Parallelisation and application of AD^3 as a method for solving large scale combinatorial auctions*

Francisco Cruz-Mencia[1,2], Jesus Cerquides[2], Antonio Espinosa[1], Juan Carlos Moure[1], and Juan A. Rodriguez-Aguilar[2]

[1] IIIA-CSIC, Campus de la UAB, s/n, Bellaterra, Barcelona
{fcruz,cerquide,jar}@iiia.csic.es,
[2] CAOS-UAB, Campus de la UAB, s/n,Bellaterra, Barcelona
{antoniomiguel.espinosa,juancarlos.moure}@uab.cat

Abstract. Auctions, and combinatorial auctions (CAs), have been successfully employed to solve coordination problems in a wide range of application domains. However, the scale of CAs that can be optimally solved is small because of the complexity of the winner determination problem (WDP), namely of finding the bids that maximise the auctioneer's revenue. A way of approximating the solution of a WDP is to solve its linear programming relaxation. The recently proposed Alternate Direction Dual Decomposition algorithm (AD^3) has been shown to efficiently solve large-scale LP relaxations. Hence, in this paper we show how to encode the WDP so that it can be approximated by means of AD^3. Moreover, we present *PAR-AD*^{3}, the first parallel implementation of AD^3. *PAR-AD*^{3} shows to be up to 12.4 times faster than CPLEX in a single-thread execution, and up to 23 times faster than parallel CPLEX in an 8-core architecture. Therefore *PAR-AD*^{3} becomes the algorithm of choice to solve large-scale WDP LP relaxations for hard instances. Furthermore, *PAR-AD*^{3} has potential when considering large-scale coordination problems that must be solved as optimisation problems.

Keywords: Combinatorial auctions, Large-scale coordination, Large-scale optimisation, Linear programming

1 Introduction

Auctions are a standard technique to solve coordination problems that has been successfully employed in a wide range of application domains [24]. Combinatorial auctions (CAs) [7] are a particular type of auctions that allow to allocate entire bundles of items in a single transaction. Although computationally very complex, auctioning bundles has the great advantage of eliminating the risk for a bidder of not being able to obtain complementary items at a reasonable price in a follow-up auction (think of a CA for a pair of shoes, as opposed to two consecutive single-item auctions for each of the individual shoes). CAs are expected to deliver more *efficient* allocations than non-combinatorial auctions complementarities between items hold.

* Research supported by MICINN projects TIN2011-28689-C02-01, TIN2013-45732-C4-4-P and TIN2012-38876-C02-01

CAs have been also employed to solve a variety of coordination problems (e.g. transportation [31], emergency resource coordination in disaster management [26], or agent coordination in agent-driven robot navigation [32]). However, although such application domains claim to be large-scale, namely involving thousands and even millions of bids, current results indicate that the scale of the CAs that can be optimally solved is small [19, 25]. For instance, CPLEX (a state-of-the-art commercial solver) requires a median of around 3 hours to solve the integer linear program encoding the Winner Determination Problem (WDP) of a hard instance of a CA with only 1000 bids and 256 goods. This fact seriously hinders the practical applicability of current solvers to large-scale CAs.

Linear Programming (LP) relaxations are a standard method for approximating combinatorial optimisation problems in computer science [5]. Yanover et al. [36] report that realistic problems with a large number of variables cannot be solved by off-the-shelf, commercial LP solvers (such as CPLEX). Instead, they propose the usage of TRBP, a message-passing, dual-decomposition algorithm, to solve LP relaxations, and show that TRBP significantly outperforms CPLEX. Since then, many other message-passing and dual decomposition algorithms have been proposed to address this very same problem [17, 18, 13, 28]. The advantage over other approximate algorithms is that the underlying optimisation problem is well-understood and the algorithms are convergent and provide certain guarantees. Moreover, there are ways of tightening the relaxation toward the exact solution [34].

In order to solve LP relaxations, there has been a recent upsurge of interest in the Alternating Direction Method of Multipliers (ADMM), which was invented in the 1970s by Glowinski and Marroco [14] and Gabay and Mercier [12]. As discussed in [6], ADMM is specially well suited for application in a wide variety of large-scale distributed modern problems. Along this line, Martins has proposed AD^3 [22], a novel algorithm based on ADMM, which proves to outperform off-the-shelf, commercial LP solvers for problems including declarative constraints. AD^3 has the same modular architecture of previous dual decomposition algorithms, but it is faster to reach consensus, and it is suitable for embedding in a branch-and-bound procedure toward the optimal solution. Martins derives efficient procedures for handling logic factors and a general procedure for dealing with dense, large, or combinatorial factors. Notice that until [21], the handling of declarative constraints by message-passing algorithms was barely addressed, and not well understood. This hindered their application to combinatorial auction WDPs, which typically require this type of constraints. Therefore, AD^3 constitutes a promising tool to solve WDPs in CAs.

As discussed in [21] (see section 7.5), AD^3 is largely amenable to parallelisation, since AD^3 separates an optimisation problem into subproblems that can be solved in parallel. Nonetheless, to the best of our knowledge there is no parallel implementation of AD^3. Therefore, the potential speedups that AD^3 may obtain when running on multicore environments remain unexplored. And yet, this path of research is encouraged by recent experiences in parallelisation of ADMM applied to solve an unconstrained optimisation problem [23]. Indeed, Miksik et al. show that a parallel implementation of ADMM delivers large speedups for large-scale problems. Notice though that the work

in [23] cannot be employed to solve the WDP for CAs because it cannot handle hard constraints.

The main purpose of this paper is to demonstrate that the optimisation and parallelisation of AD^3 can deliver enormous benefits when solving relaxations of large-scale combinatorial optimisation problems, and in particular WDPs in large-scale CAs. With this aim, we make the following contributions:

- We show how to encode the WDP for CAs so that it can be approximated by AD^3. For this endeavour we employ the computationally-efficient factors provided by AD^3 to handle hard constraints.
- We propose an optimised, parallel implementation of AD^3, the so-called *PAR-*AD^3. Our implementation is based on a mechanism for distributing the computations required by AD^3 as well as on a data structure organisation that together favor parallelism.
- We show that while AD^3 is up to 12.4 times faster than CPLEX in a single-thread execution, *PAR-*AD^3 is up to 23 times faster than parallel CPLEX in an 8-core architecture. Therefore *PAR-*AD^3 becomes the algorithm of choice to solve large-scale WDP LP relaxations.

To summarise, our results indicate that *PAR-*AD^3 obtains significant speed-ups on multi-core environments, hence increasing AD^3's scalability and showing its potential for application to large-scale combinatorial optimisation problems in particular and for large-scale coordination problems that can be cast as combinatorial optimisation problems. The rest of the paper is organised as follows. First, we introduce some background on AD^3. Next, we detail how to encode the WDP for CAs by means of AD^3. Thereafter, we thoroughly describe *PAR-*AD^3 and afterwards we present empirical results. Finally, we draw some conclusions and set paths to future research.

2 Background

Graphical models are widely used in computer vision, natural language processing and computational biology, where a fundamental problem is to find the maximum a posteriori probability (MAP) given a factor graph. Since finding the exact MAP is frequently an intractable problem, significant research has been carried out to develop algorithms that approximate the MAP.

Linear Programming (LP) relaxations have been extensively applied to approximate the MAP for graphical models since [30]. Typically, such application domains lead to sparse problems with a large number of variables and constraints (i.e beyond 10^4). As shown in [36], message passing algorithms have been proved to outperform state-of-the-art commercial LP solvers (such as e.g. CPLEX) when approximating the MAP for large-scale problems. This advantage stems from the fact that message-passing algorithms better exploit the underlying graph structure representing the problem.

Along this direction, several message passing algorithms have been proposed in the literature: ADMM [10], TRBP [35], MPLP [13], PSDD [18], Norm-Product BP [16], and more recently Alternate Direction Dual Decomposition (AD^3) [2].

As discussed in [22], the recently-proposed AD^3 has some very interesting features in front of other message passing algorithms: it reaches consensus faster than other algorithms such as ADMM, TRBP and PSDD; it does have neither the convergence problems of MPLP nor the instability problems of Norm-Product BP; and its anytime design allows to stop the optimisation process whenever a pre-specified accuracy is reached. Furthermore, as reported in [22], AD^3 has been empirically shown to outperform state-of-the-art message passing algorithms on large-scale problems.

Besides these features, AD^3 also provides a library of computationally-efficient factors that allow to handle declarative constraints within an optimisation problem. This opens the possibility of employing AD^3 to approximate constrained optimisation problems.

Algorithm 1 outlines the main operations performed by AD^3 on a factor graph G with a set of factors F, a set of variables V, and a set of edges $E \subseteq F \times V$. AD^3 receives a set of parameters θ that encode variable coeſficients and a penalty constant η able to regulate the update step size. We use the function $\partial(x)$ to denote all the neighbours (i.e. connected nodes) of a given graph node. The primal variables q and p, the dual λ as well as the unary log-potentials ξ are vectors which are updated during the execution .We refer the reader to [22] for a detailed description of the algorithm. AD^3 is an iterative three-step algorithm designed to approximate an objective function encoded as a factor graph. A key aspect of AD^3 is that it separates the optimisation problem into independent subproblems that progress to reach consensus on the values to assign to primal and dual variables. Thus, during the first step, *broadcast*, the optimisation problem is split into separate subproblems, each one being distributed to a factor. Thereafter, each factor locally solves its local subproblem. In AD^3, this computations is carried on solving a quadratic problem. During the second step, *gather*, each variable gathers the subproblems' solutions of the factors it is linked to. Finally, during the third step, *Lagrange updates*, the Lagrange multipliers for each subproblem are updated.

Algorithm 1 Alternating Directions Dual Decomposition(AD^3)

input: factor graph G, parameters θ, penalty constant η
initialize p (i.e. $p_i = 0.5 \forall i \in 1 \dots |V|$), initialize $\lambda = 0$
repeat ▷ Broadcast
  for each factor $\alpha \in F$ **do**
    for each $i \in \partial(\alpha)$ **do**
      set unary log-potentials $\xi_{i\alpha} := \theta_{i\alpha} + \lambda_{i\alpha}$
    end for
    $\hat{q}_\alpha := \textsc{SolveQP}(\theta_\alpha + \xi_\alpha, (\mathbf{p}_i)_{i \in \partial(\alpha)})$
  end for
  for each variable $i \in V$ **do** ▷ Gather
    compute avg $p_i := |\partial(i)|^{-1} \sum_{\alpha \in \partial(i)} \hat{q}_{i\alpha}$
    for each $\alpha \in \partial(i)$ **do** ▷ Lagrange updates
      $\lambda_{i\alpha} := \lambda_{i\alpha} - \eta(\hat{q}_{i\alpha} - p_i)$
    end for
  end for
until convergence
output: primal variables p and q, dual variable λ

A distinguishing feature of AD^3 is that both the broadcast and update steps can be safely run in parallel. Indeed, notice that, since subproblems are independent, they can be safely distributed in different factors so that each one independently computes a local solution. AD^3 provides a collection of factors for which their quadratic problems are defined. As an example we present how the quadratic problem for the XOR factor is solved in Algorithm 2, where the input of the algorithm are the potentials $Z_\alpha : z_0, \ldots, z_K$ relative to the factor α. Note that in Algorithm 1 the call to the SolveQP method has two parameters, the second parameter is omitted here since it is not needed to solve the XOR. Algorithm 2 proceeds as follows. Lines 11-13 are responsible of checking if the constraint XOR is already satisfied. Then, if not satisfied, the Z_α vector is transformed using the projection onto simplex method described by [9]. This method navigates through Z_α in decreasing order, to find the pivot element y_i and the value of τ. Afterwards this τ is used to perform the actual projection. To this end, two auxiliary vectors Z'_α and Y_α are used: the former will contain the algorithm output and the latter is used to contain a sorted copy of Z_α. Although there are ways to obtain the pivot without the need of sorting the vector Z_α (described in [9]), in AD^3 is preferable to have a persistent sorted vector since order of elements is commonly preserved or barely altered across the iterations. Therefore efficient sorting methods on nearly-ordered sequences can be applied. An important feature of the XOR factor is that its quadratic problem can be solved in $O(K \cdot logK)$, where K stands for the number of variables connected to the factor.

Algorithm 2 SolveQP for an XOR factor

input: $Z_\alpha : z_0, \ldots, z_K$, vector with α log-potentials

1: **function** FindTau(Y_α)
2: $\tau = 0.0$;
3: $sum := \sum_{y_i \in Y_\alpha} y_i$
4: **for each** $y_i \in Y_\alpha$ **do**
5: $\tau := \frac{sum-1}{K-i}$
6: **if** $y_i > \tau$ **then break**
7: update $sum := sum - y_i$
8: **end for**
9: return τ
10: **end function**
11: $z'_i := max(0, z_i)$, for each $z_i \in Z_\alpha$
12: $sum := \sum_{z'_i \in Z'_\alpha} z'_i$
13: **if** $sum > 1.0$ **then** ▷ Projection onto simplex
14: sort Z_α into Y_α: $y_0 \leq \ldots \leq y_K$
15: $\tau :=$ FindTau(Y_α)
16: $z'_i := \max(z_i - \tau, 0)$, for each $z_i \in Z_\alpha$
17: **end if**

output: Z'_α

As to gather, the step in which the subproblems communicate their local results, each variable can independently (from the rest of variables) gather and aggregate the

results computed by the factors it is linked to. Despite being highly prone to parallelisation, to the best of our knowledge there is only one public implementation of AD^3 and cannot run in parallel [3]. The recent contributions to the parallelisation of ADMM to solve unconstrained optimisation problems [23] are very encouraging because they show that it is possible to obtain very significant speedups by exploiting nowadays parallel hardware. This finding spurs and motivates the need for a parallel implementation of AD^3.

But before that, in the next section we show that the WDP for CAs can be solved by means of AD^3.

3 Solving combinatorial auctions with AD^3

A Combinatorial Auction (CA) is an auction in which bidders can place bids for a combination of items instead of individual ones. In this scenario, one of the fundamental problems is the Winner Determination Problem (WDP), which consists in finding the set of bids that maximise the auctioneer's benefit. Notice that the WDP is an $\mathcal{NP}$-complete problem.

Although special-purpose algorithms have addressed the WDP (e.g. [11, 29]), the state-of-the-art method for solving a WDP is to encode it as an integer linear program (ILP) and solve it using an off-the-shelf commercial solver (such as CPLEX [1] or Gurobi [15]). Nonetheless, this approach fails to scale to large CA instances. Indeed, as noticed in [31], real problems may involve up to millions of bids. Therefore, such real problems are out of reach for state-of-the-art optimal solvers, and hence the need for heuristic approaches arise.

As observed in [4], "The simplest and, perhaps most tempting approach, to an optimization-based heuristic is to round the solution to a linear programming relaxation". Furthermore, solutions to an LP relaxation can provide a very effective start to finding a good feasible solution to the non-relaxed optimisation problem. Hereafter we focus on solving the LP relaxation of the WDP by means of AD^3. Since AD^3 requires a factor graph to operate, we first show how to encode the WDP as a factor graph. Then we show how AD^3 can run on top of this factor graph. We shall start by showing such encoding by means of an example to finally derive a general procedure.

Consider an auctioneer puts on sale a pair of goods g_1, g_2. Say that the auctioneer receives the following bids: b_1 offering \$20 for g_1 ; b_2 offering \$10 for g_2; and finally b_3 offering \$35 for goods g_1 and g_2 together. The WDP for this CA can be encoded as the following ILP:

$$
\begin{array}{lll}
\text{maximise} & 20 \cdot x_1 + 10 \cdot x_2 + 35 \cdot x_3 & \\
\text{subject to} & x_1 + x_3 \leq 1 & [\textit{constraint } c_1] \\
& x_2 + x_3 \leq 1 & [\textit{constraint } c_2] \\
& x_1, x_2, x_3 \in \{0, 1\} &
\end{array}
$$

where x_1, x_2, and x_3 stand for binary decision variables that indicate whether each bid is selected or not; constraint c_1 expresses that good g_1 can only be allocated to either

[3] Available at http://www.ark.cs.cmu.edu/AD3/

bid b_1 or bid b_3 and constraint c_2 encodes that good g_2 can only be allocated to either bid b_2 or bid b_3.

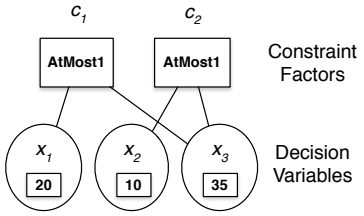

Fig. 1. Factor graph encoding of our CA example.

Now we can encode the optimisation problem above into a factor graph as illustrated in Figure 1. First, we create a variable node for each bid. Each variable contains its bid's offer (indicates the value that the auctioneer obtains when the variable is active). For instance, variable x_1 for bid b_1 contains value 20. Then we create a factor node per good, connecting the bids that compete for the good, and which are therefore incompatible. For instance, factor c_1 is linked to the variables corresponding to bids b_1 and b_3.

We observe that each factor representing a constraint in the factor graph in figure 1 corresponds to the "AtMost1" function introduced by Smith and Eisner [33], which is satisfied if there is at most one active input. Although AD^3 does not directly support "AtMost1" constraints, as seen in [21], an XOR factor can be used to define it by adding a slack variable to the factor. The XOR factor complexity is $O(K \cdot logK)$, where K stands for the number of variables connected to the XOR factor. Notice that the operation of AD^3 when solving the WDP only involves computationally-efficient factors.

4 Parallel realisation of AD^3

The AD^3 algorithm is amenable to general, architecture-level optimisation and parallelisation [21]. We propose an efficient realisation of the message-passing algorithmic pattern using shared variables and targeting multicore computer architectures. The so-called *PAR-AD*^{3}, that exploits the inherent parallelism at two dimensions: thread-level and data-level. For that, we reorganise both the data structures layout and the order of operations. The approach is generalisable to other similar graph processing algorithms. The key insights of our design are:

- An edge-centric representation of the shared variables that improves memory access performance.
- A reorganisation of the operations that promotes parallel scaling (thread parallelism) and vectorising (data parallelism).

4.1 Edge-centric shared data layout

AD^3 is a message passing algorithm that iterates on three steps: *broadcast*, *gather* and *Lagrange multiplier update*. The message passing pattern isolates the operations applied to the different elements of the graph (factors, variables and edges), so that multiple operations can be performed concurrently on the graph data. These operations and

data can then be physically distributed along different computation and storage elements.

The memory requirements of AD^3 are approximately proportional to the number of edges, and, for the problem sizes considered, they are fulfilled by most current shared-memory computer systems. In this situation, the fastest and most efficient mechanism for communication and synchronisation between processing cores is using shared variables (instead of explicit messages). The different processing cores of the computer will operate concurrently on the different elements of the graph (factors, variables or edges), both reading input data and generating new results stored in the shared memory. Execution performance is improved with a careful selection of synchronisation operations at the right point and an appropriate data structures layout.

Memory access performance is very sensitive to the data layout and data access pattern. When a loop has to iterate along a large regular data structure, the best performance is achieved when the next elements of the structure are naturally fetched from the next memory positions at each step of the iteration. Since AD^3 demands more computation work operating in edge data than in vertex or factor data, we adopt an edge-centric data representation, as reported in [27]. We want all information related to edges, such as unary log-potentials or lagrangian components, to be stored in consecutive memory positions. With this purpose, we apply a memory layout transformation that converts data structures originally designed in an Array of Structures (AOS) representation to a Structure of Arrays (SOA) representation.

Figure 2 illustrates how data was stored in memory in AD^3 and how the data layout is modified in *PAR-*AD^3. For the sake of clarity, we present data regarding 2 variables and 4 edges. AD^3 encodes the information following an AOS representation, where all properties related to each variable or edge are stored consecutively (see figure 2a). As the design is variable-centric, iterating on all the edges in the graph requires an indirect and scattered access to the variables (edges are accessed using the pointers associated to each variable). In contrast, the *PAR-*AD^3 SOA memory layout (figure 2b) stores the properties of variables and edges sequentially, thus resulting in a different array for each edge or variable property. Now, iterating on all edges of the graph requires consecutive accesses to array elements. The AOS memory representation of AD^3 benefits from memory access patterns where all the variable properties are used together, meanwhile the SOA memory representation of *PAR-*AD^3 benefits from the access of any property traversing all variables or edges.

To summarise, the *PAR-*AD^3 data representation transforms many scattered memory accesses into sequential, improving the memory access throughput. A derived advantage of the simplified edge access pattern is to foster better parallel scaling and vectorisation, but we need additional algorithmic transformations that are described in the next section.

4.2 Reordering operations

Parallel scaling means distributing compute operations on large chunks of data along different computational units sharing the same memory space. Vectorising applies data parallelism strategies inside the same computational unit, and consists in using instructions that operate simultaneously on a small vector of consecutive data elements. Both

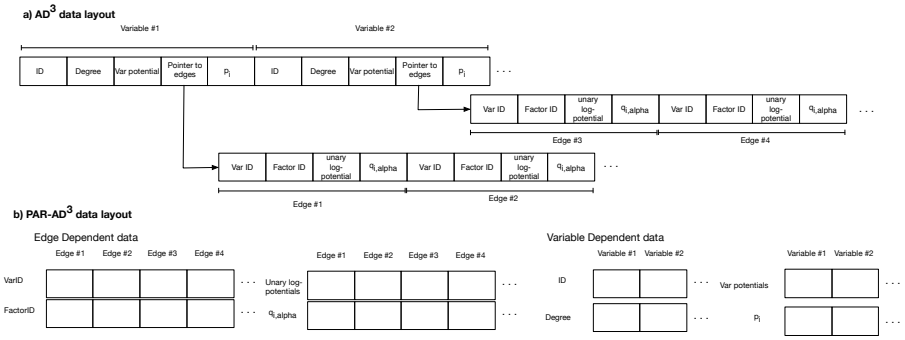

Fig. 2. a) AOS data representation of AD^3, compared to b) SOA data representation of *PAR-AD*^{3}.

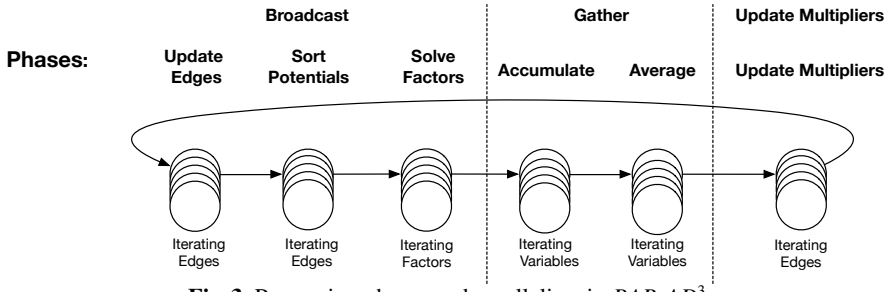

Fig. 3. Processing phases and parallelism in *PAR-AD*^{3}.

parallel scaling and vectorising are usually applied to simple loop iterations with clearly separated inputs and outputs, no recurrent dependencies, and sequential accesses to vector elements.

Our proposal reshapes the way the algorithm defines the graph operations towards a new structure of many simple consecutive loops, outlined in figure 3. The original *Broadcast* step is now split in three phases: *update edge*, *sort potential* and *solve factors*. Also, the original *Gather* step is now split in two phases: *accumulate* and *average*. Note that we iterate on factors twice and also iterate on variables twice: this makes the loops simpler and provides more data locality. As a result, all phases are now parallelised for concurrent execution (thread parallelism) and four out of six are vectorised: *update edge*, *accumulate*, *average* and *update multiplier*.

Algorithm 3 shows a pseudo-code of *PAR-AD*^{3} as a result of the optimizations applied. A pool of parallel threads is created outside of the main loop (line 2). Whenever a parallel loop inside the main loop is reached (lines 4, 9, 12, 19, 24, 27), the loop iterations are distributed to the threads for parallel execution. There is an implicit synchronisation after each loop, so that all threads wait for the generation of the results in one loop before starting the execution of the next.

As thoroughly described in the next section, these contributions have a significant impact in the sequential execution as well as allow good parallel scalability when an increasingly large number of threads are used. Since a clear trend in computer architecture is an increase of parallelism both at instruction and thread level, (for example, the intel Xeon Phi accelerator operates with 512-bit vector registers and contains more

Algorithm 3 *PAR-AD*^{3} pseudo-code

input: factor graph G, parameters θ, penalty constant η

initialize p (i.e. $p_i = 0.5 \forall i \in 1 \dots |V|$), initialize $\lambda = 0$
create threads
repeat
  parallel for $i\alpha \in E$ **do** ▷ Update edges
    Update log-potentials $\xi_{i\alpha} := \theta_{i\alpha} + \lambda_{i\alpha}$
    compute $\hat{q}_{i\alpha} = \theta_{i\alpha} + \xi_{i\alpha}$
    compute $\hat{q}'_{i\alpha} = max(0, \hat{q}_{i\alpha})$
  end for
  parallel for factor $\alpha \in F$ **do** ▷ Sort potentials
    $\hat{q}\_sorted_\alpha :=$ sort$(\hat{q}_\alpha)$
  end for
  parallel for factor $\alpha \in F$ **do** ▷ Solve factors
    $sum = \sum_{i \in \partial(\alpha)} (\hat{q}'_{i\alpha})$
    if $sum > 1.0$ **then**
      $\tau :=$ FindTau$(\hat{q}_sorted_\alpha)$
      $q'_{i\alpha} :=$ max$(q_{i\alpha} - \tau, 0)$, for each $q_{i\alpha} \in q_\alpha$
    end if
  end for
  parallel for variable $i \in V$ **do** ▷ Acummulate
    for $i \in \partial(\alpha)$ **do**
      $\tilde{p}_i := \tilde{p}_i + \hat{q}_{i\alpha}$
    end for
  end for
  parallel for variable $i \in V$ **do** ▷ Average
    $p_i := \tilde{p}_i / |\partial(i)|$
  end for
  parallel for $i\alpha \in E$ **do**
    $\lambda_{i\alpha} := \lambda_{i\alpha} - \eta(\hat{q}_{i\alpha} - p_i)$ ▷ Update multipliers
  end for
  update η
until convergence

output: primal variables p and q, dual variable λ

than 60 execution cores) the methodology applied to *PAR-AD*^{3} makes it ready to benefit from upcoming improvements.

5 Empirical evaluation

In this section, we assess *PAR-AD*^{3} performance against the state-of-the-art optimisation software CPLEX with the aim of determining the scenarios for which *PAR-AD*^{3} is the algorithm of choice. We also quantify its current gains, both in sequential and parallel executions. To this end, we first find the data distributions and range of problems that are best suited for *PAR-AD*^{3}. Thereafter, we briefly analyse two algorithmic key features: convergence and solution quality. Afterwards, we quantify the speedups of *PAR-AD*^{3} with respect to CPLEX in sequential and parallel executions. From this analysis we conclude that *PAR-AD*^{3} does obtain larger benefits from parallelisation than CPLEX. Indeed, *PAR-AD*^{3} achieves a peak speedup of 23X above CPLEX barrier, the state-of-the-art solver for sparse problems.

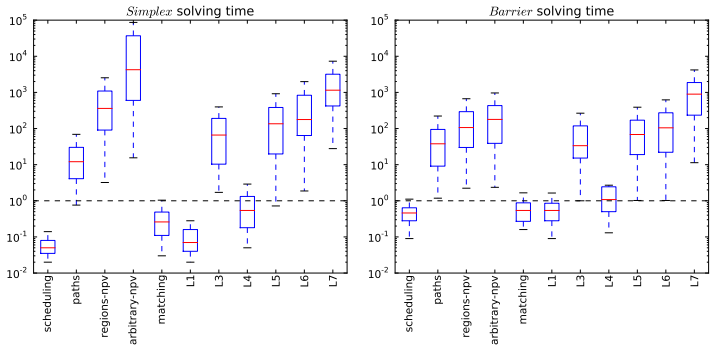

Fig. 4. Solving time for different distributions, single thread. a) Simplex b) Barrier

Experiment setup. In order to generate CA WDP instances, we employ CATS, the CA generator suite described in [20]. Each instance is generated out of the following list of distributions thoroughly described in [19]: arbitrary, matching, paths, regions, scheduling, L1, L3, L4, L5, L6 and L7. We discarded to employ the L2 distribution, because the CATS generator is not capable of generating large instances. While the first five distributions were designed to generate realistic CA WDP instances, the latter ones generate artificial instances. The main difference between the two distribution categories is the use of dummy goods that add structure to the problem inspired in some real life scenarios. i.e. Paths models the transportation links between cities; Regions models an auction of real estate or an auction where the basis of complementarity is the two-dimensional adjacency of goods; Arbitrary extends regions by removing the two-dimensional adjacency assumption, and it can be applied to model electronic parts design or procurement; Matching models airline take-off and landing rights auctions; and Scheduling models a distributed job-shop scheduling domain. Artificial (or Legacy) distributions have been often criticised [3, 20, 8] mainly due to their poor applicability, specially in the economic field. However they are interesting in order to study the algorithm performance in different situations. Both AD^3 and $PAR\text{-}AD^3$ are well suited for large-scale hard problems. For this reason, we first determine which of these distributions are hard to solve, putting special attention to the realistic ones. For our experimentation, we considered a number of goods within $[10^3, 10^4]$ in steps of 10^3 goods. Furthermore, the number of bids ranged within $[10^4, 4 \cdot 10^4]$ in steps of 10^4 bids. Each problem scenario is characterised by a combination of distribution, number of goods, and number of bids. Our experiments consider 5 different instances for each problem scenario and we analyse their mean value. Experiments are executed in a computer with two four-core Intel Xeon Processors L5520 @2.27GHz with 32 GB RAM with the hyper-threading mechanism disabled.

Different distributions hardness. We empirically determine the hardness of the relaxation for our experimental data by solving the LP using CPLEX simplex (simplex henceforth), CPLEX barrier (barrier henceforth), the state-of-the art algorithms. Results are plot in figures 4a and 4b. According to the results, scheduling and matching from the realistic distributions and L1, L4 from the legacy ones are very well addressed by

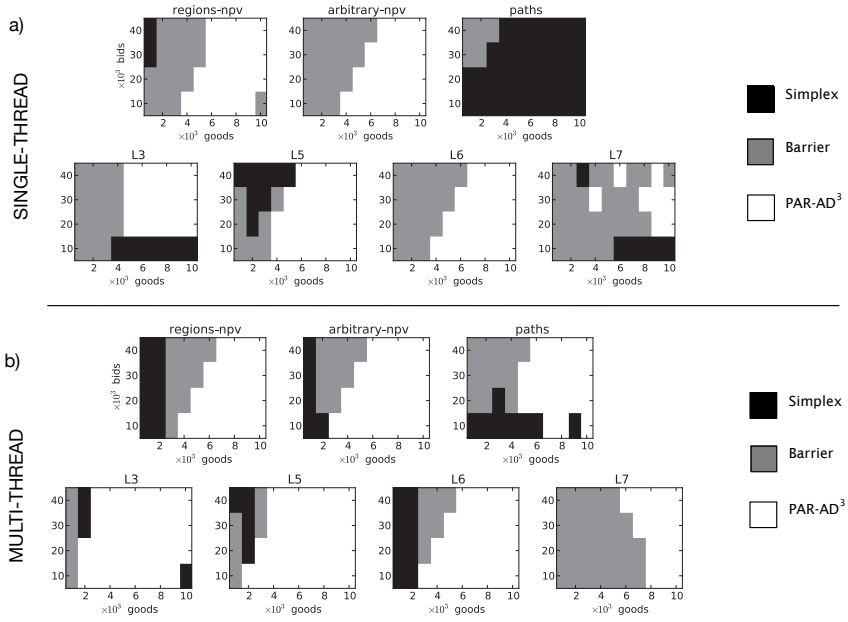

Fig. 5. Fastest algorithm solving different distributions and problem sizes. a) Single-thread, b) Multi-thread.

simplex, where solving time is, in general, less than one second. Both AD^3 and *PAR-*AD^3 are not competitive in this scenario. Applicability of *PAR-*AD^3 will be shown to be effective to the rest of distributions, especially in hard instances. Barrier is also doing a good job when the problems are hard, particularly in the arbitrary and regions distributions, where the representation matrix is more sparse.

Single-thread analysis. After comparing the publicly-available version of AD^3 against sequential *PAR-*AD^3, we observed that *PAR-*AD^3 outperformed AD^3 even in sequential execution, reaching an average speedup of 3X and a peak speedup of 12.4X. Moreover, we observed that the harder the instances, the larger the speedups of *PAR-*AD^3 with respect to AD^3. Since both algorithms are well suited for hard instances, this is particularly noticeable. Next, we compared the single-thread average performance of *PAR-*AD^3 against simplex and barrier. The results are plot in Figure 5a ,where we display the best algorithm for the different distributions and problem sizes. *PAR-*AD^3 is shown to be well suited for larger problems (the upper-right corner) in almost all the distributions. In general, barrier is the best algorithm in the mid-sized problems, while simplex applicability is limited to a small number of cases. Distribution paths presents a different behaviour, where adding goods increases the average bid arity and this is beneficial for simplex, which runs better in dense problems.

In general, the larger the WDP instances, the larger the *PAR-*AD^3 benefits. Single-threaded *PAR-*AD^3 reaches a peak speedup of 12.4 for the hardest distribution when compared to barrier, the best of the two state-of-the-art solvers.

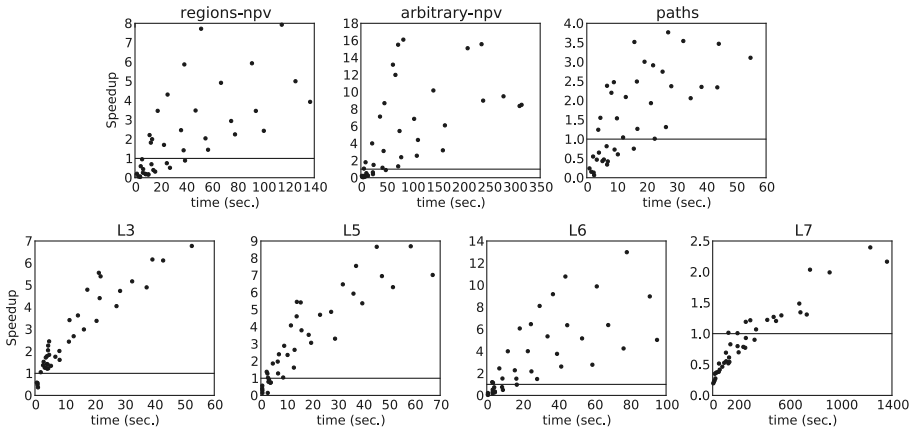

Fig. 6. Speedup of *PAR-AD*^{3} for different distributions against barrier in a multi-thread execution

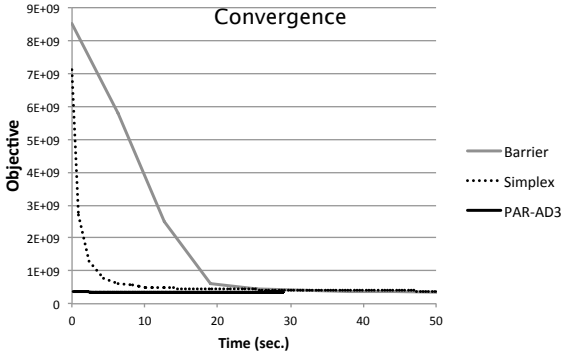

Fig. 7. Convergence of simplex, barrier and *PAR-AD*^{3}

Convergence and solution quality. Figure 7 shows a trace of an execution that illustrates the way the different solvers approximate the solution over time (using a regions distribution, 5×10^3 goods, and 10^4 bids). We chose this run because the similar performance of the three algorithms made them comparable. Note that *PAR-AD*^{3} converges to the solution in 29 sec., while barrier requires 102 sec. and simplex 202 sec. (not visible in the figure). Furthermore, notice that *PAR-AD*^{3} quickly reaches a high-quality bound, hence promptly guaranteeing close-to-the-solution anytime approximations. In general, our experimental data indicate that the initial solution provided by *PAR-AD*^{3} is always significantly better than the one assessed by both simplex and barrier. Finally, upon convergence, there is a maximum deviation of 0.02% between *PAR-AD*^{3} solutions and those assessed by CPLEX. Note that we run CPLEX with default parameters, has the feasibility tolerance set to 10^{-6}. This means that CPLEX solutions may be infeasible up to a range of 10^{-6} per variable. In the same sense, *PAR-AD*^{3} feasibility tolerance is set to 10^{-12}. This good initial solution is a nice property that makes *PAR-AD*^{3} suitable to be used as a method able to obtain quick bounds, either to be embedded in a MIP solver or also to provide a fast solution able to be used towards an approximate solution.

Multi-thread analysis. We have run *PAR-AD*^{3}, simplex and barrier with 8 parallel threads each, hence using the full parallelism offered by our computer. The results are displayed in figure 5b. When comparing with figure 5a (corresponding to the single-thread execution), we observe that *PAR-AD*^{3} outperforms simplex and barrier in many more scenarios, and in general *PAR-AD*^{3} applicability grows in concert with the parallel resources in all cases. Hence, we infer that *PAR-AD*^{3} better benefits from parallelisation than simplex and barrier. The case of the paths distribution is especially remarkable since simplex is faster than other algorithms when running in a single-thread scenario. Nonetheless, as *PAR-AD*^{3} better exploits parallelism, it revealed to be the most suitable algorithm for hard distributions when running in multi-threaded executions, including paths. In accordance with those results, it is expected that increasing the number of computational units will widen the range of applicability of *PAR-AD*^{3}.

Finally, we compared *PAR-AD*^{3} performance against barrier using 8 threads. We only compare *PAR-AD*^{3} to barrier since it is the best suited algorithm for the selected distributions (i.e in some executions *PAR-AD*^{3} can be up to three orders of magnitude faster than simplex). Figure 6 shows the average performance speedup of *PAR-AD*^{3} versus barrier as a function of the total running time of the execution of barrier (shown in the X-axis). We observe a clear trend in all scenarios: the harder the problem becomes for barrier, the larger the speedups obtained by *PAR-AD*^{3}. Our peak speedup is 23X (16X when taking the mean execution time of the different instances). The best results are achieved in the arbitrary distribution, which in addition was significantly better solved by barrier than by simplex according to figure 4. We recall that arbitrary is a distribution that can be applied to the design of electronic parts or procurement since it removes the two-dimensional adjacency of regions. In arbitrary, larger speedups correspond to the more sparse scenario, i.e. the bottom-right corner in figure 5.

6 Conclusions

In this paper we have tried to open up a path towards solving large-scale CAs. We have proposed a novel approach to solve the LP relaxation for the WDP. Our approach encodes the optimisation problem as a factor graph and uses *AD*^{3}, a dual-decomposition message-passing algorithm, to efficiently find the solution.

In order to achieve higher efficiency, we identified some of the bottlenecks found in message-passing graph-based algorithms and proposed some techniques to achieve good performance and scalability, in particular when executing in parallel. As a result of this analysis, we rearranged the operations performed by *AD*^{3} providing a new algorithm, the so-called *PAR-AD*^{3}, which is an optimised and parallel version of *AD*^{3}.

Our experimental results validate *PAR-AD*^{3} efficiency gains in large scale scenarios. We have shown that *PAR-AD*^{3} performs better than CPLEX for large-scale CAs in the computationally hardest distributions, both in single- and multi-threaded scenarios, with a peak speedup of 23X. Furthermore, the speedup is larger in multi-threaded scenarios, showing that *PAR-AD*^{3} scales better with hardware than CPLEX. Therefore, *PAR-AD*^{3} has much potential to solve large-scale coordination problems that can be cast as optimisation problems.

References

1. IBM ILOG CPLEX Optimizer. urlhttp://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, Last 2010.
2. P. Aguiar, E. P. Xing, M. Figueiredo, N. A. Smith, and A. Martins. An augmented lagrangian approach to constrained map inference. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 169–176, 2011.
3. A. Andersson, M. Tenhunen, and F. Ygge. Integer programming for combinatorial auction winner determination. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*, pages 39–46. IEEE, 2000.
4. M. O. Ball. Heuristics based on mathematical programming. *Surveys in Operations Research and Management Science*, 16(1):21–38, Jan. 2011.
5. D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1st edition, 1997.
6. S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.
7. P. Cramton, Y. Shoham, and R. Steinberg. *Combinatorial auctions*. MIT Press, 2006.
8. S. De Vries and R. V. Vohra. Combinatorial auctions: A survey. *INFORMS Journal on computing*, 15(3):284–309, 2003.
9. J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. Efficient projections onto the l 1-ball for learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning*, pages 272–279. ACM, 2008.
10. J. Eckstein and D. P. Bertsekas. On the douglas?rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming*, 55(1-3):293–318, 1992.
11. Y. Fujishima, K. Leyton-Brown, and Y. Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 548–553, 1999.
12. D. Gabay and B. Mercier. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & Mathematics with Applications*, 2(1):17–40, 1976.
13. A. Globerson and T. S. Jaakkola. Fixing max-product: Convergent message passing algorithms for map lp-relaxations. In *Advances in neural information processing systems*, pages 553–560, 2008.
14. R. Glowinski and A. Marroco. Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de dirichlet non linéaires. *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique*, 9(R2):41–76, 1975.
15. Z. Gu, E. Rothberg, and R. Bixby. Gurobi 4.0.2. software, Dec. 2010.
16. T. Hazan and A. Shashua. Norm-product belief propagation: Primal-dual message-passing for approximate inference. *Information Theory, IEEE Transactions on*, 56(12):6294–6316, 2010.
17. V. Kolmogorov. Convergent tree-reweighted message passing for energy minimization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(10):1568–1583, 2006.
18. N. Komodakis, N. Paragios, and G. Tziritas. Mrf optimization via dual decomposition: Message-passing revisited. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE, 2007.
19. K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM (JACM)*, 56(4):22, 2009.

20. K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76. ACM, 2000.
21. A. F. T. Martins. *The Geometry of Constrained Structured Prediction: Applications to Inference and Learning of Natural Language Syntax*. PhD thesis, Columbia University, 2012.
22. A. F. T. Martins, M. A. T. Figueiredo, P. M. Q. Aguiar, N. A. Smith, and E. P. Xing. Ad^3: Alternating directions dual decomposition for map inference in graphical models. *Journal of Machine Learning Research*, 46, 2014. to appear.
23. O. Miksik, V. Vineet, P. Perez, and P. H. S. Torr. Distributed non-convex admm-inference in large-scale random fields. In *British Machine Vision Conference (BMVC)*, 2014.
24. S. Parsons, J. A. Rodriguez-Aguilar, and M. Klein. Auctions and bidding: A guide for computer scientists. *ACM Comput. Surv.*, 43(2):10:1–10:59, Feb. 2011.
25. S. D. Ramchurn, C. Mezzetti, A. Giovannucci, J. A. Rodriguez-Aguilar, R. K. Dash, and N. R. Jennings. Trust-based mechanisms for robust and efficient task allocation in the presence of execution uncertainty. *Journal of Artificial Intelligence Research*, 35(1):119, 2009.
26. S. D. Ramchurn, A. Rogers, K. Macarthur, A. Farinelli, P. Vytelingum, I. Vetsikas, and N. R. Jennings. Agent-based coordination technologies in disaster management. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers*, pages 1651–1652, 2008.
27. A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.
28. A. M. Rush, D. Sontag, M. Collins, and T. Jaakkola. On dual decomposition and linear programming relaxations for natural language processing. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1–11. Association for Computational Linguistics, 2010.
29. T. Sandholm, S. Suri, A. Gilpin, and D. Levine. Cabob: A fast optimal algorithm for combinatorial auctions. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 1102–1108, 2001.
30. E. Santos Jr. On the generation of alternative explanations with implications for belief revision. In *Proceedings of the Seventh conference on Uncertainty in Artificial Intelligence*, pages 339–347. Morgan Kaufmann Publishers Inc., 1991.
31. Y. Sheffi. Combinatorial auctions in the procurement of transportation services. *Interfaces*, 34(4):245–252, 2004.
32. C. Sierra, R. de López Màntaras, and D. Busquets. Multiagent bidding mechanisms for robot qualitative navigation. In *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 198–212. Springer, 2001.
33. D. Smith and J. Eisner. Dependency parsing by belief propagation. In *Proceedings of the Conference on Empirical Conference on Empirical Methods in Natural Language Processing*, number October, pages 145–156, 2008.
34. D. Sontag, T. Meltzer, A. Globerson, T. S. Jaakkola, and Y. Weiss. Tightening lp relaxations for map using message passing. *arXiv preprint arXiv:1206.3288*, 2012.
35. M. J. Wainwright, T. S. Jaakkola, and A. S. Willsky. Tree-reweighted belief propagation algorithms and approximate ml estimation by pseudo-moment matching. In *Workshop on Artificial Intelligence and Statistics*, volume 21, page 97. Society for Artificial Intelligence and Statistics Np, 2003.
36. C. Yanover, T. Meltzer, and Y. Weiss. Linear programming relaxations and belief propagation – an empirical study. *J. Mach. Learn. Res.*, 7:1887–1907, Dec. 2006.