Parallel Auction Algorithm for Linear Assignment Problem

Xin Jin

1 Introduction

The (linear) assignment problem is one of classic combinatorial optimization problems, first appearing in the studies on matching problems in the 1920s. Since it closely relates to a wide range of important problems, such as min-cost network flow, weighted matching problem, and linear programming, it accumulates a large amount of literature in history.

Despite its difficulty, the problem can be described rather simply: given n persons and n items, and each person-item pair is associated with a payoff, then what is the optimal assignment with respect to the total payoff in which everyone gets exactly one item.[1] The payoffs can be summarized in a matrix, as illustrated below:

	item 1	item 2	item 3
person 1	7	9	8
person 2	8	5	7
person 3	1	6	6

In this particular example, the optimal assignment is apparent: person 1 gets item 2, person 2 gets item 1, and person 3 gets item 3. Each payoff can also be $-\infty$, indicating that some item cannot be assigned to some of the people.

The long history of improvements of the algorithms on the assignment problem originated from the 1940s with complexity $O(2^n n^2)$. The most famous one among these algorithms is the *Hungarian algorithm*, proposed in Kuhn [5]. It has time complexity $O(n^4)$, and is later improved to $O(n^3)$.

[1] Most of the literature uses the equivalent min-cost version of the problem. But since the "auction" setting only makes sense with the objective of maximizing payoffs, we instead use max-payoff version.

There have been over 10 algorithms that are similar in performances (and which one is the best depends on specific situations) these days, and active studies are still on-going.

In this paper, we analyze the auction algorithm, and discuss a couple ways to parallelize it. We then give empirical performance results on a C++ implementation. In addition, we also make some preliminary analysis on possible distributed implementations on a Spark-like configuration.

2 Mathematical Formulation

Suppose the payoffs are given by a matrix $A \in \bar{\mathbb{R}}^{n\times n}$, where $\bar{\mathbb{R}} = \mathbb{R} \cup \{-\infty\}$ and A_{ij} denote the payoff of item j to person i. The purpose of allowing A_{ij} to be $-\infty$ is to capture that some assignment is not feasible. The linear assignment problem can then be written as an integer programming problem

$$\max_{X\in\mathbb{R}^{n\times n}} \quad \sum_{i=1}^{n}\sum_{j=1} A_{ij}X_{ij} \tag{2.1}$$

$$\text{subject to} \quad \sum_{i=1}^{n} X_{ij} = 1, \quad i = 1, 2, \ldots, n \tag{2.2}$$

$$\sum_{j=1}^{n} X_{ij} = 1, \quad j = 1, 2, \ldots, n. \tag{2.3}$$

$$X_{ij} \in \{0, 1\}. \tag{2.4}$$

The constraints are simply saying that the argument matrix X must be a permutation matrix. Since the constraint matrices are totally unimodular, the integer programming problem (2.1) is equivalent to its linear programming relaxation:[2]

$$\max_{X\in\mathbb{R}^{n\times n}} \quad \sum_{i=1}^{n}\sum_{j=1} A_{ij}X_{ij} \tag{2.5}$$

$$\text{subject to} \quad \sum_{i=1}^{n} X_{ij} = 1, \quad i = 1, 2, \ldots, n \tag{2.6}$$

$$\sum_{j=1}^{n} X_{ij} = 1, \quad j = 1, 2, \ldots, n. \tag{2.7}$$

$$X_{ij} \geq 0. \tag{2.8}$$

[2] See, for instance, Papadimitriou and Steiglitz [6].

As this is a linear programming problem, standard LP algorithms can be used to solve it reasonably efficiently. However, the features of this problem enable us to find more efficient algorithms.

Let λ_i and p_i denote the multipliers (or dual variables) associated to the constraints, we have the dual problem:

$$\min_{\lambda_i, p_i} \quad \sum_{i=1}^{n} \lambda_i + \sum_{i=1}^{n} p_i \tag{2.9}$$

$$\text{subject to} \quad \lambda_i + p_i \le A_{ij}, \qquad i, j = 1, 2, \ldots, n. \tag{2.10}$$

The *complementary slackness* is

$$X_{ij}(A_{ij} - \lambda_i - p_j) = 0, \qquad i, j = 1, 2, \ldots, n. \tag{2.11}$$

If X^* is feasible for the primal and $\{\lambda_i, p_i\}$ are feasible for the dual and they together satisfy (2.11), then they must be optimal respectively for the primal and the dual.

There are various algorithms that exploit either the primal or the dual alone. Some algorithms also deal with both formulations together, e.g., the famous *Hungarian algorithm* proposed in Kuhn [5]. The *auction algorithm* studied in this paper was proposed by Bertsekas [1], and approaches the problem via a slightly different but equivalent dual formulation:

$$\min_{p_j} \left\{ \sum_{i=1}^{n} \max_j \left\{ A_{ij} - p_j \right\} + \sum_{j=1}^{n} p_j \right\} \tag{2.12}$$

As we will see, the variables p_j's have very intuitive explanations, and will be called the *prices* in the rest of the paper.

Proposition 2.1. *Let $\{p_j\}$ be a set of prices, and $\{(i, k_i)\}$ a feasible assignment for the dual problem* (2.9)*, i.e., item k_i is assigned to person i. If $\{p_j\}$ and $\{(i, k_i)\}$ satisfy the complementary slackness condition*

$$A_{ik_i} - p_{k_i} = \max_j \{A_{ij} - p_j\}, \tag{2.13}$$

then $\{p_j\}$ is optimal for (2.12) *and $\{(i, k_i)\}$ is optimal for* (2.9)*. Moreover, the optimal values of the two problems are the same.*

Proof. Let . And let $\{p_j\}$ be any specification of prices. We have

$$A_{ik_i} \le \left(A_{ik_i} - p_{k_i}\right) + p_{k_i} \le \max_j \left\{A_{ij} - p_j\right\} + p_{k_i}.$$

Hence the total payoff for the assignment $\{(i, k_i)\}$ satisfies

$$\sum_{i=1}^{n} A_{ik_i} \leq \sum_{i=1}^{n} \max_j \{A_{ij} - p_j\} + \sum_{j=1}^{n} p_j.$$

That is, any value of the objective function of (2.12) is at least as large as the optimal value of (2.9).

Now suppose $\{p_j\}$ and $\{(i, k_i)\}$ satisfy (2.13). Then

$$\sum_{i=1}^{n} A_{ik_i} = \sum_{i=1}^{n} \left(A_{ik_i} - p_{k_i}\right) + \sum_{j=1}^{n} p_j = \sum_{i=1}^{n} \max_j \{A_{ij} - p_j\} + \sum_{j=1}^{n} p_j.$$

It then follows that $\{p_j\}$ and $\{(i, k_i)\}$ are respectively optimal for (2.12) and (2.9), and the optimal values of the two problems are equal. □

In terms of asymptotic complexity, auction algorithm is pseudo-polynomial in that it also depends on the largest element of the payoff matrix, and thus seems inferior compared to the $O(n^3)$ implementation of Hungarian algorithm. However, auction algorithm has a very intuitive analogy (the real world auction bidding), is a lot easier to implement, and in random experiments studies have shown that it often outperforms the Hungarian algorithm; more importantly, in the world of parallel algorithms, auction algorithm is rather easy to parallelize. We will discuss the details in the next section.

3 Auction Algorithm

Due to the analogy to the auction setting, we shall call the persons bidders. The formulation (2.12) gives an easy heurstic (sequential) algorithm as follows:

1. Start with a set U of all bidders. U denotes the set of all *unassigned bidders*. We also maintain a set of prices which are initialized to all 0, and any structure that stores the current tentative (partial) assignment.

2. Pick any bidder i from U. Search for the item j that gives her the highest net payoff $A_{ij} - p_j$, and also an item k that gives her the second highest net payoff.

3. The price p_j of item j is updated to be $p_j \leftarrow p_j + \left(A_{ij} - p_j\right) - (A_{ik} - p_k)$. This update simply says that p_j is raised to the level at which bidder i is different (in terms of net payoff) bewteen item j and item k, i.e., the updated prices satisfy $A_{ij} - p_j = A_{ik} - p_k$.

4. Now assign item j to bidder i. If item j was previously assigned to another bidder s, then remove that assignment and add s to U.

5. If U becomes empty, the algorithm is over; otherwise, go back to Step (2).

As we can see, this algorithm resembles the procedure of English auction with no reserve prices (which is roughly equivalent to a second-price auction):

- The prices start from 0.
- In each round, one person makes a bid that is higher than the current highest bid by someone else, and tentatively gets that item. The updated price in Step (3) is exactly the new bid here.
- At the end of the auction, the person with the highest bid for some item gets that item.

We called the above algorithm only heuristic because it turns out that it is flawed: it may happen that several persons "kick" each other out (turned back to U) in a cycle, making the loop indefinite; the situation occurs when these persons' highest net payoffs are all equal to the second highest net payoffs, which then doesn't actually increase the prices. One resolution to this is to find a approximately optimal solution: instead of the original updating rule:

$$p_j \leftarrow p_j + \left(A_{ij} - p_j\right) - \left(A_{ik} - p_k\right),$$

we raise prices by

$$p_j \leftarrow p_j + \left(A_{ij} - p_j\right) - \left(A_{ik} - p_k\right) + \epsilon,$$

where ϵ is some positive real number that is pre-chosen. The idea is that, a cycle occurs when prices do not increase at all starting from some period, so by making updated prices always strictly larger, we guarantee that a stagnation never happens. The final assignment found by this modified algorithm can generate total payoff at most $n\epsilon$ apart from the optimal assignment. In most realistic applications, payoffs are integral, and thus if we take $\epsilon < 1/n$, the near-optimal assignment in fact must be optimal.

4 Parallel Version

In this section, we describe two main ways to parallelize the auction algorithm. In the literature, they are called Gauss-Seidel version and Jacobi version, in analogy to the iterative methods to solving linear system of equations. Finally, we compare the emprical results of these two algorithms as well as a hybrid version of the two. In what follows, we let p denote the number of CPU cores we use.

4.1 Gauss-Seidel Version

The most natural way to make the auction algorithm parallel is to parallelize Step (2). The search for the best item for some bidder on an iteration can be performed by multiple threads searching in

partitions of the items.

The biggest drawback is that, after the separate searches, the search results have to be merged in some way to give the overall best item and second-best item. This synchronization cost turns out to be quite significant in practice, bottlenecking the performance of the whole algorithm.

4.2 Jacobi Version

Instead of different threads searching over different partitions, we can also parallelize the algorithm by searching for the bids of multiple bidders at the same time. Clearly this reduces the number of iterations up to p times. It may happen that two or more bidders make bids for the same item on one iteration; in this case, we can only make one of them the tentative owner of the item. In practice, the reduction in iterations are rather unpredictable, but on average it's quite effective. Some literature calls this parallel algorithm the block Gauss-Seidel, and instead call an algorithm that updates all the unassigned bidders on one iteration the Jacobi version. But in this paper, we shall call it Jacobi, since it's more suitable for machines with limited number of CPU cores. Essentially, these two different Jacobi algorithsm are very similar, and if we use a queue to update unassigned bidders then they are exactly the same.

There is also one synchronization stage at the end of every iteration: we have to make sure several bidders bidding for the same item do not conflict, since the prices used to search for the best item may be outdated. Interestingly, this kind of synchronization can actually be avoided, resulting in a asynchronous version of the Jacobi algorithm. That is, one can prove that, even with outdated prices during the search, updating the price as long as the new price is higher than the orignal (but latest) price is still correct. The proof and implementation details are a bit tricky. Full details of the proof can be found in [3], and the URL of the code can be found in the next section.

4.3 Complexity

Let's first consider the sequential version. The time complexity turns out to be rather difficult to obtain. Bertsekas [1] gave a worst-case performance of $O(n^3) + O(Cn^2)$ (for fully dense payoff matrix, i.e., no $-\infty$), where C is the magnitude of the largest element in the payoff matrix. This looks pessimistic compared to the Hungarian algorithm, but in practice it's often more efficient, suggesting that worst-case bound in this case isn't a very good indicator of performance. As we will see below, this is more so in the parallel world. Schwartz [7] analyzed the average complexity by making a very strong assumption on the disitributional properties; he estimates the expected performance to be $O(n^2 \log n)$ (under the strong assumption).

For the parallel algorithms, the Gauss-Seidel version apparently reduces the search cost by a factor of p, with only an additional "minor" (unfortunately it's not minor in practice, as we shall

see) cost of $O(p)$ to merge the results. The number of iterations is exactly the same as the sequential version.

The search cost on one iteration in the Jacobi version is the same as the sequential version. Moreover, the synchronization cost is nearly 0, as we can directly write the new prices into memory (under protection of mutex) without a merge stage. So the only difference (advantage) of the Jacobi version is the reduced number of iterations. Hence theorectically the speedup is simply the ratio between numbers of iterations. As mentioned previously, depending the specific graph structure, the reduction in the number of iterations are rather unpredictable (though obviously the upper bound is p). So our analysis would mainly rely on the empirical results, as in most other literature analyzing the auction algorithm.

4.4 Implementation and Empirical Analysis

I implemented a multithreaded C++ program for a general hybrid algorithm, that is, one can search bids in partitions (Gauss-Seidel component) as well as update multiple bidders at the same time (Jacobi component). And the Jacobi part uses the asynchronous idea mentioned above. The program can deal with arbitrary integral payoff specifications (not necessarily dense) as long as there exists at least one feasible assignment. The source code can be found on https://github.com/xin-jin/pauc.

To compare performances, we generate fully dense graphs with different size n and magnitude C of largest element. Given a choice of C, each payoff is drawn uniformly from 1 to C. For the following results, all C are taken to be 1000. So I only list the values of n in the table. Time unit is second. s denotes the number of unassigned bidders to update each iteration, and b denotes the number of partitions to search for the best and the second best net payoffs.

s	b	$n = 1000$	$n = 2000$	$n = 3000$	$n = 4000$	$n = 5000$
1	1	0.64	3.90	28.63	170.63	205.53
4	1	0.55	2.79	16.92	96.20	111.63
2	2	1.24	6.10	38.97	224.25	233.76
1	4	2.09	10.69	74.59	386.93	424.84

As we see from the empirical performances, the Jacobi version does effectively reduces the computation time. However, the Gauss-Seidel part, no matter alone or combined with a Jacobi component, actually slows down the program. I performed profiling analysis with *perf*, and it

shows that (see (4.1)), a roughly 45.88% of time is spent in the merge and mutex/lock of the synchronization stage! This bad performance may also be due to implementation details, but at this point we list a number of the main factors for the huge effect of synchronization:

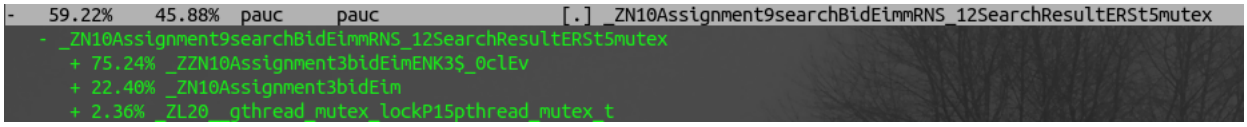

Figure 4.1: Perf

1. Even though the algorithm overall is not very fast (in the order of $O(n^3)$), the sequential search cost on each iteration is not large: $O(n)$. For today's computers, one iteration may only take a very short amount of time, so any slightly additional cost (those costs listed below) for every iteration may appear to be significant.

2. At the beginning of the search, the driver thread has to dispatch the tasks to the worker threads one-by-one.

3. The kernel time of dealing with multiple mutex locks and wake-ups (I use condition variables).

4. Similar to Reason (2), the synchronization stage has to wait for the worker threads to finish, and checking states happens sequentially.

The following table gives an empirical investigation of the reduction of iterations. We used the same matrices as above.

s	b	$n = 1000$	$n = 2000$	$n = 3000$	$n = 4000$	$n = 5000$
1	1	56797	308600	2245420	12347619	13478860
4	1	40481	147538	558954	3629568	3372102

Interestingly, for $s = 4$ and $b = 1$, the number of iterations for $n = 5000$ is in fact smaller than that for $n = 4000$, confirming that the number of iterations are highly unpredictable.

5 Distributed Version

The paper mainly focuses on the parallel algorithm, so here we only provide a very preliminary analysis of the potential distributed implementation. Before all iterations, we have to transfer the

payoff matrix to worker machines. For the Gauss-Seidel version, this requires p one-to-one communication of data of size $O(n^2/p)$ (again we assume a fully dense matrix). If the whole matrix is originally stored on a driver machine, then the communication cost of this is $O(n^2)$. For the Jacobi version, it requires a one-to-all broadcast of the whole $n \times n$ payoff matrix.

Synchronization cost is similar to that for the Gauss-Seidel parallel algorithm. However, it's different for the Jacobi version, because now we cannot directly write the updated price into memory. After each iteration, every machine needs to transfer the new bid to the driver, the driver use these bids to update the price vector, and one-to-all broadcast the updated price vector to all machines. As a result, the communication cost in the synchronization stage for each iteration is $O(n)$; depending on the particular distributed machanism, this may be reduced to $O(p)$, since we essentially only have to change at most p of the prices. The Gauss-Seidel version also needs to transfer the updated price, but only one price needs to be updated.

6 Conclusion

In this paper, we presented the mathematical formulation of the linear assignment problem. We analyzed the classical auction algorithm as well as several parallel versions of it, with comparisons from a multithreaded C++ implementation. In the end, we also briefly considered the distributed implementations.

References

[1] Bertsekas, Dimitri P. "A new algorithm for the assignment problem." Mathematical Programming 21.1 (1981): 152-171.

[2] Bertsekas, Dimitri P. Linear network optimization: algorithms and codes. MIT Press, 1991.

[3] Bertsekas, Dimitri P., and David A. Castanon. "Parallel synchronous and asynchronous implementations of the auction algorithm." Parallel Computing 17.6 (1991): 707-732.

[4] Burkard, Rainer E., Mauro Dell'Amico, and Silvano Martello. Assignment Problems, Revised Reprint. Siam, 2009.

[5] Kuhn, Harold W. "The Hungarian method for the assignment problem." Naval research logistics quarterly 2.1-2 (1955): 83-97.

[6] Papadimitriou, Christos H., and Kenneth Steiglitz. Combinatorial optimization: algorithms and complexity. Courier Corporation, 1982.

[7] Schwartz, B. L. "A computational analysis of the auction algorithm." European journal of operational research 74.1 (1994): 161-169.