow to benefit of massive performance boosts when facing large scale decomposable problems. Our framework proves potential to tackle complex problems with large scale data, with a completely generic approach, while providing improvements in memory management and very fast computing times by exploiting effectively distributed computation.

The most clear improvement with respect to earlier attempts as [17] is structural: we are able to exploit clusters of machines which do not share memory space, as opposed to state-of-the-art approaches like [17] which instead assume tasks to always run on the same physical machine. Additionally, our experiments highlight improvements also when compared on a *single* physical machine, proving them to be due to a better architecture and additional algorithmic techniques.

When compared to state-of-the-art branch-and-cut solvers, when attempting to solve the same MIPs, our algorithms produce similar bounds, and are systematically able to produce tight dual bounds on instances where CPLEX at the root node either goes timeout, or runs out of memory. On MIPs for a MDVCSBP, our algorithms run one order of magnitude faster.

In particular, our experiments highlight how maximizing asynchronous computation is fundamental for faster convergence. Logically splitting the optimization of the master problem from the other components appears to be a key step. Our smart filtering and rebalancing strategies, which are based more on properties of the optimization process than on simple computing loads, play also a sensible role in producing an effective framework.

In perspective, studying how generation of supplementary heuristic columns impact computations looks promising. In particular, for instances that present a limited number of hard pricing subproblems, the concurrent run of different heuristics for the same subproblem might yield further improvements.

Acknowledgments

The authors wish to thank three anonymous reviewers, whose insightful comments helped to improve the paper. The work has been partially funded by University of Milan, Piano Sostegno alla Ricerca.

References

[1] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, J.P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey", Journal of Systems Architecture 98: 289-330 (2019).

[2] IBM Cplex webpage: https://www.ibm.com/analytics/cplex-optimizer (last access Nov. 2020)

[3] GUROBI webpage: http://www.gurobi.com (last access Nov. 2020)

[4] FICO xpress webpage: http://www.fico.com/en/products/fico-xpress-optimization-suite (last access Nov. 2020)

[5] M. Bergner, A. Caprara, A. Ceselli, F. Furini, M. Lübbecke, E. Malaguti, E. Traversi "Automatic Dantzig—Wolfe reformulation of mixed integer programs", Mathematical Programming A, 149(1-2): 391–424 (2015)

[6] S. Basso, A. Ceselli "Computational evaluation of ranking models in an automatic decomposition framework", Proc. of EURO/ALIO 2018, Electronic Notes in Discrete Mathematics, Volume 69: 245-252 (2018)

[7] G. Desaulniers, J. Desrosiers, M.M. Solomon (Eds.) "Column Generation", Springer, Berlin (2005)

[8] S. Basso, A. Ceselli "Computational Evaluation of Data Driven Local Search for MIP Decompositions", Proc. of ODS 2019, Advances in Optimization and Decision Science for Society, Services and Enterprises. AIRO Springer Series, vol 3: 207-217 (2019)

[9] O. Tombus, T. Bilgic "A column generation approach to the coalition formation problem in multi-agent systems", Computers and Operations Research 31: 1635 –1653 (2004)

[10] R. Martinelli, C. Contardo "Exact and heuristic algorithms for capacitated vehicle routing problems with quadratic costs structure", INFORMS Journal on Computing 27: 658-676 (2015)

[11] T. Wang, B. Jaumard, C. Develder "Anycast (re)routing of multi-period traffic in dimensioning resilient backbone networks for multi-site data centers", proceedings of the 18th International Conference on Transparent Optical Networks (ICTON) (2016)

[12] D.W. Casbeer, R.W. Holsapple "A Column Generation Approach to the Vehicle Routing Problem", AIAA Infotech@Aerospace (2010)

[13] V. Huart, S. Perron, G. Caporossi, C. Duhamel "A Column Generation Heuristic for the Time-Dependent Vehicle Routing Problem with Time Windows", in "Computational Management Science", Lecture Notes in Economics and Mathematical Systems 682: 73-78 (2016)

[14] F. Fischer "Dynamic Graph Generation and an Asynchronous Parallel Bundle Method Motivated by Train Timetabling", PhD thesis, Chemnitz University of Technology (2013)

[15] K. Kim, C.G. Petra , V.M. Zavala "An Asynchronous Bundle-Trust-Region Method for Dual Decomposition of Stochastic Mixed-Integer Programming", SIAM Journal on Optimization 29(1): 318–342 (2019)

[16] F. Iutzeler, J. Malick, W. de Oliveira "Asynchronous level bundle methods", Mathematical Programming 184: 319–348 (2020)

[17] S. Basso, A. Ceselli "Asynchronous Column Generation", Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX), 197-206 (2017)

[18] A. Pessoa, R. Sadykov, E. Uchoa F. Vanderbeck "In-Out Separation and Column Generation Stabilization by Dual Price Smoothing", proc. of SEA 2013. LNCS 7933 (2013)

[19] S. Basso "Data driven algorithms and distributed computing for automatic MIP decompositions", PhD Thesis - University of Milan (2021)

[20] F. Vanderbeck and M.W.P. Savelsbergh "A generic view of Dantzig–Wolfe decomposition in mixed integer programming", Operations Research Letters 34(3); 296-306 (2006)

[21] R. Baldacci, M. Battarra, D. Vigo "Routing a heterogeneous fleet of vehicles", Operations Research/ Computer Science Interfaces Series, 43, 3-27 (2008)

[22] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, C.V. Packer, "BEOWULF: A parallel workstation for scientific computation", Proceedings, International Conference on Parallel Processing 95, 11-14 (1995)

A Appendix

A.1 Parallel and distributed computing concepts

Numerical resolution of ILPs requires to intertwine algorithm design and algorithm *engineering.* To clarify our approach we recall the following key architectural concepts.

- Algorithms are composed by *statements.*
- In a *sequential computing* setting statements are to be executed one after another; the execution order in the computer implementation is determined by a single *thread* of control flow (i.e. a sequence of instructions executed on the same CPU).
- A *parallel computing* setting is obtained by allowing *more than one* computing thread, each potentially running on a different CPU simultaneously. That is, *more than a single statement might be executed simultaneously.*
- The ratio between the computing time of an algorithm parallel implementation and the sequential one is called *speedup.*
- When statements in different computing threads *do not share data*, these threads are said to be *disjoint.* It is a very desirable property: since disjoint threads do not interfere one another, their overall computing time corresponds only to the computing time of the longest one (instead of their sum), thus yielding high speedup. Theoretically, a *speedup which is linear in the number of available physical resources* is the best one can expect, as it corresponds to all the computing being disjoint and perfectly split and balanced.
- Perfect parallelism can be obtained only if an algorithm can be fully decomposed, which is normally not possible: different threads need to *share data*; when threads are able to *share system memory*, sharing data means to have variables for which different threads need to write and read.
- Simultaneous access to shared variables by different threads may lead to inconsistencies; therefore, threads which are not disjoint need to be *synchronized* (only) when performing these operations.
- Synchronization is obtained by enforcing *mutual exclusion (mutex)* regions in computer codes where shared variables are used.
- A few paradigms exist in the design of such regions, according to the semantics of the access. The most common one is the *competition synchronization* need, where a mutex is created to prevent multiple threads to write in the same variable concurrently.
- The more synchronization needs, the less parallelism can be exploited. A lack of synchronization leads to a potentially (unwanted) non-deterministic behaviour, known as *race condition.* An excess of synchronization leads to threads waiting other threads to operate, ultimately requiring CPUs to simply wait in a so-called *idle state*, and thereby lowering the speedup.

- By proper synchronization a computer code is made *thread safe*, meaning it allows multiple thread execution without race conditions.

- The CPUs where threads run are meant as *logical cores*. That is, parallel algorithms are designed to work independently on the number of actual CPUs in the system. Then, a specific run of an algorithm implementation exploits a number of *physical cores*, that are physical CPUs (or CPU portions) allowing to actually host simultaneous control flows. The number of logical and physical threads is therefore *not required to match* in well designed parallel implementations. Still, some overhead is expected when the number of logical threads greatly exceeds that of physical ones, as physical CPUs need to simulate the executing of logical ones by *switching* from one logical thread to another according to some policy, saving and retrieving the so-called *context* of each logical thread at each switch.

- A *distributed computing* setting is obtained when CPUs of machines which *do not share memory space* are employed to run computing threads. These are normally a so-called *cluster* of computers connected one another by network links, but can also be virtual machines hosted on the same computer. Since no memory space is shared among different machines, distributed algorithms need further design, as the synchronization of threads running on CPUs of different computers *(a) cannot rely on shared variables (b) needs to rely on alternative communication methods such as messages on (potentially slow) networks.* It is therefore of paramount importance to keep synchronization and communication data as low as possible.

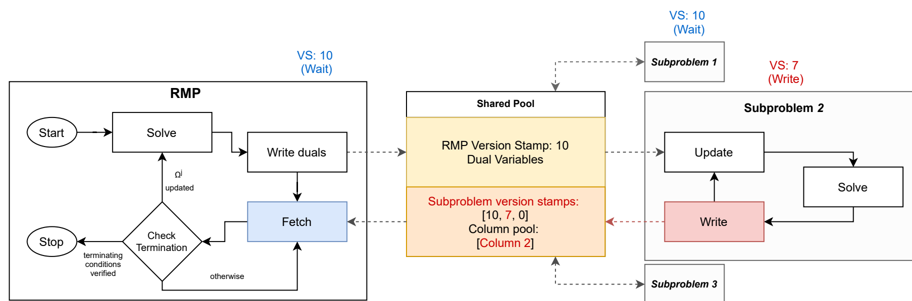
- *Broadcasting* refers to the operation of one node to transmit the same message to all other connected nodes. If the message is data with involved structure (e.g. a collection of values) they need to be *serialized* to be transmitted from one node to another, that is transmitted one component after another, byte by byte, in a specific sequence which is known to both the sender and the receiver. Serialization and broadcasting can indeed be considered time consuming operations in a high-performance computing setting.

A.2 ACG: example

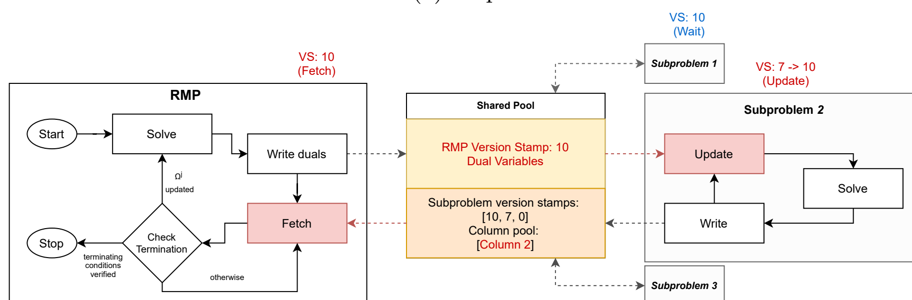
We report in Figure 6 an example with 4 steps of the ACG algorithm, with a small setup that includes only 3 subproblems. Additional information about the version stamp (VS) of the dual variables taken into account by each task is also reported in the figure.

We focus on the following scenario: the RMP has finished computing iteration 10 and is waiting to collect, in the `fetch` step, new columns in the shared pool. Subproblem 1 has finished solving a configuration with the latest iteration of the dual variables, and it is waiting for a new update. Subproblem 2 is currently solving VS 7 whilst Subproblem 3 is still computing VS 1. We remark that in a sequential or synchronous column generation implementation, the whole system would still be at the first iteration.

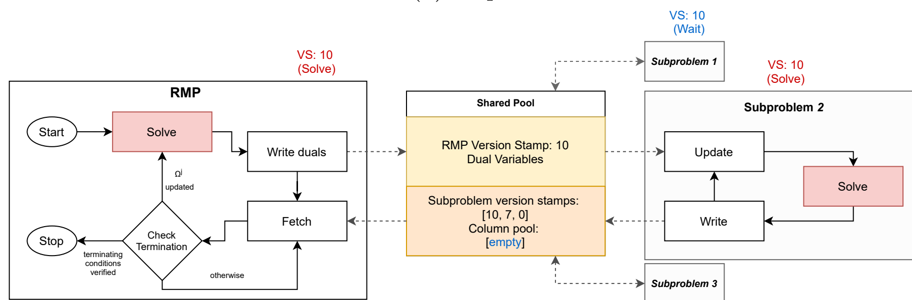
In Figure 6a, Subproblem 2 has finished computing a solution: therefore, it records the latest version stamp solved and writes the new column in the pool. Then, in 6b, the RMP fetches that column and updates its configuration. At



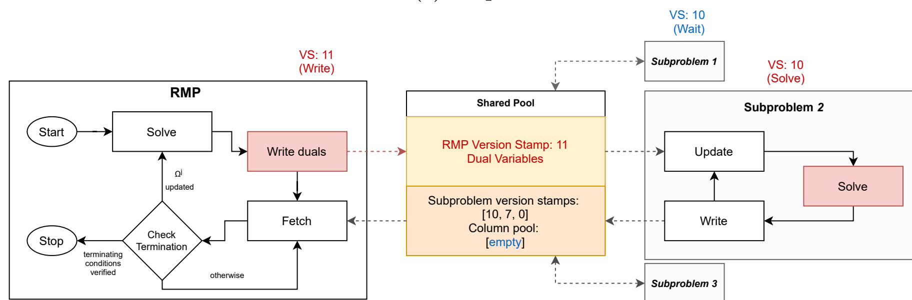
(a) Step 1



(b) Step 2



(c) Step 3



(d) Step 4

Figure 6: Asynchronous Column Generation example

the same time, Subproblems 2 updates with the latest dual variables that are available in the pool. Both RMP and Subproblem 2 start solving their new configurations in 6c. Finally, in 6d, the RMP finishes computation and updates both its version stamp and the dual variables in the pool.

A.3 A Multi-Dimensional Variable Size and Cost Bin-Packing Problem

Let I be a set of items and let K be a set of resources. A weight coefficient w_{ik} is given for each item $i \in I$ and resource $k \in K$. Additionally, let J be a set of bin types. Each bin type $j \in J$ is characterized by a given set of available bins N_j, along with a capacity C_j and a cost V_j.

We model the Multi-Dimensional Variable Size and Cost Bin-Packing Problem (MDVCSBP) as follows:

$$\min \sum_{j \in J} \sum_{q \in N_j} V_j y_{qj} \tag{10}$$

$$\text{s.t.} \sum_{j \in J} \sum_{q \in N_j} x_{iqj} = 1 \qquad \forall i \in I \tag{11}$$

$$\sum_{i \in I} w_{ik} x_{iqj} \leq C_j y_{qj} \qquad \forall j \in J, \forall q \in N_j, \forall k \in K \tag{12}$$

$$x_{iqj} \in \{0,1\} \qquad \forall i \in I, \forall j \in J, \forall q \in N_j \tag{13}$$

$$y_{qj} \in \{0,1\} \qquad \forall j \in J, \forall q \in N_j \tag{14}$$

and we give value 1 to variable x_{iqj}, if we assign item i to bin q of type j. It has value 0 otherwise. If we use a bin q of type j, $y_{qj} = 1$, otherwise $y_{qj} = 0$. The objective function (10) minimizes the overall cost of using bins. Constraints (11) enforce that every item has to be assigned to exactly one bin. Additionally, (12) impose that items cannot be assigned to unused bins and that the sum of the weights assigned to a given bin of type j, is not greater than its capacity, for each resource. Integrality conditions are then imposed by (13) and (14).

The extentend formulation, obtained through Dantzig-Wolfe decomposition, has one column for each feasible pattern of assignment of items to bins:

$$\min \sum_{p \in \Omega^j} \sum_{j \in J} V_j z_p^j \tag{15}$$

$$\text{s.t.} \sum_{p \in \Omega^j} \sum_{j \in J} x_{ip}^j z_p^j = 1 \qquad \forall i \in I \tag{16}$$

$$\sum_{p \in \Omega^j} z_p^j \leq |N_j| \qquad \forall j \in J$$

$$0 \leq z_p^j \leq 1 \qquad \forall j \in J, \forall p \in \Omega^j \tag{17}$$

where $x_{ip}^j = 1$ if item i is chosen, for a bin type j, in a pattern p. It is equal to 0 otherwise.

For each $j \in J$, Ω^j is set of indices of extreme points in the convex hull of $\{(x_1, \ldots, x_{|I|}) \in \mathbb{B}^{|I|} | \sum_{i \in I} w_{ik} x_i \leq C_j \forall k \in K\}$, that correspond to the feasible solutions of the following pricing problem:

$$
\begin{aligned}
\min\ & V_j y - \sum_{i \in I} \lambda_i x_i \\
\text{s.t.} & \sum_{i \in I} w_{ik} x_i \leq C_j y && \forall k \in K \\
& y \in \{0,1\} \\
& x_i \in \{0,1\} && \forall i \in I
\end{aligned}
$$

in which dual variables λ_i are associated to constraints (16).

A.4 A Vehicle Routing Problem with Time Windows

Let G = (V, A) be a graph that represents our transportation network, where V = {0, 1, . . . , n} is a set of n customers. Let node 0 and d be two depots where, respectively, vehicle routes start and terminate. Each node $i \in V$ requires a supply of q_i that can be satisfied by servicing it, with a service time s_i, during specific time windows that begin at a_i and end at b_i. Let l_{ij} be the length of each arc $(ij) \in A$ going from node i to node j. Let K be the set of vehicles and let H be the set of vehicle types. Each vehicle of type $h \in H$ requires a fixed cost F_h and is equipped with capacity W_h.

The model for the Vehicle Routing Problem with Time Windows (VRPTW) is the following:

$$
\begin{aligned}
\min\ & \sum_{i \in V} \sum_{j \in V} \sum_{k \in K} l_{ij} x_{ijk} + \sum_{h \in H} \sum_{k \in K} F_h z_{hk} && && (18) \\
\text{s.t.}\ & \sum_{j \in V} x_{jik} = \sum_{j \in V} x_{ijk} && \forall i \in V \setminus \{0, d\}, \forall k \in K && (19) \\
& \sum_{j \in V} x_{0jk} = 1 && \forall k \in K && (20) \\
& \sum_{j \in V} x_{jdk} = 1 && \forall k \in K && (21) \\
& \sum_{j \in V} \sum_{k \in K} x_{jik} = 1 && \forall i \in V \setminus \{0, d\} && (22) \\
& y_{ik} = \sum_{j \in V} x_{jik} && \forall i \in V, \forall k \in K && (23) \\
& \sum_{i \in V} w_i y_{ik} \leq \sum_{h \in H} W_h z_{hk} && \forall k \in K && (24) \\
& \sum_{h \in H} z_{hk} = 1 && \forall k \in K && (25) \\
& t_j \geq t_i + s_i + l_{ij} x_{ijk} - M(1 - x_{ijk}) && \forall i, j \in V, \forall k \in K && (26) \\
& x_{ijk} \in \{0,1\} && \forall i, j \in V, \forall k \in K && (27) \\
& y_{ik} \in \{0,1\} && \forall i \in V, \forall k \in K && (28) \\
& z_{hk} \in \{0,1\} && \forall h \in H, \forall k \in K && (29) \\
& a_j \leq t_j \leq b_j && \forall j \in V && (30)
\end{aligned}
$$

where $x_{ijk} = 1$ when vehicle k is travelling on arc (ij), $z_{hk} = 1$ when vehicle k is of type h, and $y_{ik} = 1$ when vehicle k visits node i. They are set to 0 otherwise. Finally, t_i decides the arrival time at node i.

The objective function (18) minimizes the travel cost for visiting the customers and the fixed costs for the heterogeneous fleet. Integrality constraints are managed in (27), (28) and (29). For each vehicle, constraints (19) impose that the number of incoming arcs in a node is equal to the number of outgoing arcs. The starting and ending depots are considered two special cases and are enforced respectively in (20) and (21). Constraints (22) impose that each node is visited exactly by one vehicle; this is also used in (23) to set variable y_{ik}. Capacity constraints are enforced in (24). (25) imposes that each vehicle can be at most of one type. Finally, (26) are subtour elimination constraints in Miller-Tucker-Zemlin form: the coefficient M needs to be set to a sufficiently large value (e.g. $\sum_{i \in V, j \in V} l_{ij}$). They both impose the consistency of time values t_j and forbid cycles. Together with (30), they impose time windows on variables t_j. We report that, from a practical resolution point of view, some simplification is possible. For instance, coefficients F_h can be mapped to the costs of arcs leaving the depot. Furthermore, all those pairs of nodes having $a_i + s_i + l_{ij} > b_j$ yield $x_{ijk} = 0$; the corresponding constraints (26) can be removed accordingly. Constraints (26) for pairs of nodes having $b_i + s_i + l_{ij} \leq a_j$ are redundant as well.

The extentend formulation, obtained through Dantzig-Wolfe decomposition, has one column for each feasible route of a given vehicle type:

$$\min \sum_{h \in H} \sum_{p \in \bar{\Omega}^h} C_p^h z_p^h \tag{31}$$

$$\text{s.t.} \sum_{h \in H} \sum_{p \in \bar{\Omega}^h} \sum_{j \in V} x_{jip}^h z_p^h \geq 1 \qquad \forall i \in V \setminus \{0, d\} \tag{32}$$

$$\sum_{h \in H} \sum_{p \in \bar{\Omega}^h} z_p^h \leq K \tag{33}$$

$$0 \leq z_p^h \leq 1 \qquad \forall h \in H, \forall p \in \bar{\Omega}^h \tag{34}$$

$$\tag{35}$$

where coefficients $x_{ijp}^h = 1$ if arc ij is included in a route p for vehicle of type h, 0 otherwise. Constraints (32) have been relaxed as inequalities without loss of generality, since an optimal solution always exists in which each node is not visited more than once. This allows to restrict in sign the corresponding dual variables.

For each $h \in H$, $\bar{\Omega}^h$ is the set of indices of feasible routes for vehicle h, which can be formally defined as the vectors $\{(x_{00}, \ldots, x_{|V||V|}) \in \mathbb{B}^{|V||V|}\}$ corresponding to the feasible solutions of the following pricing problem:

$$\min F^h + \sum_{i \in V} \sum_{j \in V} l_{ij} x_{ij} - \sum_{i \in V} y_i \lambda_i - \mu^h \tag{36}$$

$$\text{s.t.} \sum_{j \in V} x_{ji} = \sum_{j \in V} x_{ij} \qquad \forall i \in V \setminus \{0, d\} \tag{37}$$

$$\sum_{j \in V} x_{0j} = 1 \tag{38}$$

$$\sum_{j \in V} x_{jd} = 1 \tag{39}$$

$$y_i = \sum_{j \in V} x_{ji} \qquad \forall i \in V \tag{40}$$

$$\sum_{i \in V} w_i y_i \leq W^h \tag{41}$$

$$t_j \geq t_i + s_i + l_{ij} x_{ij} - M(1 - x_{ij}) \qquad \forall i, j \in V \tag{42}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in V \tag{43}$$

$$y_i \in \{0, 1\} \qquad \forall i \in V \tag{44}$$

$$a_j \leq t_j \leq b_j \qquad \forall j \in V \tag{45}$$

where λ_i are the dual variables associated to constraints (32) and μ^h is the dual variable associated to (33).

A.5 A Multi-Depot Vehicle Routing Problem with Time Windows

The Multi-Depot variant of the VRPTW that we consider (MDVRPTW) has the following additional feature: a set S of depots are located on the network defined in the previous section. Each vehicle can start from a different depot $s \in S$; any vehicle starting from s must also go back to s at the end of its route. Each depot s has a potentially different distance from each node of the graph G, and a potentially different fixed usage cost, which is incurred whenever each vehicle starts from that depot. We omit the compact formulation of the MDVRPTW, since it was never used in our experiments. The extentend formulation, obtained through Dantzig-Wolfe decomposition, has one column for each feasible route of a given vehicle type, starting from a given depot:

$$\min \sum_{s \in S} \sum_{h \in H} \sum_{p \in \bar{\Omega}^{s,h}} C_p^{s,h} z_p^{s,h} \tag{46}$$

$$\text{s.t.} \sum_{s \in S} \sum_{p \in \bar{\Omega}^{s,h}} \sum_{j \in V} \sum_{h \in H} x_{jip}^{s,h} z_p^{s,h} \geq 1 \qquad \forall i \in V \setminus \{0, d\} \tag{47}$$

$$\sum_{s \in S} \sum_{h \in H} \sum_{p \in \bar{\Omega}^{s,h}} z_p^{s,h} \leq K \tag{48}$$

$$0 \leq z_p^{s,h} \leq 1 \qquad \forall s \in S, \forall h \in H, \forall p \in \bar{\Omega}^{s,h} \tag{49}$$

$$\tag{50}$$

where coefficients $x_{ijp}^h = 1$ if arc ij is included in a route p for vehicle of type h starting from depot s, 0 otherwise. For each $s \in S$ and $h \in H$, $\bar{\Omega}^{s,h}$ is the set

of indices of feasible routes for vehicle h starting from s, which can be formally defined as before as vectors $\{(x_{00}, \ldots, x_{|V||V|}) \in \mathbb{B}^{|V||V|}\}$ corresponding to the feasible solutions of one in a set of $|S| \times |H|$ pricing problems. Each of them is identical to that of Appendix A.4, except that both s and h are fixed, defining costs F_h as well as the distance from the depot to the first customer in the route, and the last customer in the route and the depot.

A.6 Full experimental results

A.6.1 Vehicle Routing Problem with Time Windows

We report in Table 9 the wall clock time (Time) for each instance (Inst.) when solving VRPTW with DCG_{opt} over 1, 2, 3 and 4 nodes. We also present the speed-up (Spd-up) obtained over a single node. Average values are also presented for each class.

		1	2		3		4	
Class	Inst.	Time	Spd-up	Time	Spd-up	Time	Spd-up	Time
	R103	416.99	1.00	417.45	7.16	58.25	1.38	302.08
	R104	423.92	0.87	486.55	1.31	324.06	0.90	470.94
	R106	119.62	1.53	78.14	0.39	304.73	1.70	70.51
A	R107	309.43	0.81	384.11	1.02	302.57	1.02	302.75
	R108	414.94	0.70	592.72	1.19	348.33	0.68	610.99
	R110	311.31	1.01	308.80	1.66	187.31	1.03	302.74
	R111	202.21	0.59	342.20	0.59	340.21	0.66	304.37
A Overall		314.06	0.93	372.85	1.90	266.49	1.05	337.77
	R103	775.26	0.85	917.37	1.07	724.68	1.37	566.71
	R104	2969.57	1.80	1652.62	1.51	1960.82	1.98	1503.42
	R106	198.15	0.76	260.30	0.64	311.00	1.26	157.28
B	R107	715.74	0.80	894.02	0.85	838.60	1.11	645.51
	R108	2113.00	1.42	1483.10	1.34	1575.29	1.41	1503.31
	R110	405.55	0.95	425.81	1.10	368.72	1.62	249.79
	R111	557.23	1.09	510.35	1.17	476.04	1.19	469.84
B Overall		1104.93	1.10	877.65	1.10	893.59	1.42	727.98
	R103	1415.12	1.66	853.09	1.53	927.57	1.18	1202.90
	R104	4458.34	1.37	3242.78	0.94	4735.31	0.95	4675.51
	R106	209.54	1.25	167.84	1.29	162.67	0.94	222.52
C	R107	1137.63	0.99	1150.69	1.49	765.91	0.91	1245.15
	R108	3838.58	1.48	2599.65	1.32	2905.42	1.85	2079.46
	R110	598.96	0.97	617.09	1.48	404.53	1.43	419.96
	R111	702.27	1.27	555.11	1.21	581.49	1.26	557.89
C Overall		1765.78	1.28	1312.32	1.32	1497.56	1.22	1486.20
Overall		1061.59	1.10	854.28	1.16	885.88	1.23	850.65

Table 9: DCG_{opt} speed-up profiling over 1, 2, 3 and 4 machines when solving VRPTW

A.6.2 Vehicle Routing Problem with ad-hoc pricing algorithms

In Table 10 we present the wall clock time (Time) for each instance (Inst.) for solving VRPTW with DCG_{opt}, when using ad-hoc pricing algorithms, over 1, 2 and 4 nodes. Additionally, we report the speed-up (Spd-up) obtained over a single node.

Class		Inst.	1 Time	2 Spd-up	2 Time	4 Spd-up	4 Time
A		R103	35.84	1.14	31.32	0.86	41.83
		R104	659.85	1.31	504.33	1.36	486.76
		R106	5.11	0.79	6.47	1.41	3.61
		R107	97.38	1.32	73.67	1.65	59.13
		R108	741.94	1.14	648.54	1.33	557.02
		R110	5.96	1.02	5.84	1.17	5.11
		R111	21.84	1.16	18.82	0.80	27.35
	A Overall		223.99	1.13	184.14	1.22	168.69
B		R103	11.43	1.10	10.37	1.05	10.87
		R104	604.35	2.79	216.29	2.40	251.38
		R106	1.58	0.96	1.65	0.73	2.16
		R107	33.25	1.09	30.64	1.29	25.88
		R108	652.37	1.10	591.59	1.42	457.87
		R110	2.16	0.82	2.61	0.69	3.10
		R111	8.27	1.08	7.68	1.13	7.29
	B Overall		187.63	1.28	122.97	1.25	108.36
C		R103	14.08	1.94	7.27	1.95	7.22
		R104	279.78	1.60	175.29	1.35	207.76
		R106	1.58	1.11	1.43	0.81	1.94
		R107	21.41	0.92	23.36	0.90	23.71
		R108	512.98	0.96	537.03	1.49	345.40
		R110	1.65	0.83	1.99	0.68	2.43
		R111	6.97	1.49	4.67	1.14	6.12
	C Overall		119.78	1.26	107.29	1.19	84.94
	Overall		177.13	1.22	138.13	1.22	120.66

Table 10: DCG_{opt} speed-up profiling over 1, 2 and 4 machines when solving VRPTW with ad-hoc pricing algorithms

A.6.3 Multi Depot Vehicle Routing Problem

Finally, we report results for DCG_{opt} when solving MDVRPTW over 1, 2 and 4 nodes in Table 11. For each instance (Inst.) we present the wall clock time (Time) and the the speed-up (Spd-up) obtained over a single node. Average value are also presented for each class. They do not take into account instances for which a timeout has happened.

A.6.4 Comparison with CPLEX

In Table 12 we report the performance gains of DCG when solving every instance of MDVCSBP and VRPTW as the relative wall clock ratio (Spd-up) with respect to CPLEX. Values above 1 mean that DCG performs better. We also mark the instances in which CPLEX could not finish before the timelimit (TL) and the ones that reported an out of memory error (OOM) in the status column.

Class		Inst.	1	2		4	
			Time	Spd-up	Time	Spd-up	Time
		R101	71.54	1.44	49.81	2.60	27.47
		R102	1032.05	2.30	447.84	4.74	217.80
		R103	1803.60	2.12	849.56	5.56	324.25
		R104	1625.53	1.76	921.95	4.40	369.58
		R105	305.66	2.11	144.55	5.28	57.90
A		R106	1140.28	2.78	410.84	5.21	218.77
		R107	1474.37	2.15	686.71	5.06	291.26
		R108	1864.88	1.97	944.53	4.71	396.02
		R109	599.08	2.51	238.80	4.97	120.58
		R110	1222.62	2.29	534.75	4.94	247.63
		R111	1388.16	2.64	525.51	4.88	284.26
	A Overall		1138.89	2.19	523.17	4.76	232.32
		R101	50.30	1.96	25.65	4.17	12.06
		R102	1421.68	2.33	610.94	4.63	307.12
		R103	4377.66	1.69	2595.99	4.47	978.93
		R104	Timeout		8304.74		4223.15
		R105	343.41	2.28	150.67	4.29	80.14
B		R106	1853.13	2.61	709.79	4.51	410.66
		R107	4087.94	1.91	2138.61	4.84	845.29
		R108	22677.21	2.08	10910.36	7.72	2935.58
		R109	620.63	2.14	290.05	4.26	145.56
		R110	1989.61	2.29	867.01	3.72	535.22
		R111	2225.92	1.82	1221.33	3.72	597.76
	B Overall		3964.75	2.11	1952.04	4.63	684.83
		R101	45.57	1.90	24.03	3.56	12.81
		R102	1383.68	2.54	545.55	3.86	358.68
		R103	11762.76	4.03	2920.83	5.53	2125.85
		R104	Timeout		Timeout		Timeout
		R105	290.02	1.91	151.90	3.95	73.47
C		R106	1495.05	1.55	965.43	3.76	398.04
		R107	10484.52	4.55	2304.67	8.87	1182.11
		R108	Timeout		32837.00		14809.17
		R109	685.40	2.12	323.14	5.10	134.33
		R110	2694.77	2.18	1234.78	3.85	700.57
		R111	2940.33	2.26	1301.86	3.57	824.00
	C Overall		3531.35	2.56	1085.80	4.67	645.54
	Overall		2798.58	2.27	1168.25	4.69	507.12

Table 11: DCG_{opt} speed-up profiling over 1, 2 and 4 machines when solving MDVRPTW

MDVCSBP					VRPTW		
Objects	Bins	ID	Spd-up	Status	Instance	Spd-up	Status
500	250	1	80.70		LS_R103A	-	OOM
1000	250	2	25.82		LS_R103B	-	OOM
500	500	3	73.39		LS_R103C	-	OOM
1000	500	4	69.69	TL	LS_R104A	111.09	TL
500	750	5	279.22		LS_R104B	18.36	TL
1000	750	6	41.71	TL	LS_R104C	-	OOM
500	250	7	53.91		LS_R106A	-	OOM
1000	250	8	34.31		LS_R106B	-	OOM
500	500	9	63.65	TL	LS_R106C	-	OOM
1000	500	10	53.05	TL	LS_R110A	-	OOM
500	750	11	188.80	TL	LS_R110B	-	OOM
1000	750	12	11.11	TL	LS_R110C	-	OOM
500	250	13	75.08		LS_R111A	-	OOM
1000	250	14	53.33		LS_R111B	-	OOM
500	500	15	39.73		LS_R111C	-	OOM
1000	500	16	43.90		LS_R107A	-	OOM
500	750	17	229.14		LS_R107B	-	OOM
1000	750	18	20.51	TL	LS_R107C	-	OOM
500	250	19	76.00		LS_R108A	-	OOM
1000	250	20	16.96		LS_R108B	-	OOM
500	500	21	182.66		LS_R108C	-	OOM
1000	500	22	2.80				
500	750	23	122.35				
1000	750	24	20.85	TL			
500	250	25	305.05				
1000	250	26	43.03				
500	500	27	116.71				
1000	500	28	33.38	TL			
500	750	29	220.76				
1000	750	30	37.10	TL			
Overall			87.16			64.73	

Table 12: Summary of performance of DCG_{opt} against CPLEX in solving the specific MIPs used in the test.