Comput Optim Appl (2009) 43: 411–436
DOI 10.1007/s10589-007-9146-5

Towards auction algorithms for large dense assignment problems

Libor Buš · Pavel Tvrdík

Received: 12 October 2006 / Revised: 9 October 2007 / Published online: 16 November 2007

Abstract In this paper, we focus on the problem of solving large-scale instances of the linear sum assignment problem by auction algorithms. We introduce a modified auction algorithm, called look-back auction algorithm, which extends the forward auction algorithm by the ability of reusing information from previous bids. We show that it is able to reuse information from the previous bids with high efficiency for all tested types of input instances. We discuss then the design and implementation of a suite of sequential and distributed memory auction algorithms on a Linux cluster with the evaluation on several types of input instances of the linear sum assignment problem. Our results show that the look-back auction algorithm solves sequentially nearly all types of dense instances faster than other evaluated algorithms and it is more stable than the forward-reverse auction algorithm for sparse instances. Our distributed memory auction algorithms are fully memory scalable.

Keywords Linear sum assignment problem · Auction algorithm · Linear optimization · Distributed algorithms

1 Introduction

One of the classic combinatorial optimization problems is the *assignment problem*. The task is to find an optimal assignment of *n persons* to *n objects*. A specific type of the assignment problem depends on the interpretation of the optimality measure.

This research has been supported by IGA CTU under grant CTU0308013 and under research program MSMT 6840770014.

L. Buš · P. Tvrdík (✉)
Department of Computer Science and Engineering, Czech Technical University, Prague, Czech Republic
e-mail: tvrdik@fel.cvut.cz

L. Buš
e-mail: xbus@fel.cvut.cz

In this paper, we consider the Linear Sum Assignment Problem (LSAP). Its inputs are two distinct sets of n persons and n objects and a *cost matrix* $A_{n\times n} = [a_{ij}]$, where a_{ij} is the cost of the assignment of object j to person i.

The goal is to find an *optimal assignment* $M = \{(i, j_i),\ i = 0, \ldots, n-1\}$, i.e., one-to-one mapping from the set of persons to the set of objects such that $\sum_{i=0}^{n-1} a_{ij_i}$ is maximum. We consider the *maximization* formulation of the LSAP, since it is closer to real-world auctions. Both minimization and maximization versions are computationally equivalent and one can be transformed to the other one by a simple transformation

$$a_{ij} = C - a_{ij}, \quad \forall\, 0 \le i, j \le n-1, \tag{1}$$

where $C = \max_{i,j}\{a_{ij}\}$. A LSAP with the same number of persons and objects is called *symmetric*, otherwise it is called *asymmetric*.

The LSAP can also be formulated using graph theory terminology as a *maximum weight matching problem* in bipartite graphs or as a primal *linear programming problem*:

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{i=0}^{n-1} x_{ij} = 1, \quad \forall\ j = 0, \ldots, n-1, \\
& \sum_{j=0}^{n-1} x_{ij} = 1, \quad \forall\, i = 0, \ldots, n-1, \\
& x_{ij} \in \{0, 1\}, \quad \forall\, i, j = 0, \ldots, n-1.
\end{aligned} \tag{2}$$

A solution matrix $[x_{ij}]$ is *primal feasible* if it satisfies the constraints and it is *primal infeasible* otherwise. It corresponds to an assignment M as follows: $(i, j) \in M$ iff $x_{ij} = 1$.

The *dual problem* to (2) is to

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=0}^{n-1} r_i + \sum_{j=0}^{n-1} p_j \\
\text{subject to} \quad & r_i + p_j \ge a_{ij}, \quad \forall\, i, j = 0, \ldots, n-1,
\end{aligned} \tag{3}$$

where *dual variables* r_i (p_j) correspond to the first (second, respectively) set of equality constraints (2). A pair of vectors $\vec{r}$, $\vec{p}$ is *dual feasible* if both vectors satisfy the constraints of problem (3) and it is *dual infeasible* otherwise. In the further text, we call p_j the *price* of object j and r_i the *profit* of person i.

According to the famous linear programming *Duality Theorem*, if the primal problem (2) has an optimal solution, then the dual problem (3) has also an optimal solution and the values of objective functions in problems (2) and (3) are the same.

To recover the optimal primal solution when only an optimal dual solution is known, we can use the *Complementary Slackness Theorem*: the primal solution $[x_{ij}]$

and the dual solution $\vec{p}, \vec{r}$ are optimal iff

$$x_{ij}(a_{ij} - r_i - p_j) = 0, \quad \forall\, 0 \leq i, j \leq n-1. \tag{4}$$

We say that a LSAP is *sparse* if its cost matrix has sparsity (the ratio of zero elements to all elements) less than 10%, *fairly dense* if the sparsity is between 10% and 90%, and *dense* if the sparsity is more than 90%.

LSAP problems appear in several application domains. Most of them deal with sparse LSAP instances and therefore, the literature focuses mainly on algorithms for sparse problems. For instance, in paper [15], authors described a row pivoting method for a sparse linear solver based on maximization of the product (or sum) of diagonal elements of the input matrix by means of the LSAP.

Even though many real world problems can be modeled as sparse LSAPs, we focus in this paper on large dense and fairly dense LSAPs which usually brings the complication that they do not fit in the memory of the computing processor. Our approach is to distribute data among more processors and solve the LSAP in parallel.

Another technique used in literature is based on simple sparsification. It consists in keeping for each person i only k highest costs a_{ij} and solving the modified instance [22]. If the solution does not satisfy the optimality condition for the original instance, l highest costs that do not satisfy this condition are added to each person and the instance is solved until the optimal solution is found. The experimental results revealed that for random dense instances (see Sect. 6 for description of input instances), this approach works quite well, while for geometric instances, more rounds are needed and the performance is degraded.

The most important application domain of the dense LSAP we found in the literature are approximation algorithms and heuristics for the Traveling Salesman Problem (TSP) and more general Vehicle Routing Problem, which are both NP-hard. The importance of the LSAP follows mainly from the fact that it can be obtained as a relaxation of the asymmetric TSP. For large problems, its solution is usually quite close to the optimal tour length. In paper [25], the authors even report on solving large dense random asymmetric TSP instances upto 500 000 cities optimally by the LSAP. It is also used in several asymmetric TSP heuristics [16, 17, 21, 29].

The most important structure of instances for the symmetric TSP are two-dimensional geometric instances (with, e.g., Euclidean measure). Symmetric TSP algorithms usually compute the distances online and thus, they take linear amount of memory. General asymmetric TSP heuristics do not allow linear size instance representation, since there is a variety of possible structures and therefore, they have to allocate the whole matrix in the memory. This is the reason why experiments with asymmetric TSP heuristics deal with much smaller sizes of instances than for the symmetric TSP: the largest asymmetric TSP instance in TSPLIB[1] has only 443 cities. This limitation triggers the need for an implementation of a LSAP algorithm developed for huge dense instances.

Next application domain of the dense LSAP is in object recognition and in computer vision. In paper [2], the authors presented an approach for computing similarity

[1] A free online library of sample instances of TSP and related problems, see http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html.

between two 2D shapes. They represented shapes as long sequences of points sampled from the shape contours. For each point they computed a shape context, i.e., a distribution of the rest of the shape with respect to the chosen point. The correspondence problem deals with finding points on both shapes with the most similar shape contexts and it was formulated as a dense LSAP. We have observed that the computation of the cost matrix is an inherently parallel task efficiently implementable on all kinds of parallel platforms. The paper also describes experiments based on categorization of simple images. The authors used 100 to 300 sample points and they solved the LSAP by the sequential shortest-path algorithm proposed in [20]. As we will show in Sect. 6.2, our combination of the look-back and forward auction algorithms performs better for geometrical instances even of smaller sizes than the algorithm in [20]. Because of small number of sample points used in this approach, distributed auction algorithms are not applicable.

In this paper, we present two new contributions to solving large fairly dense and dense assignment problems. The first one is a implementation of *look-back auction algorithm* based on look-back bidding technique which is generalization of an idea suggested in [7] to consider in bidding also the third highest benefit. This algorithm appeared also to be efficient and stable for large sparse random instances with degree of each row sufficiently large (say greater than 300).

The second result is a design and implementation of 3 truly distributed memory auction algorithms: forward, forward-reverse, and look-back. They allow to solve LSAPs of arbitrary size, and thus also fairly dense and dense assignment instances which do not fit into main memory of a uniprocessor computer. Another advantage is that they can be combined with distributed memory parallelization of the cost matrix computation task efficiently. We have implemented them on a cluster of Linux workstations and evaluated on large data instances which did not fit into memory of a single processor.

The main advantage of the look-back bidding is that in sequential auction algorithms, it reduces the operational complexity of the evaluation of bids and in distributed auction algorithms, it reduces interprocessor communication, since the processors can evaluate more bids locally than in distributed memory forward and forward-reverse auction algorithms.

This paper is organized as follows: At first, we review existing LSAP algorithms in Sect. 2. In Sect. 3, we give an overview of the existing auction algorithms for the LSAP. In Sect. 4, we describe our look-back auction algorithm and in Sect. 5, our design and implementation of distributed memory auction algorithms. Finally, in Sect. 6, we show experimental results on various input instances.

2 Review of LSAP algorithms

In this section, we will survey some sequential and parallel algorithms developed for the LSAP. For a comprehensive list, we refer the reader to report [12].

The sequential algorithms can be classified as primal-dual algorithms or linear programming algorithms. The primal-dual algorithms work with an infeasible primal solution $[x_{ij}]$ and a feasible dual solution $\vec{r}$, $\vec{p}$ and update iteratively these solutions

until the primal solution becomes feasible. Most of them are derived from the *Hungarian algorithm* (HA) [23] in which a maximal alternating forest is constructed in each iteration to augment the primal solution. The HA has computational complexity $O(n^3)$. Another primal-dual algorithm is the *auction algorithm* (AA) first proposed in [4]. The main difference is that in the HA, the dual objective function increases in each iteration, while the AA guarantees that it does not decrease by more than ϵ. The second difference is that in the AA assigned persons may become unassigned, while in the HA they remain assigned. The complexity of scaled AAs is $O(nm \log(nC))$, where $C = \max_{ij}\{a_{ij}\}$ and m is the number of arcs in the bipartite graph corresponding to the input data. We will give more details on AAs in Sect. 3. Variations of the HA are the *shortest path algorithms* (SPA), which use only shortest alternating paths to augment primal solutions. Most of them use the Dijkstra's algorithm for the shortest augmenting path computation. For example, see paper [20] that describes an $O(n^3)$ implementation of the SPA based on combination with an initialization phase running auctions with $\epsilon = 0$. The idea of this initialization is originated in paper [5]. Better and optimal complexity $O(\sqrt{n}m \log(nC))$ can be achieved by combination of the AA with the SPA such that in each scaling phase the AA assigns the first $n - O(\sqrt{n})$ persons and the SPA finishes the assignment of the last $O(\sqrt{n})$ persons [7]. The last subclass of primal-dual algorithms consists of *pseudoflow algorithms* [18] which solve the corresponding minimum cost flow problem also with the optimal complexity $O(\sqrt{n}m \log(nC))$.

The linear programming algorithms solve the LSAP by general linear programming approaches, e.g., by a *simplex method* or by an *interior point method* [22, 26].

In [19] authors described a CRCW PRAM linear programming algorithm for the LSAP with operational complexity $O(\sqrt{m} \log^2 n \log nC)$ using m^3 processors, where m is again the number of arcs, but this result is only of theoretical interest. The real parallel implementations of the LSAP algorithms are based on parallel versions of shortest augmenting path constructions [1, 9] and AAs [8].

Distributed memory architectures, namely clusters, have several advantages compared to shared memory architectures—low price/performance ratio and memory scalability. For this reason, clusters were our primary interest for solving large scale dense LSAPs. Unlike in shared memory architectures, there is an additional parameter that shall be taken into consideration when designing a distributed memory algorithm—the latency of communication, which can be several orders higher than local computation complexity. It makes distributed memory architectures less efficient. We are aware of only 2 papers describing parallel AAs on a distributed memory MIMD computer or a cluster.

Paper [27] compares implementations of the HA, SPA, and AA on a MEIKO distributed memory machine with 16 nodes. Their implementation of SPA and AA stores a whole copy of the cost matrix on each processor. Thus, it does not take advantage of the aggregate memory capacity. Authors concluded that AAs lead to an easy and efficient parallelization.

In paper [11], there is also a section describing an implementation of the forward AA on a cluster of workstations interconnected with a 10 Mb Ethernet. Authors used columnwise mapping of the cost matrix. The paper does not give any specific performance results. Our understanding is that the results were disappointing due to

expensive broadcast and reduce operations and led the authors to a conclusion that a distributed memory implementation is not worth considering.

From the previous work, it follows that the AAs have several advantages compared to alternative primal dual algorithms mentioned above. The main advantage is good performance and efficiency of the AAs for large and dense LSAPs (see papers [14, 18], and Sect. 6 for comparison with implementation of the SPA published in [20]). Since AAs are inherently distributed, it is easier to implement them on distributed memory parallel architectures and therefore, they enable to solve problems of arbitrary size. Moreover, hybrid AAs support easy adaptation of computation to communication ratio. For these reasons, we focus in the following text only on AAs.

3 Auction algorithms

Let us describe briefly the forward and forward-reverse AAs for the LSAP. For more details, we refer the reader to book [7].

Let j_i denote the object assigned to person i. The *complementary slackness* (CS) condition for assignment M and vector $\vec{p}$ to be primal and dual optimal is

$$a_{ij_i} - p_j \geq \max_{0 \leq k \leq n-1} \{a_{ik} - p_k\}, \quad \forall\, (i, j_i) \in M. \tag{5}$$

A very important parameter of AAs is the *minimal bidding increment* ϵ, allowing persons to be assigned to objects that come within ϵ of attaining the maximum in (5). Then, we get the ϵ*-complementary slackness* (ϵ-CS) condition

$$a_{ij_i} - p_{j_i} \geq \max_{0 \leq k \leq n-1} \{a_{ik} - p_k\} - \epsilon, \quad \forall\, (i, j_i) \in M. \tag{6}$$

An AA using constant $\epsilon < \frac{1}{n}$ is called *unscaled*. However, in most cases, the total number of bids needed to find a complete assignment can be reduced if the AA scales down the value of ϵ in more ϵ*-scaling phases*, so that each phase has a fixed $\epsilon > 0$. This approach is used in *scaled* AAs. We will discuss these issues in Sect. 3.2.

3.1 Forward AA

The input of a *forward AA* (FAA) [4] is a cost matrix $A_{n \times n} = [a_{ij}]$ and the output is an assignment M. A FAA starts with an assignment M and a price vector $\vec{p}$ that satisfies the ϵ-CS condition (6). This condition is preserved, as the FAA proceeds. Initially, M can be empty and $\vec{p}$ the zero vector, i.e., $p_j = 0$ for all j.

A scaled FAA repeats ϵ-scaling phases until an optimal assignment M is found. A given phase terminates if the number of still unassigned persons decreases to zero. Then the current price vector $\vec{p}$ is passed to the next phase that starts with a new (smaller) ϵ and the empty assignment.

More specifically, each phase consists of one or more *cycles*. Each cycle starts with a set of currently unassigned persons R. In the first cycle of a given phase, R is the set of all persons. Cycles are repeated within a phase until $|R| = 0$.

During one cycle, the FAA executes one or more *bidding iterations*. In each of them, it chooses a *block* $I \subseteq R$ of *bidders* and removes I from R, i.e., each bidder can participate in one bidding in each cycle. The persons in I try to acquire objects *simultaneously*. There are three possible approaches to choose a block I for bidding. In the *Gauss-Seidel* approach, I consists of just one unassigned person: $|I| = 1$. In the *Jacobi* approach, I consists of all currently unassigned persons: $I = R$. The *hybrid* approach is a combination of both: $1 < |I| < |R|$.

Each bidding iteration starts with a *bidding round* followed by an *assignment round*. In a bidding round, for each person $i \in I$, the FAA finds the object j_i with the *maximal benefit* v_i, where

$$v_i = a_{ij_i} - p_{j_i} = \max_{0 \le j \le n-1} \{a_{ij} - p_j\}$$

and also the value of the *second maximal benefit* w_i, where

$$w_i = \max_{j, j \neq j_i} \{a_{ij} - p_j\}.$$

Then the *bid value* b_{ij_i} of person i is computed as

$$b_{ij_i} = p_{j_i} + v_i - w_i + \epsilon$$

and person i submits bid b_{ij_i} for object j_i. In the Jacobi and hybrid versions of the FAA, more persons are bidding concurrently, and therefore, the conflicts in which several persons submit a bid to the same object may appear. Let B_j denote the set of persons that submitted bids for object j.

In the following assignment round, the algorithm finds for each object j that has received at least one bid, i.e., $B_j \neq \emptyset$, the maximal bid $b_{i_j j} = \max_{i \in B_j} \{b_{ij}\}$ of a person i_j. If object j is already assigned to a person i_j^*, it unassigns the old *owner* i_j^* and updates the price for object j:

$$p_j = b_{i_j j}.$$

The old owner i_j^* is included into the set R_{new} of all unassigned persons for the next cycle.

The FAA guarantees that in each cycle $|R|_{\text{new}} \le |R|$ and that an object that receives a bid is assigned until the scaling phase finishes. The next important property of the FAA is that the prices of objects are not decreasing. More specifically, the price of an object that received a bid increases by at least ϵ.

3.2 ϵ-scaling

It has been shown, see for example book [7], that for input instances with integer costs, the assignment found by the FAA scaling phase with $\epsilon < \frac{1}{n}$ is optimal.

We have already mentioned that an unscaled FAA runs one scaling phase with constant $\epsilon < \frac{1}{n}$, e.g., $\epsilon = \frac{1}{n+1}$. But the so called *price wars* can increase the number of cycles and considerably degrade the performance of unscaled FAAs.

A price war arises when a group of bidders competes for a smaller group of objects such that each object provides approximately the same profit to each bidder. They can terminate only if sufficient number of bidders give up since the objects become overpriced. The ϵ-scaling is absolutely necessary for reasonably fast resolution of price wars.

According to our knowledge, all published AAs use the same mechanism of scaling ϵ in each scaling phase. More specifically, in the first scaling phase, an AA uses a large initial value $\epsilon = \epsilon_b$ and decreases ϵ by a factor $\Theta > 1$ in each subsequent scaling phase while $\epsilon > \frac{1}{n+1}$. The complete assignment found in the last scaling phase, in which $\epsilon = \frac{1}{n+1} < \frac{1}{n}$, is an optimal solution of a given LSAP. The choice of ϵ-scaling parameters is given in paper [14], e.g., $\epsilon_b = C/5$, where $C = \max_{i,j}\{a_{ij}\}$ is the maximal cost, and $\Theta = 5$ for dense LSAPs.

The number of ϵ-scaling phases is $O(\log(nC))$ and each phase takes $O(n^3)$ time. Thus, the complexity of an ϵ-scaled AA is $O(n^3 \log(nC))$ [7]. However, an interesting analysis in paper [28] reveals that the real running time for dense random instances of the LSAP is only $O(m \log(n))$.

3.3 Reverse & forward-reverse AAs

If we interchange the roles of persons and objects in a FAA, we get a *reverse AA* (RAA), in which the objects compete for persons. Analogously to vector $\vec{p}$ of object prices in the FAA, the RAA maintains a profit r_i for each person i.

Briefly, let S be the set of all unassigned objects at the beginning of a RAA cycle. In each bidding iteration, the RAA chooses a nonempty subset $J \subseteq S$ of unassigned objects and for each object $j \in J$, it computes its bid $b_{i_j j}$. Then it chooses the maximal bid b_{ij} for each person i that has received at least one bid, increases its profit r_i, and assigns person i to object j.

A *forward-reverse AA* (FRAA) [6] is a hybrid of a FAA and a RAA. It maintains simultaneously both the price vector $\vec{p}$ and profit vector $\vec{r}$. Thus, the ϵ-CS condition for an assignment M and vectors $\vec{p}, \vec{r}$ is defined as:

$$\begin{aligned} r_i + p_j &\geq a_{ij} - \epsilon, \quad \forall\, 0 \leq i, j \leq n-1, \\ r_i + p_j &= a_{ij}, \qquad \forall\, (i,j) \in M. \end{aligned} \tag{7}$$

It is easy to verify that condition (7) implies condition (6) for the FAA and a similar condition for the RAA.

One scaling phase of the FRAA alternates *forward stages* with *reverse stages*: A forward stage consists of several cycles of the FAA and sets the profit of each assigned person i to $r_i = a_{ij_i} - p_{j_i}$. A reverse stage consists of several cycles of the RAA and changes the price of each assigned object j to $p_j = a_{i_j j} - r_{i_j}$. During a forward stage, the prices increase and profits decrease and during a reverse stage, the changes are inverse. An important property is that these updates maintain the ϵ-CS condition (7) for M, $\vec{p}$, and $\vec{r}$.

To avoid cycling of a FRAA, we must ensure an irreversible progress before switching between FAA and RAA stages. This is guaranteed if the switching is done only when the assignment is enlarged in the current FAA or RAA stage.

Although we consider in this paper only the symmetric version of the LSAP, a FRAA can be slightly modified to solve the asymmetric LSAPs [3]. This approach is even extended in paper [10] for several types of inequality contrained assignment problems such as multiassignment problems where persons may be assigned to several objects. Our distributed memory implementation of FRAA described in Sect. 5 can be easily extended for these problems.

4 Look-back auction algorithm

Our analysis of the behavior of the FAA for randomly generated instances revealed that the set of objects to which each person i has been assigned in the last few assignments does not change much as the algorithm proceeds. The explanation is that the persons fight in small groups for objects in price wars. Therefore, each person is assigned to just a relatively small number of objects during the execution of the FAA. This observation motivated our further research work on modifications of AAs.

The main bottleneck of existing AAs are the bidding rounds, in which the whole row of A corresponding to person i is searched for 2 best objects, i.e., 2 objects giving the highest benefits. Thus, each bidding round has complexity of $O(|I|n)$ local operations where I denotes the block defined in Sect. 3.1.

4.1 The idea of the look-back bidding

The basic idea *look-back AA (LBAA)* [13], is the following: The information about previous biddings of each person i is kept in a *working set* W_i and whenever possible, it is reused, instead of scanning the whole row i of matrix A as in the standard AAs.

First, we give more details about the working set data structure. A working set W_i of a person i is a set of $m \ll n$ pairs $(j, z(j))$, where j is the object number and $z(j) = a_{ij} - p_j$ is the value of its benefit.

In the first iteration in which person i performs bidding, each working set W_i is initialized to m pairs with the highest value of $z(j) = a_{ij} - p_j$ chosen among all n pairs $(j, z(j))$. For each initialized working set W_i, we define its *limit* $l_i = \min_{(j,z(j))\in W_i} z(j)$. Hence, l_i is the lowest benefit of objects in W_i. Since working sets undergo changes during the LBAA, the limits are used for checking that a given working set can be used for bidding instead of scanning the whole corresponding row i of A together with the whole current price vector $\vec{p}$.

Since the LBAA uses the same ϵ-scaling mechanism as we described in Sect. 3.2, we will focus our explanation on one bidding round of the LBAA.

One LBAA bidding round:
Assume that W_i is already initialized. Let $i \in R$ be a chosen person.

1. For each $(j, z(j)) \in W_i$, compute $\omega = a_{ij} - p_j$. If $\omega \geq l_i$, then replace $(j, z(j))$ with (j, ω) in W_i. Otherwise, remove $(j, z(j))$ from W_i.
2. If $|W_i| \geq 2$, we say that the look-back is *successful* and the person i performs a *look-back bid*:

(a) Let $(j_i', z(j_i')) \in W_i$ be a pair with the highest current benefit, i.e.,

$$z(j_i') = \max_{(j,z(j))\in W_i} \{z(j)\}$$

and let $(j_i'', z(j_i''))$ be a pair with the second highest current benefit, i.e.,

$$z(j_i'') = \max_{(j,z(j))\in W_i, j\neq j_i'} \{z(j)\}.$$

(b) Compute the bid value $b_{ij_i} = p_{j_i} + z(j_i') - z(j_i'') + \epsilon$ and submit the bid to object j_i'.

3. If $|W_i| < 2$, then the look-back *failed* and W_i must be reinitialized:
   (a) Reinitialize W_i by going through the whole row i (see the rules above) and compute the new value of the limit l_i.
   (b) Compute and submit the bid in the same way as in steps 2(a) and 2(b).

Lemma *The LBAA is semantically equivalent to the FAA.*

Proof In case of a failed look-back bidding, the LBAA performs the standard search in the whole row i of A and finds the same 2 best objects j_i' and j_i'' like the FAA.

Assume that the look-back bidding is successful and the LBAA has chosen objects j_i', j_i'' from W_i. Recall that object prices are not decreasing during the execution of the FAA and therefore, the benefits are not increasing. Each object whose benefit is less than l_i is removed from W_i. No object can be added to W_i during a successful look-back bidding. Hence, objects j_i' and j_i'' have the best two benefits in the whole row i.

Assume that a successful look-back bidding in the current working set W_i finds 2 different best objects j_i' and j_i'' than the FAA would find in the row i. Therefore, there exists an object j_i^*, $(j_i^*, z(j_i^*)) \notin W_i$ with $z(j_i^*) > z(j_i'')$ or even $z(j_i^*) > z(j_i')$. Then $z(j_i^*) > z(j_i'') \geq l_i$. Therefore, object j_i^* must have been included into W_i initially and must still belong to W_i, since the condition for removal in step 1 is not satisfied, which is a contradiction. □

Since the FAA is correct and finishes in a finite number of iterations with a correct solution, the LBAA is also correct, performs the same number of iterations, and finds the same solution as the FAA.

4.2 Related work

The first approach to reusing bids was described in book [6]. The author proposed to consider in bidding rounds also the third highest benefit. Our LBAA is a generalization of this idea, which corresponds to our working sets of size 3.

The idea of reusing information from previous scans of rows may be applied also for some other primal dual algorithms. We have found recently that the pseudoflow primal-dual algorithm described in [18] uses a similar idea of reusing already scanned values. The authors proposed a heuristic for their pseudoflow cost scaling assignment algorithm consisting in reusing 4 best values from the last double-push operation of

a given row. Hungarian and shortest augmenting path algorithms are not suitable for utilizing working set technique, since they perform search in dynamic sets of unscanned columns (not the whole row i). Some implementations of these methods use advanced data structures (e.g., d-heaps, see [1]) for a fast selection of the minimum from a set of columns, but they offer no reusability across multiple augmenting path constructions, since they shall be reinitialized for each new augmenting path.

4.3 Implementation issues and complexity of LBAA

An important issue of the implementation of the LBAA is to design an efficient data structure for handling the working sets. In the following text, we speak about ordering of elements of a working set. In fact, we need to introduce the relation of ordering to explain the initialization of working sets, but then the ordering of elements in working sets is unimportant. An initialization of W_i consists in finding m greatest values within an n-element unsorted array, where usually $n \gg m$, so that the first, the second, and the m-th greatest values can easily be found. We have used a *binary heap* mapped to an array with the root (the first element of the array) representing the m-th best value, i.e., the limit l_i of W_i.

Let us look now at the complexity of the LBAA bidding round. A look-back failure results in initialization of W_i which in the worst case consists in inserting n values into a binary heap. An insert into a binary heap of size m takes $O(\log m)$ operations, thus complexity of a failed LBAA bidding round is in $O(n \log m)$. On the other hand, a successful search in a working set has complexity $O(m)$ per one bidding round, since all elements of W_i must be updated by new prices. It follows that the real performance of the LBAA depends on the structure of the input instance.

As we will show in Sect. 6, the typical behavior of the LBAA is that the rate of successful look-back bids is low in the first few phases, in which the assignments fluctuate due to high value of ϵ, and increases with every further phase. To save a lot of useless initializations of working sets during the first few phases, the FAA can be applied to find a good starting vector of prices $\vec{p}$ and then the LBAA in the remaining scaling phases. It is also possible to find the starting vector $\vec{p}$ with the FRAA. However, we should note that it is difficult to combine the FRAA and the LBAA, because each reverse stage of the FRAA invalidates all working sets (the object prices are decreased).

While implementing the LBAA, we have put a lot of effort into keeping its mathematical equivalence with the FAA implementation. However, due to small differences of order 10^{-12} caused by rounding errors of arithmetic operations with `double` data, the LBAA has chosen in some cases a different best object j_i for bidding person i from W_i than the FAA would choose from the original row of the cost matrix A. However, since this behavior appears rarely, the number of bidding iterations has been just slightly different compared to the FAA. Nevertheless, the implementation of the LBAA is still correct.

5 Our parallel distributed memory AAs

In this section, we describe our implementation of the FAA, RAA and FRAA on a distributed memory parallel computer (DMPC). Then a description of a distributed

memory LBAA follows. In the further text, these 4 algorithms are denoted by DMFAA, DMRAA, DMFRAA, and DMLBAA, respectively, and a general distributed memory AA is denoted by DMAA.

In a DMAA, each processor works with its local part of the cost matrix A and synchronously executes the bidding iterations like in a sequential AA. Due to this distribution of A among the processors, the DMAAs are fully memory scalable. They can solve LSAP instances of any size by allocating enough processing nodes with sufficient memory.

Since one basic communication operation in a DMPC takes much more time than one basic CPU instruction, the performance of a DMAA is strongly dependent on the distribution of data structures. To achieve the best performance of DMAAs, we should use different distribution of data structures among the processors for the DMFAA, DMRAA, and DMFRAA.

Let P denote the number of processors. The input matrix A is distributed uniformly among local memories of processors. For dense instances each processor stores n^2/P elements of A. Analogically, for sparse instances each processor keeps locally n adjacency lists of persons and n adjacency lists of objects and each element $a_{ij} \neq 0$ is stored at the same processor as in the dense case. Although there are many possible distributions of matrix A among processors, only *columnwise*, *rowwise*, and *diagonalwise* distributions are useful for DMAAs (see Fig. 1).

All DMAAs work according to the same scheme: all processors execute ϵ-scaling phases, cycles, and bidding iterations in the same way as in sequential AAs, see Fig. 2. In contrast to the sequential AAs, the DMAAs split a bidding round to a *local bidding round* working just with locally stored data, followed by an *all-to-all reduction*, concluded with a *global bidding round*. This gives the same globally best bid as in sequential AAs.

At first, we explain distribution of data structures of the DMFAA. The data structures of the FAA consist of an input matrix A, a price vector $\vec{p}$, an assignment M,

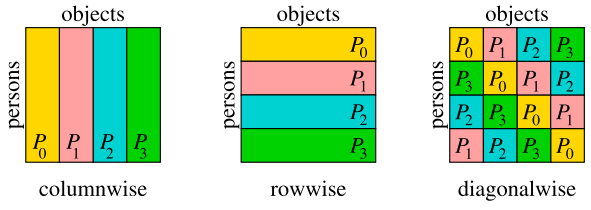

Fig. 1 Block columnwise, rowwise, and diagonalwise distributions of the input cost matrix A

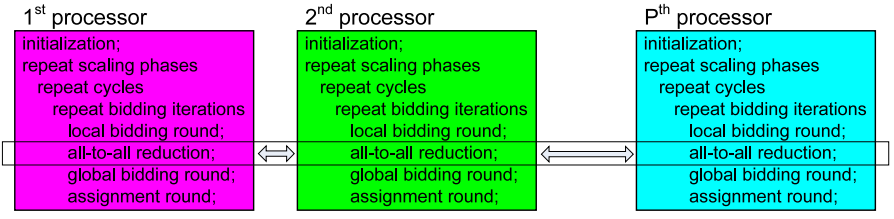

Fig. 2 Processors collaboration scheme in DMAAs

and a set of unassigned persons R. A should be implemented as a 2D array for dense instances or by an array of linked lists for sparse or fairly dense instances. A straightforward implementation of $\vec{p}$, $\vec{r}$, and M are arrays indexed by the object numbers. In the rest of the paper, we assume that set R is implemented as a linked list. Now we will show how these structures will be distributed among processors for the Gauss-Seidel, Jacobi, or hybrid DMFAA. Then we will give a detailed description of the DMFAA bidding.

The Gauss-Seidel DMFAA should use columnwise distribution of A among processors, since the rowwise distribution would result in just one processor working in each iteration. We used *cyclic columnwise* mapping of A, i.e., processor s is assigned column j iff $j \bmod P = s$. Similarly, the mapping of $\vec{p}$ and $\vec{r}$ is cyclic. We denote further set of columns j stored locally at processor s by J_s. The diagonalwise distribution could also be utilized, but it would not bring any advantage compared to the columnwise distribution.

In the bidding round of each iteration, all processors must agree on the same unassigned person i. This is easily implemented by replicating the list R on all processors and keeping it synchronized. Similarly, $\vec{p}$ and $\vec{r}$ and M are replicated and kept synchronized by all processors.

The Jacobi version of the DMFAA can use also columnwise distribution of A and the same discussion as for the Gauss-Seidel version of the DMFAA would follow. The rowwise distribution of A is also possible: a processor storing row i computes a bid of person i without the need of extra communication. This is inefficient, since a considerable portion of processors may stay idle while the rest may be overloaded. On a shared memory parallel computer, it is easy to achieve uniform load of processors, e.g., in paper [8] the authors described shared memory synchronous and asynchronous Jacobi versions of the FAA with a shared list of unassigned persons R. This load balancing approach cannot be used in the DMFAA. We have one interesting observation which was also published in papers [11, 28]—the number of unassigned persons $|R|$ is very small in most of FAA iterations. In fact, the less persons are unassigned, the more iterations it takes to assign a next unassigned person. If $|R| < P$, then some processors stay idle.

Neither Jacobi nor Gauss-Seidel approach provide the optimality. For example, too many persons involved in one bidding iteration may worsen the performance due to conflicts among bidders, e.g., during price wars. To achieve the best performance, we used hybrid DMFAA with adaptive size of the block of bidders $b = |I|$.

Even though other approaches might be useful, we have implemented the adaptivity based on a simple *linear interpolation*. The block size b in iteration k is computed from the number of unassigned persons $|R|$ at the beginning of the corresponding cycle as follows. Let l_t be a threshold on $|R|$. If $|R| < l_t$, then hybrid DMFAA proceeds in the Gauss-Seidel mode, i.e., $b = 1$. If $|R| \geq l_t$ then

$$b = b_e + (b_b - b_e)(|R| - l_t)/(n - l_t), \tag{8}$$

where b_b is the initial size of a block (i.e., n), and b_e is the block size for $|R| = l_t$.

In the DMFAA bidding, each processor computes at first partial bids for each bidder i by searching locally stored columns J_s. Then these bids are evaluated by parallel all-to-all reduction. So, each processor knows globally best bids for the chosen group

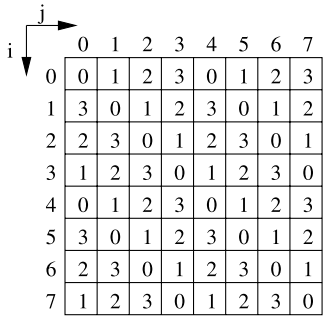

Fig. 3 The cyclic diagonalwise mapping of $A_{8\times 8}$ to 4 processors in the DMFRAA. The number in each cell $a_{i,j}$ determines the number of the processor in whose memory a_{ij} is stored

of bidders. During the assignment round, all processors update locally primal and dual variables $\vec{p}$, $\vec{r}$, and M together with the set of unassigned bidders for next iterations. So, these variables have the same value on all processors during the whole algorithm.

Since the DMRAA is a transposed DMFAA, we leave out its description here since it can be obtained by analogy.

The DMFRAA alternates the DMFAA with the DMRAA. One possibility is therefore to use the columnwise or rowwise distribution. However, forward and reverse stages are not treated symmetrically and, as a result, such DMFRAA suffers from the problems discussed before for the possible rowwise distribution f A in the Jacobi DMFAA. Therefore we use *cyclic diagonalwise mapping*, i.e., matrix A is uniformly distributed as depicted on Fig. 3: It is divided into P^2 square submatrices A_{kl}, $0 \leq k, l \leq P-1$, where $a_{ij} \in A_{kl}$ iff $i \bmod P = k$ and $j \bmod P = l$. Processor s stores locally P submatrices A_{kl} such that $(k+s) \bmod P = l$. The other data structures, i.e., assignment vectors M_o and M_p, price vector $\vec{p}$, profit vector $\vec{r}$, the lists R and S are replicated in local memories of all processors and kept synchronized.

As discussed for the DMFAA, a straightforward implementation of price vector $\vec{p}$ (profit vector $\vec{r}$) is an array indexed by object numbers j (person numbers i, respectively). For M, there should be two arrays M_o and M_p. M_o is used in the forward stages and is indexed by object numbers j and M_p is used in reverse stages and is indexed by person numbers i. Also, lists R for forward stages and S for reverse stages should be maintained. All these structures are replicated on each processor and kept synchronized.

In the DMLBAA, A is mapped to the processors in the same way as in the DMFAA, i.e., by cyclic columnwise mapping, and each processor has a copy of M, $\vec{p}$, and R. Additionally, each processor maintains for each person i his working set W_i. The same would apply if the DMLBAA was implemented with cyclic diagonalwise mapping, but this mapping would not produce any advantage comparing to the cyclic columnwise mapping.

Since all processors keep the content of all working sets synchronized a successful search in a working set is a *purely local operation*. The computational complexity of a DMLBAA successful search is therefore only $O(m)$ and no collective communication operation is needed! This is the main source of performance improvement of the DMLBAA w.r.t. the DMFAA and the DMFRAA. A look-back failure fires local

searches of m best elements among $\frac{n}{P}$ local elements of matrix A, followed by an all-to-all reduction of the working sets.

Compared to the Gauss-Seidel approach for the DMLBAA, the Jacobi and hybrid approach cause the following problem: if the same block contains a bidder whose look-back bid succeeds and a bidder whose look-back bid fails, a bidding phase communication must be performed. Therefore, we propose to modify the cycle of the Jacobi and hybrid DMLBAA to perform Gauss-Seidel auctions with the bidders from the list R for which the look-back bidding is successful and then perform one Jacobi bidding round or more hybrid bidding rounds with unsuccessful bidders.

One of the main differences comparing to the LBAA is that each working set W_i consists of triples $(j, z(j), a_{ij})$ instead of pairs $(j, z(j))$. The reason is that each processor performing DMLBAA needs to access a_{ij} when updating the working set W_i (computing difference $a_{ij} - p_j$). In general, a_{ij} can be stored on another processor therefore in order to avoid additional communication the corresponding cost a_{ij} shall be appended to each element of the local working set W_i submitted to the all-to-all reduction communication operation. We implemented the all-to-all reduce operation using `MPI_Allreduce` function. Each message contains an array of at most $|I|$ working sets (for Gauss-Seidel $|I| = 1$), each represented by m triples $(j, z(j), a_{ij})$. Because the whole vector of prices $\vec{p}$ is kept synchronized, the difference $a_{ij} - p_j$ is evaluated correctly. It is obvious that a greater value of m increases the latency of the reduction operation.

6 Experimental results

We have tested the AAs on several types of dense and sparse input problems that are used in literature for comparison of the LSAP algorithms. Let C be an integer parameter.

- RAND—dense random instance. Numbers a_{ij} are generated by an uniform random generator over integer interval $\langle 1, \ldots, C \rangle$.
- GEOM—geometrical instance. First, $2n$ random points are generated by uniform random generator in a 2D square of size $\langle 1, \ldots, C \rangle \times \langle 1, \ldots, C \rangle$. Then each a_{ij} is assigned the integer equal to the Euclidean distance between points i and j.
- IxJ—Machol-Wien instance, in which $a_{ij} = ijC$ [24]. In our experiments, we use $C = 1$.
- SPARSE_RAND—sparse random instance. First, a random permutation π of sequence $[0, 1, \ldots, n-1]$ is generated and edges (i, π_i) are added to the graph by setting a_{ij} to a integer generated by a uniform generator over integer interval $\langle 1, \ldots, C \rangle$. Then for each person, $n \cdot D/100 - 1$ edges with random costs are added to the graph in the same way, where D is the requested density in percents. Finally, one additional edge is added to each object incident to only 1 edge.

All random values were generated by function `drand48()`[2] using different random seeds for each run. Parameter C of each generator was 1000 for *low-cost instances* and 100 000 for *high-cost instances* (except for IxJ).

[2]The `glibc` implementation of a 48-bit random number generator.

Table 1 Scaling parameters used in the following experiments with scaled AAs

	ϵ_b	Θ
high-cost RAND	$C/5^5$	5
low-cost RAND	$C/5^4$	5
GEOM	$\max\{a_{ij}\}/5$	5
IxJ	$n^2/5$	5
high-cost SPARSE_RAND	$C/5^4$	5
low-cost SPARSE_RAND	$C/5^3$	5

Our implementation of AAs supports 2 types of storing input instances in the memory. A dense instance is stored as an $n \times n$ array of integers with row major ordering. We have observed that solving a dense instance by the FRAA can be significantly influenced by cache misses during reverse stages. Using an extra $n \times n$ array with column major ordering for reverse stages of the FRAA solves this problem, but the space complexity doubles.

A sparse instance is stored as n adjacency lists of persons L_i in the FAA and both adjacency lists of persons L_i and objects L_j in the FRAA. Adjacency lists contain only nonzero elements of A, i.e., each element $a_{ij} \neq 0$ is in both L_i and L_j. We stress that this has a significant impact on the semantics of an AA, since scanning rows in bidding phases is performed differently. In case of dense matrix implementation, the whole row including possible zero elements a_{ij} is scanned by an AA, but if the matrix is implemented as sparse, only the adjacency list of elements is scanned with skipping possible zeros.

Our DMFAA uses cyclic columnwise mapping and DMFRAA cyclic diagonalwise mapping of instances to the memory of each processor, as was described in Sect. 5. It means that for dense instances each processor stores $n \times n/P$ elements of A. Analogically, for sparse instances each processor keeps locally n adjacency lists of persons and n adjacency lists of objects and each element $a_{ij} \neq 0$ is stored at the same processor as in the dense case. In cases that this mapping scheme overloads some processors, a different scheme can be used, since the ordering of row scanning is not important.

Our experiments confirmed behavior of AAs described in paper [14]. Therefore, we set the ϵ-scaling parameters of our AAs similarly. However, for high-cost (low-cost, resp.) RAND instances, we found that it pays off to skip first 4 (5, resp.) scaling phases. Table 1 summarizes values of scaling parameters used in experiments with scaled AAs. Recall that ϵ_b denotes the value of ϵ in the first phase and Θ denotes the divisor of ϵ in each phase.

The cluster for evaluation of our implementation consisted of 16 nodes with Linux OS connected with a Myrinet network. Each node had Intel Pentium Celeron processor at 1.7 GHz, 128 kB cache with 256 MB memory. We have implemented the algorithms in C++ using Myricom MPICH-GM implementation of MPI-1. The code was compiled with g++ using the option `-O2`.

6.1 LBAA

In this section, we will illustrate behavior of the sequential Gauss-Seidel LBAA.

Table 2 The dependence of the performance of the LBAA on the value of m for high-cost RAND and GEOM instances of size $n = 5000$

	RAND		GEOM	
m	time	fails	time	fails
7	1.26 s	13.5%	42.59 s	71.1%
15	0.73 s	6.3%	29.60 s	44.4%
31	0.93 s	5.6%	27.38 s	30.4%
63	0.92 s	5.6%	27.28 s	30.4%

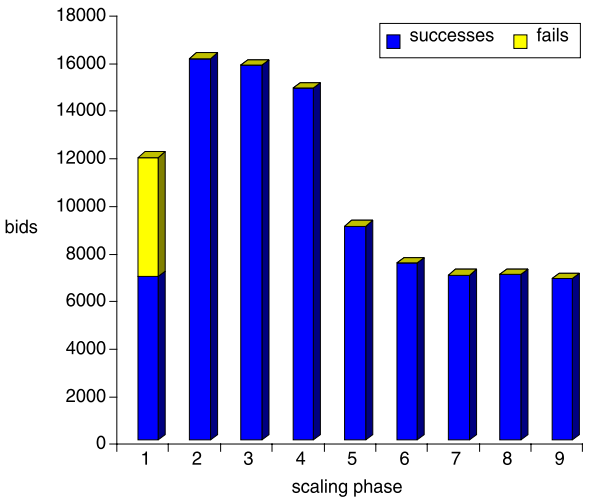

Fig. 4 A look-back bidding statistics for a high-cost RAND input instance of size $n = 5000$

At first, we have run several experiments exploring the influence of the parameter m (the size of working sets) on the performance of the LBAA for a fixed size n and a fixed type of input instances. Table 2 illustrates that larger values of m reduce the ratio of failures, but not necessarily the time. The reason is that increasing m leads to increasing complexity of initialization of working sets. Even though the value of m achieving the minimal time depends on n, we have set it up constant for simplicity in the current implementation of the LBAA. All further described experiments with the LBAA were run with $m = 31$.

In the next set of experiments, we have run the LBAA for various types of instances and tested the rate of successful look-back bids. Figure 4 illustrates how the number of successful and failed look-back bids per each scaling phase is evolving, as the algorithm proceeds for a high-cost RAND instance. The first column corresponds to the first scaling phase ($\epsilon = C/5$), the last column corresponds to the last scaling phase (with $\epsilon \leq \frac{1}{n+1}$). A very interesting observation is that the LBAA performed only n unsuccessful look-back bids necessary for the initialization of all working sets in the first scaling phase and then all look- back bids succeeded. These unsuccessful bids correspond to compulsory initialization of working sets during the first n bidding rounds. We have run many similar experiments also for low-cost RAND instances and observed the same behavior.

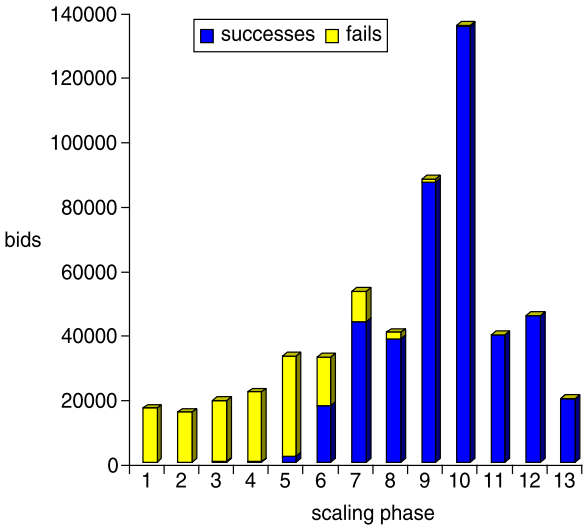

Fig. 5 A look-back bidding statistics for a high-cost GEOM input instance of size $n = 5000$

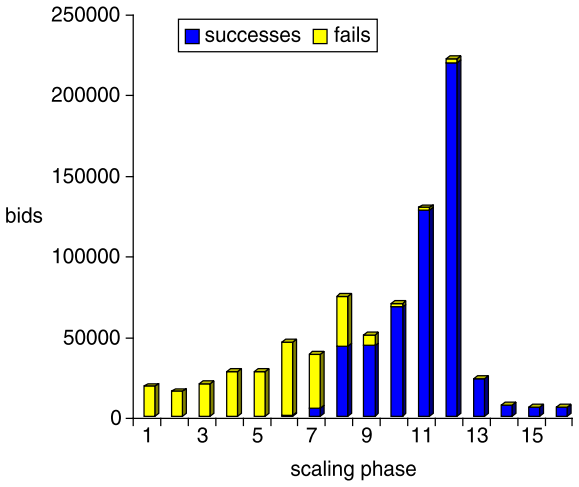

Fig. 6 A look-back bidding statistics for an IxJ input instance of size $n = 5000$

We show here also results for the types of instances considered hard to solve by any LSAP algorithm. Specifically, results for high-cost GEOM instances are on Fig. 5 and for an IxJ instance on Fig. 6. In both cases, the ratio of successful look-back bids in the first few scaling phases was very low, but it increased up to almost 100% in final phases. The average ratio of successful look-back bids was 76.4% for the GEOM instances and 70.2% for the IxJ instance. We have also observed that for low-cost GEOM instances, this ratio is smaller. This observation corresponds to the fact that in these instances the same number of 2D points is compressed into a smaller 2D area and therefore, the persons and objects are more influenced by each other.

It is also worth noticing that the first phases for both GEOM and IxJ instances were relatively short and the LBAA performed most of the bids in the middle phases.

We will show further in Sect. 6.2 that the performance is improved if the LBAA is combined with the FAA. The FAA is run for first few phases to obtain a good starting price vector $\vec{p}$ and then the computation is completed with the LBAA. We denote this hybrid of FAA and LBAA by LBFAA.

6.2 Evaluation of sequential LSAP algorithms

In the next series of experiments, we have compared the performance of the LBAA with other LSAP algorithms for various types of dense instances. More specifically, we have compared it with our implementation of the sequential FAA and FRAA, which we obtained by rewriting FORTRAN codes available at `http://www.mit.edu:8001/people/dimitrib/auction.txt` into C++ and with C implementation of algorithm JV for dense instances obtained from `http://www.magiclogic.com/assignment.html`. We do not compare the LBAA with the JV for sparse instances, since the author provides only a Pascal code that we were not able to incorporate in our testing framework. The reason for the comparison with the unscaled FRAA is that paper [14] reported that it outperformed scaled AAs for RAND and SPARSE_RAND instances. In the following text, we consider only Gauss-Seidel AAs.

We present our results in several tables. Each table gives the average time τ in seconds, the average number of bids β (for AAs), and the average number of failed look-back bids ϕ (for LBAA). The corresponding standard deviations are denoted by Δ_τ, Δ_β, and Δ_ϕ respectively. Ten different assignment problems were generated for each experiment. In order to demonstrate the behavior of the FRAA not influenced by cache miss effects, we report also times of dense implementation with the extra column-major matrix for the reverse stage. We denote it by *FRAA2* and the corresponding time by τ_2.

Table 3 contains our results for high-cost RAND instances and Table 4 for low-cost RAND instances. As we see, the scaled LBAA was the fastest for both types of RAND instances, it was 6 times faster than the FAA and also much faster than JV and the number of missed look-back bids was just slightly bigger than the lower bound n.

Table 3 A comparison of the performance of sequential AAs for high-cost RAND instances

		n		
		1000	3000	5000
Scaled	$\tau(\Delta_\tau)$	0.41 (0.12)	2.62 (1.15)	5.00 (0.12)
FAA	$\beta(\Delta_\beta)$	33 324 (9782)	76 098 (33 670)	87 842 (2126)
Unscaled	$\tau(\Delta_\tau)$	0.23 (0.04)	2.50 (0.50)	7.23 (0.91)
FRAA	$\tau_2(\Delta_{\tau_2})$	0.07 (0.01)	0.60 (0.03)	–
	$\beta(\Delta_\beta)$	4837 (419)	15 574 (857)	26 198 (1003)
Scaled	$\tau(\Delta_\tau)$	0.11 (0.01)	0.46 (0.05)	0.93 (0.01)
LBAA	$\beta(\Delta_\beta)$	33 422 (9785)	76 480 (33 052)	89 649 (3007)
	$\phi(\Delta_\phi)$	1000 (0)	3000 (0)	5000 (0)
JV	$\tau(\Delta_\tau)$	0.16 (0.01)	2.42 (0.12)	7.90 (0.57)

Table 4 A comparison of the performance of sequential AAs for low-cost RAND instances

		n		
		1000	3000	5000
Scaled	$\tau(\Delta_\tau)$	0.20 (0.06)	4.1 (0.50)	14.70 (0.63)
FAA	$\beta(\Delta_\beta)$	16 538 (4933)	117 238 (14 348)	256 140 (11 174)
Unscaled	$\tau(\Delta_\tau)$	0.60 (0.65)	3.09 (0.80)	13.17 (4.65)
FRAA	$\tau_2(\Delta_{\tau_2})$	0.10 (0.06)	0.60 (0.06)	–
	$\beta(\Delta_\beta)$	6053 (5820)	15 607 (1716)	32 918 (5967)
Scaled	$\tau(\Delta_\tau)$	0.09 (0.01)	0.51 (0.02)	1.35 (0.02)
LBAA	$\beta(\Delta_\beta)$	16 400 (4745)	113 480 (9843)	255 040 (12 145)
	$\phi(\Delta_\phi)$	1000 (0)	3025 (19)	5688 (25)
JV	$\tau(\Delta_\tau)$	0.14 (0.01)	2.3 (0.09)	5.96 (0.21)

Table 5 A comparison of the performance of sequential AAs for high-cost GEOM instances

		n		
		1000	3000	5000
Scaled	$\tau(\Delta_\tau)$	0.88 (0.31)	10.80 (2.77)	31.70 (7.5)
FAA	$\beta(\Delta_\beta)$	71 680 (26 152)	309 127 (79 950)	545 366 (129 762)
Scaled	$\tau(\Delta_\tau)$	3.99 (0.79)	94.47 (22.76)	318.25 (65.34)
FRAA	$\tau_2(\Delta_{\tau_2})$	1.06 (0.14)	16.32 (3.24)	–
	$\beta(\Delta_\beta)$	74 920 (10 533)	425 827 (85 248)	815 194 (133 428)
Scaled	$\tau(\Delta_\tau)$	1.12 (0.08)	9.40 (0.93)	27.32 (2.85)
LBAA	$\beta(\Delta_\beta)$	66 832 (14 546)	305 556 (70 361)	598 911 (273 198)
	$\phi(\Delta_\phi)$	17 372 (1392)	85 607.7 (8238)	181 950 (18 418)
Scaled	$\tau(\Delta_\tau)$	0.55 (0.07)	6.85 (0.77)	20.39 (2.77)
LBFAA	$\beta(\Delta_\beta)$	72 029 (19 222)	319 753 (74 540)	632 469 (205 268)
	$\phi(\Delta_\phi)$	17 869 (1206)	89 490 (7962)	178 068 (20 291)
JV	$\tau(\Delta_\tau)$	1.44 (0.03)	25.37 (0.67)	116.66 (2.06)

The FRAA2 was nearly as fast as the LBAA for small and medium instances, but it could not be used for larger instances due its memory limitation. Although the complexity of all AAs is dependent on the cost range C (see Sect. 3.2), the low-cost RAND instances, due to more price war conflicts, result in worse performance of AAs than high-cost RAND instances.

A comparison for GEOM instances is shown in Tables 5 and 6. Since GEOM instances are much harder to solve than RAND instances, we compared only scaled AAs. The LBAA again achieved the best results for high-cost GEOM instances among all tested algorithms, but not so significantly as for RAND instances. We explain this by the fact that the ability of the LBAA to reuse bids is worse than for RAND instances due to the structure of GEOM instances. The low-cost instances, as discussed above, do not allow to utilize fully the values in working sets and therefore, the LBAA is even slower than the scaled FAA. We can also see that the LBFAA

Table 6 A comparison of the performance of sequential AAs for low-cost GEOM instances

		n		
		1000	3000	5000
Scaled	$\tau(\Delta_\tau)$	0.87 (0.23)	11.98 (2.76)	39.71 (6.00)
FAA	$\beta(\Delta_\beta)$	69 299 (18 187)	342 819 (79 443)	689 438 (104 844)
Scaled	$\tau(\Delta_\tau)$	4.19 (0.74)	91.78 (18.84)	368.05 (69.00)
FRAA	$\tau_2(\Delta_{\tau_2})$	1.01 (0.10)	15.534 (1.86)	–
	$\beta(\Delta_\beta)$	72 286 (7169)	420 110 (74 444)	879 714 (82 761)
Scaled	$\tau(\Delta_\tau)$	1.11 (0.07)	14.783 (2.48)	47.48 (8.24)
LBAA	$\beta(\Delta_\beta)$	62 690 (10 180)	332 993 (65 027)	686 506 (118 915)
	$\phi(\Delta_\phi)$	17 212 (924)	137 842 (23 275)	324 849 (57 493)
Scaled	$\tau(\Delta_\tau)$	0.56 (0.09)	11.37 (2.13)	41.15 (6.77)
LBFAA	$\beta(\Delta_\beta)$	65 752 (13 133)	329 518 (59 405)	706 117 (106 122)
	$\phi(\Delta_\phi)$	22 419 (1682)	144 647 (20 280)	343 190 (45 692)
JV	$\tau(\Delta_\tau)$	0.75	19.38	70.35

Table 7 A comparison of the performance of sequential AAs for IxJ instances

		n		
		1000	3000	5000
Scaled	τ	0.82	10.74	32.46
FAA	β	51 188	247 597	450 543
Scaled	τ	4.90	98.43	329.14
FRAA	τ_2	3.22	45.99	–
	β	86 535	411 250	800 962
Scaled	τ	3.7	43.82	133.872
LBAA	β	51 188	247 597	450 543
	ϕ	26 135	118 288	222 740
Scaled	τ	0.74	8.33	24.35
LBFAA	β	51 188	247 597	450 543
	ϕ	41 340	158 468	255 441
JV	τ	25.13	129.98	553.22

(the combination of the FAA and the LBAA), yields a performance improvement compared to the LBAA.

Table 7 summarizes our results for IxJ instances which are considered to be one of the most difficult LSAP instances for any algorithm. The measured times confirmed this assumption for all types of LSAP algorithms.

Tables 8 and 9 show the results for SPARSE_RAND instances with degree 5%, i.e., each person has links to 5% of objects. We did experiments also with fixed degree SPARSE_RAND instances, keeping the degree of each person equal to 300 and changing the size of instances, and, we received similar results as shown in Tables 8 and 9. For both 5% and fixed degree instances the unscaled FRAA was the fastest algorithm, since the reverse phase shortens price wars of forward phase. But, for

Table 8 A comparison of the performance of sequential AAs for high-cost 5% SPARSE_RAND instances

		n		
		5000	8000	11 000
Scaled	$\tau(\Delta_\tau)$	0.93(0.24)	2.26 (0.52)	4.25 (0.68)
FAA	$\beta(\Delta_\beta)$	137 426 (21 892)	215 118 (53 266)	249 250 (41 983)
Unscaled	$\tau(\Delta_\tau)$	0.18 (0.01)	0.50 (0.23)	1.23 (0.06)
FRAA	$\beta(\Delta_\beta)$	28 875 (2742)	56 364 (28 614)	66 450 (3133)
Scaled	$\tau(\Delta_\tau)$	0.52 (0.04)	0.99 (0.11)	1.53 (0.10)
LBAA	$\beta(\Delta_\beta)$	137 190 (21 721)	216 647 (53 816)	250 778 (43 514)
	$\phi(\Delta_\phi)$	5000 (0)	8000 (0)	11 000 (0)

Table 9 A comparison of the performance of sequential AAs for low-cost 5% SPARSE_RAND instances

		n		
		5000	8000	11 000
Scaled	$\tau(\Delta_\tau)$	0.62 (0.03)	1.69 (0.08)	4.68 (0.30)
FAA	$\beta(\Delta_\beta)$	90 352 (3742)	164 633 (8565)	283 639 (19 509)
Unscaled	$\tau(\Delta_\tau)$	0.17 (0.05)	0.64 (0.62)	6.29 (6.41)
FRAA	$\beta(\Delta_\beta)$	29 783 (10 967)	78 767 (78 832)	353 064 (371 400)
Scaled	$\tau(\Delta_\tau)$	0.38 (0.01)	0.87 (0.02)	1.57 (0.05)
LBAA	$\beta(\Delta_\beta)$	89 528 (4422)	165 588 (9073)	280 062 (21 950)
	$\phi(\Delta_\phi)$	5000 (0.3)	8024 (7)	11 495 (67)

low-cost instances, it was quite unstable (some instances were computed slowly—with long price wars, while other instances very fast). The LBAA was slower but it showed better stability than the FRAA across all generated sparse instances.

With decreasing degree the LBAA becomes even more slower comparing to the FRAA and therefore is not worth to consider it for solving very sparse problems. The reason is that the number of related objects per person becomes more and more closer to the size of working set m and therefore benefit of using working set is continuously lost. In the extreme, where it decrease to m, the LBAA mutate into the FAA with extra cost for maintaining working sets and therefore we can expect similar performance as the FAA.

6.3 Comparison of distributed AAs

First, we have investigated the speedup of DMAAs. We should not expect any real speedup, since DMAAs iterate in the same way as their sequential counterparts. The latency of collective communication operations in DMAAs on most real machines is higher than the time of local computation saved by employing more processors to speed up the bidding evaluation. Our experiments confirmed this expectation, as shown in Table 10 for the scaled DMFAA, unscaled DMFRAA, and scaled DML-BAA, computing high cost RAND instances of size $n = 5000$. For other types of instances, we observed even worse slowdown. Interesting speedup values can be seen

Table 10 Parallel times of AAs for high cost RAND instances with size $n = 5000$

	$\tau(\Delta_\tau)$		
	Scaled	Unscaled	Scaled
P	DMFAA	DMFRAA	DMLBAA
1	5.00 (0.12)	7.23 (0.91)	0.93 (0.01)
2	4.31 (0.12)	4.15 (0.36)	0.96 (0.01)
4	5.14 (0.15)	2.89 (0.15)	0.94 (0.04)
8	6.64 (0.18)	2.72 (0.53)	1.00 (0.2)

Table 11 A comparison of the performance of distributed AAs for high-cost RAND instances

		n		
		10 000 ($P = 4$)	15 000 ($P = 8$)	20 000 ($P = 15$)
Scaled	$\tau(\Delta_\tau)$	18.68 (0.50)	55.20 (1.60)	131.41 (4.64)
DMFAA	$\beta(\Delta_\beta)$	214 041 (5761)	372 024 (10 975)	522 822 (17 013)
Unscaled	$\tau(\Delta_\tau)$	11.82 (3.9)	39.19 (31.15)	68.92 (65.31)
DMFRAA	$\beta(\Delta_\beta)$	56 913 (7141)	159 789 (89 125)	225 212 (204 058)
Scaled	$\tau(\Delta_\tau)$	2.66 (0.02)	5.53 (0.12)	11.99 (0.30)
DMLBAA	$\beta(\Delta_\beta)$	210 966 (6433)	368 757 (10 565)	513 835 (19 436)
	$\phi(\Delta_\phi)$	10 049 (13)	16 584 (267)	27 292 (586)

only for the DMFRAA, but not in general. The first reason is that the DMFRAA faces less number of cache misses per processor during the reverse stage than the sequential FRAA. The second reason is that for high-cost RAND instances, the unscaled DMFRAA is able to solve majority of instances without price wars, but for the other types, this is not true, see Sect. 6.2 for real behavior of the unscaled FRAA on various types of instances. We can conclude that our distributed memory AAs are not able to achieve any reasonable speedup for instances small enough to fit into the memory of a single processor.

However, our distributed memory AAs are memory scalable. They are to be used for solving large instances exceeding memory of a single computer.

Similarly to the previous section, we compare distributed Gauss-Seidel AAs running RAND and SPARSE_RAND instances.

Tables 11 and 12 compare distributed memory AAs for high-cost and low-cost RAND instances. We have chosen the size of test instances to fit into memory of 4, 8, and 15 processors, respectively. The DMLBAA is superior among all tested algorithms. For high-cost instances, it achieves almost optimal rate of information reusability (the lower bound for the number of fails is n, the compulsory fails during the initialization process), e.g., for $n = 20\,000$ only 5% of bids failed.

Comparison of DMAAs on high-cost SPARSE_RAND instances is given in Table 13 and for low-cost instances in Table 14. For both, the DMLBAA achieves almost 100% rate of information reusability, and thus needs the least communication among all compared DMAAs. For this reason, it is for high-cost instances more than twice

Table 12 A comparison of the performance of distributed AAs for low-cost RAND instances

		n		
		10 000 ($P = 4$)	15 000 ($P = 8$)	20 000 ($P = 15$)
Scaled	$\tau(\Delta_\tau)$	42.58 (6.20)	81.02 (5.08)	184.63 (3.72)
DMFAA	$\beta(\Delta_\beta)$	511 406 (75 306)	536 610 (37 037)	723 509 (13 305)
Unscaled	$\tau(\Delta_\tau)$	39.04 (8.79)	53.74 (12.85)	74.68 (14.38)
DMFRAA	$\beta(\Delta_\beta)$	127 453 (25 117)	175 377 (40 886)	216 553 (38 771)
Scaled	$\tau(\Delta_\tau)$	4.82 (0.60)	11.16 (0.18)	24.87 (0.26)
DMLBAA	$\beta(\Delta_\beta)$	490 344 (78 322)	553 930 (25 014)	708 047 (15 435)
	$\phi(\Delta_\phi)$	15 115 (1514)	32 060 (167)	56 977 (252)

Table 13 A comparison of the performance of distributed AAs for high-cost 5% SPARSE_RAND instances

		n		
		16 000 ($P = 4$)	23 000 ($P = 8$)	30 000 ($P = 15$)
Scaled	$\tau(\Delta_\tau)$	20.78 (1.05)	44.59 (1.41)	86.68 (1.72)
DMFAA	$\beta(\Delta_\beta)$	375 480 (20 679)	558 603 (17 083)	743 181 (12 998)
Unscaled	$\tau(\Delta_\tau)$	5.72 (0.17)	11.97 (0.62)	21.83 (0.91)
DMFRAA	$\beta(\Delta_\beta)$	102 554 (2918)	148 414 (67 007)	189 223 (8327)
Scaled	$\tau(\Delta_\tau)$	3.40 (0.06)	5.92 (0.11)	9.32 (0.11)
DMLBAA	$\beta(\Delta_\beta)$	375 847 (21 265)	559 059 (9880)	757 048 (21 587)
	$\phi(\Delta_\phi)$	16 000 (10)	23 010 (22)	30 247 (690)

Table 14 A comparison of the performance of distributed AAs for low-cost 5% SPARSE_RAND instances

		n		
		16 000 ($P = 4$)	23 000 ($P = 8$)	30 000 ($P = 15$)
Scaled	$\tau(\Delta_\tau)$	25.59 (2.22)	69.29 (9.88)	148.519 (19.60)
DMFAA	$\beta(\Delta_\beta)$	466 709 (40 235)	855 609 (128 901)	1 274 620 (151 769)
Unscaled	$\tau(\Delta_\tau)$	42.68 (61.62)	39.26 (58.27)	309.44 (728.29)
DMFRAA	$\beta(\Delta_\beta)$	804 734 (1 180 850)	496 004 (736 843)	2 642 530 (6 172 700)
Scaled	$\tau(\Delta_\tau)$	4.19 (0.17)	10.09 (0.54)	18.17 (0.61)
DMLBAA	$\beta(\Delta_\beta)$	459 769 (37 924)	852 354 (109 869)	1 280 620 (145 777)
	$\phi(\Delta_\phi)$	21 288 (531)	41 416 (1009)	59 667 (593)

faster than the DMFRAA and for low-cost instances, it is even 8 times faster than the DMFAA. For low-cost SPARSE_RAND instances, the DMFRAA reveals the same type of instability as the FRAA, as we have discussed in Sect. 6.2.

7 Conclusion

We have designed and implemented a general bidding technique for the FAAs reusing information from previous bids, called look-back bidding. The corresponding algorithm is called the look-back AA. We have also designed and implemented a library of sequential and distributed memory AAs: forward, forward-reverse, and look-back. In all cases, we have performed extensive testing on several types of input instances of the LSAP.

A very interesting observation is that the look-back AA is able to reuse information from the previous bids with high efficiency for all tested types of input instances. More specifically, for RAND instances it is almost 100% and for GEOM and IxJ instances it is higher than 70%.

The results also show that the look-back AA solved sequentially nearly all dense types of instances faster than other evaluated algorithms. There were two exceptions. The scaled forward AA was slightly faster for low-cost GEOM and IxJ instances. The unscaled FRAA was faster for SPARSE_RAND instances, but it was unstable.

The superiority of the look-back bidding appears even more in the case of distributed memory AAs. Distributed memory look-back AAs perform most computation locally and this allows them to save a significant amount of communication in comparison to the distributed memory implementations of the FAAs and FRAAs. For this reason, the distributed memory look-back AA outperforms the other distributed memory AAs significantly. In some cases, it is several times faster. Moreover, all our distributed memory AAs are fully memory scalable. The performance and efficiency can be finely tuned by choosing exactly the least number of processors needed to distribute the input instance data (the cost matrix and the price and profit vectors) into local memories of processors. Our library is fully portable, since it is written in C++ using MPI.

References

1. Balas, E., Miller, D., Pekny, J., Toth, P.: A parallel shortest path algorithm for the assignment problem. J. ACM **38**, 985–1004 (1991)
2. Belongie, S., Malik, J., Puzicha, J.: Shape matching and object recognition using shape contexts. IEEE Trans. Pattern Anal. Mach. Intel. **24**(24), 509–522 (2002)
3. Bertsekas, D., Castañon, D.: A forward/reverse auction algorithm for asymmetric assignment problems. Comput. Optim. Appl. **1**(3), 277–297 (1992)
4. Bertsekas, D.P.: A distributed algorithm for the assignment problem. Laboratory for Information and Decision Systems Working Paper, (M.I.T.), March 1979
5. Bertsekas, D.P.: A new algorithm for the assignment problem. Math. Program. **21**, 152–171 (1981)
6. Bertsekas, D.P.: Linear Network Optimization: Algorithms and Codes. MIT Press, Cambridge (1991)
7. Bertsekas, D.P.: Network Optimization: Continuous and Discrete Models. Athena Scientific, Belmont (1998)
8. Bertsekas, D.P., Castañon, D.A.: Parallel synchronous and asynchronous implementations of the auction algorithm. Parallel Comput. **17**(6–7), 707–732 (1991)
9. Bertsekas, D.P., Castañon, D.A.: Parallel asynchronous hungarian methods for the assignment problem. ORSA J. Comput. **5**, 261–274 (1993)
10. Bertsekas, D.P., Castañon, D.A., Tsaknakis, H.: Reverse auction algorithm and the solution of inequality constrained assignment problems. SIAM J. Optim. **3**, 268–299 (1993)

11. Brady, M., Jung, K.K., Nguyen, H.T., Raghavan, R., Subramonian, R.: The assignment problem on parallel architectures. In: Network Flows and Matching. DIMACS, pp. 469–517. American Mathematical Society, Providence (1993)
12. Burkard, R.E., Çela, E.: Linear assignment problems and extensions. In: Handbook of Combinatorial Optimization, pp. 75–149. Kluwer Academic, Dordrecht (1999)
13. Buš, L., Tvrdík, P.: Look-back auction algorithm for the assignment problem and its distributed memory implementation. In: Proceedings of the 15th IASTED International Conference Parallel and Distributed Computing and Systems, pp. 551–556. Acta Press, Anaheim (2003)
14. Castañon, D.A.: Reverse auction algorithms for assignment problems. In: Network Flows and Matching. DIMACS, pp. 407–429. American Mathematical Society, Providence (1993)
15. Duff, I.S., Koster, J.: On algorithms for permuting large entries to the diagonal of a sparse matrix. SIAM J. Matrix Anal. Appl. **22**(4), 973–996 (2001)
16. Frieze, A., Galbiati, G., Maffioli, F.: On the worst-case performance on some algorithms fo the asymmetric traveling salesman problem. Networks **12**, 23–39 (1982)
17. Glover, F., Gutin, G., Yeo, A., Zverovich, A.: Contruction heuristics and domination analysis for the asymmetric tsp. Eur. J. Oper. Res. **129**, 555–568 (2001)
18. Goldberg, A., Kennedy, R.: An efficient cost scaling algorithm for the assignment problem. Math. Program. **75**, 153–177 (1995)
19. Goldberg, A., Plotkin, S., Shmoys, D., Tardos, E.: Using interior point methods for fast parallel algorithms for bipartite matching and related problems. SIAM J. Comput. **21**(1), 140–150 (1992)
20. Jonker, R., Volgenant, A.: A shortest augmenting path algorithm for dense and sparse linear assignment problems. Computing **38**, 325–340 (1987)
21. Karp, R.M., Steele, J.M.: Probabilistic analysis of heuristics. In: The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, Chichester (1985)
22. Korimont, T., Burkard, R., Çela, E.: An interior point approach for weighted bipartite matching. Technical Report FB196, Technische University Graz (2000)
23. Kuhn, H.W.: The hungarian method for the assignment and transpotation problems. Nav. Res. Logist. Q. **2**, 83–97 (1955)
24. Machol, R., Wien, M.: A 'hard' assignment problem. Oper. Res. **24**, 190–192 (1976)
25. Miller, D.L., Pekny, J.F.: Exact solution of large asymmetric traveling salesman problems. Science **251**, 754–761 (1991)
26. Ramakrishnan, K.G., Karmarkar, N.K., Kamath, A.P.: An approximate dual projective algorithm for solving assignment problems. In: Network Flows and Matching. DIMACS, pp. 431–451. American Mathematical Society, Providence (1993)
27. Schütt, C., Clausen, J.: Parallel algorithms for the assignment problem—an experimental evaluation of three distributed algorithms. In: Parallel Processing of Discrete Optimization Problems. DIMACS, pp. 337–351. American Mathematical Society, Providence (1995)
28. Schwartz, B.L.: A computational analysis of the auction algorithm. Eur. J. Oper. Res. **74**, 161–169 (1994)
29. Zhang, W.: Truncated branch-and-bound: A case study on the asymmetric tsp. In: Proceedings of AAAI 1993 Spring Symposium: AI and NP-Hard Problems, pp. 160–166. Stanford, CA (1993)