Auctioning Algorithms

Subham Ghosh
sg46649

Manu Viswanadhan
mv26898

***Abstract*—In this paper, we look at Auctioning Algorithms to solve the classical assignment problem. A brief overview of the standard auction algorithm along with the mathematical reasoning is presented initially. We then build from this and present three stages of implementation. We start off with a centralized approach and then gradually progress towards a parallel asynchronous solution. Finally, we analyze the time taken by the system as we vary the problem size, i.e, the number of the persons and objects. We also vary certain other hyper parameters and see how it impacts the time taken by the system.**

I. Introduction

In the classical assignment problem, there are N persons and M objects that we have to match on a one-to-one basis. The objective is to maximize the total benefit a_{ij} for matching person i with object j. Mathematically, we want to find a one-to-one assignment [a set of person-object pairs $(1, j_1), ..., (n, j_n)$, such that the objects $j_1, ..., j_n$ are all distinct] that maximizes the total benefit $\sum_{i=1}^{n} a_{ij_i}$.

There are various applications of the general assignment problem. One such class is resource allocation problems, such as assigning jobs to machines. Another application is the stable marriage problem which we studied in class where men have to be matched with women given their preferences. There can also be instances where the assignment problem is a sub problem while solving more complex tasks.

II. Auction Process

Let us look into the economic equilibrium problem, which is equivalent to the assignment problem, to get a good understanding of the auction algorithm. We have to match n objects with n persons, each trying to maximize their profit.

Suppose that object j has a price p_j and that the person who receives the object must pay the price p_j. Then, the (net) value of object j for person i is $a_{ij} - p_j$ and each person i would logically want to be assigned to an object j_i with maximal value, that is, with

$$a_{ij_i} - p_{ji} = \max_{j=1,...,n} a_{ij} - p_j \tag{1}$$

Person i is judged to be happy if this condition holds for person i and consequently, when all persons are happy, the assignment and its corresponding set of prices are said to be at equilibrium.

Let us first look at one basic auction algorithm. Like any conventional auction, the process goes in rounds. At the beginning of each round, there is an assignment and set of prices for each object. If all persons are *happy* with the given assignment, the process terminates. Otherwise, we randomly choose some person who is not *happy*.

This person, call him i, finds an object j_i which offers maximal value, that is,

$$j_i \in \arg \max_{j=1,...,n} a_{ij} - p_j \tag{2}$$

and then,

- Exchanges objects with the person assigned to j_i at the beginning of the round,
- Sets the price of the best object j_i to the level at which he is indifferent between j_i and the second best object, that is, he sets

$$p_{ji} = p_{ji} + \gamma_i \tag{3}$$

where

$$\gamma_i = v_i - w_i \tag{4}$$

v_i is the best object value,

$$v_i = \max_j a_{ij} - p_j, \tag{5}$$

and w_i is the second best object value

$$w_i = \max_{j \neq j_i} a_{ij} - p_j \tag{6}$$

that is, the best value over objects other than j_i. (Note that γ_i is the largest increment by which the best object price p_{j_i} can be increased, with j_i still being the best object for person i).

This process is repeated in a sequence of rounds until all persons are happy. We can readily see that this process is equivalent to an auction where each bidder raises the price of his/her preferred object by a value of γ_i. One point to note here is that γ_i is always non negative (as $w_i \leq v_i$), making sure that the prices of the objects never come down. Let us now look into whether the above algorithm will ensure completion. In a real auction, the increment of price of an item by a bidder will make it less preferable to others, eventually leading to termination. But, a closer look at the algorithm will show us that if the two most preferred items have the same value for the bidder, γ_i would be 0 and the process will continue forever.

If we draw parallel from the real auction process, every bidding has to be an incremented amount to the present value, which is what we are missing in the proposed algorithm. So we introduce a positive scalar perturbation factor ϵ and say that a person i is almost happy with an assignment and a set of prices if the value of its assigned object j_i is within ϵ of being maximal, that is

$$a_{ij_i} - p_{ji} = \max_{j=1,\ldots,n} a_{ij} - p_j - \epsilon \tag{7}$$

When all persons are almost happy with the given assignment and its corresponding prices it is *almost at equilibrium*. The condition was first introduced in 1979, and is known as ϵ-*complementary slackness*. When ϵ is equal to 0, we get back the ordinary complementary slackness (Equation (1)). Now with this addition, we can reformulate the algorithm such that there is an increment of at least ϵ for each bid. The resulting method, called the *auction algorithm*, is almost identical to the basic algorithm mentioned above, except that the bidding increment γ_i is modified to

$$\gamma_i = v_i - w_i + \epsilon \tag{8}$$

rather than $\gamma_i = v_i - w_i + \epsilon$ as given in Equation (4).We can see that each bidder would be almost happy at end of each round with his choice (instead of happy). Also the γ_i defined above is the maximum value that can be incremented for an item at each round, and still maintain almost happy restriction. We can choose smaller increments for γ_i (making sure $\gamma_i \geq \epsilon$), although higher value will accelerate in the convergence process.

Now we will show that the algorithm terminates in finite number of steps. Once an item is bid by a person, he remains to be almost happy as long as he holds on to it (as the prices of other items cannot come down in the course of the algorithm). Also, once an item is bid by a person it will always be assigned to someone, as it will reassign itself only if another person has bid for it. This essentially means that once every item receives at least one bid the algorithm terminates. Another property to note is that after m bids, the item's price is increased by at least $m\epsilon$ from its initial price. As m gets higher the item will appear less attractive to the bidder compared to an unbid item, and finally will result in other items being bid. This combined with the previous property will ensure that given enough steps the algorithm will terminate.

A. Optimality and ϵ scaling

Once we have reached a final assignment, how do we check whether it is an optimal one. That is directly dictated by the value of ϵ fixed. In a real life auction the bidder will only increment the price of the object marginally so as to maximize his profit. Drawing from this example, we will now show that the assignment problem is almost optimal for small values of ϵ. In fact, *the total benefit of the assignment would be within $n\epsilon$ of the optimal value*. We can understand this by looking at the solution as at equilibrium for a slightly different problem where a_{ij} are the same as before, while the benefit for each of the n assigned pairs is shifted by no more than ϵ.
Now lets look at more relaxed condition where each of a_{ij} is an integer (we can apply it to non integer cases as well by scaling the weights to integer values). If we ensure that $n\epsilon$ is less than 1, then the almost optimal $n\epsilon$-solution converges to the optimal assignment. That is if

$$\epsilon < \frac{1}{n} \tag{9}$$

and the benefits a_{ij} are all integers, then the assignment obtained at termination is optimal.

The number of steps needed for convergence of the auction algorithm depends mainly on the value of ϵ and on the maximum absolute object value

$$C = \max_{j=1,\ldots,n} a_{ij} \tag{10}$$

Also note that there is a dependence on the initial prices of the objects. For the cases where the initial prices are low, the number of bidding cycles will be dependent on C/ϵ. On the other hand, when the prices are already near optimal the bidding converges much faster.
Taking into mind these properties, we can fasten up the process with a method called ϵ scaling. The idea is to start with higher values of ϵ and gradually decrease to it less than the critical value (usually $1/n$ when the benefits are integers). Every iteration of the algorithm would provide a good initial price for the next iteration.

Now that an overall idea of the auction algorithm and its computational aspects are covered, let us proceed with its implementation and analysis. We have designed two approaches for the auction process namely centralized and parallel-asynchronous algorithms. The algorithms will handle the extended cases where the number of persons and objects are not equal. We analyze the computational performance and also the impact of ϵ scaling.

III. Centralized Algorithm

We first propose a centralized version for the auction process, where a controller works like an *auctioneer* and proceeds to do the auction in *rounds*. At each round, the controller picks a person and asks that person to find the best object according to the criteria. This process goes on until all persons are assigned an object or it is known that the remaining persons cannot be assigned an object. We now go into further depths of the implementation process which we have followed.

A. Implementation Details

We have laid out the algorithm 1 for the *Controller* and the algorithm 2 for the *Person* instances. We spawn a separate thread for the Controller process. Each person instance is also spawned in a separate thread. We use RMI

calls to communicate between threads. RMI calls are used to communicate between the threads. The controller uses the variable $nSet$ to keep track of the number of persons who are still unassigned. It also uses two arrays, $price$ and $parent$ to track the price associated with the matching of each object and the index of the person assigned to each object respectively. The variable $lSet$ is used to simulate a queue of persons who will be addressed in subsequent rounds.

Algorithm 1 Centralized Controller

```
1:  N1 ← numberOfPersons
2:  N2 ← numberOfObjects
3:  ϵ ← 1/(2 * N1)
4:  i ← 0
5:  for i < N1 do
6:      lSet_i ← i
7:      i ← i + 1
8:  i ← 0
9:  for i < N2 do
10:     price_i ← 0.0
11:     parent_i ← null
12:     i ← i + 1
13: nSet ← N1
14: while nSet > 0 do
15:     currPerson ← lSet_{nSet−1}
16:     state ← PENDING
17:     while state = PENDING do
18:         WAIT
19:         state, obj, objPrice ← RUN(currPerson, price)
20:     if obj = null then
21:         nSet ← nSet − 1
22:     else
23:         if parent_obj = null then
24:             nSet ← nSet − 1
25:         else
26:             lSet_{nSet−1} ← parent_obj
27:         parent_obj = currPerson
28:         price_obj = objPrice
```

At each round, the Controller polls the person to be addressed from the $lSet$ data structure. It initializes the state of the person to be *PENDING* and launches the thread corresponding to the specific person. While launching the person instance, the current prices of the objects are also passed to the person as it will be using these current prices to decide the value of the objects. The Controller now periodically waits and checks if the Person instance has reached *DECIDED* state.

Algorithm 2 Person i

```
1:  function RUN(prices)
2:      state ← PENDING
3:      myObj ← null
4:      myObjPrice ← null
5:      max ← 0.0
6:      maxIndex ← −1
7:      j ← 0
8:      for j < numberOfObjects do
9:          if a_ij − prices_j > max then
10:             max ← a_ij − prices_j
11:             maxIndex ← j
12:         j ← j + 1
13:     if maxIndex = −1 then
14:         state ← TERMINATED
15:         myObj ← null
16:         myObjPrice ← null
17:         return state, myObj, myObjPrice
18:     else
19:         secondMax ← 0.0
20:         j ← 0
21:         for j < numberOfObjects do
22:             if a_ij − prices_j > max ∧ j ≠ maxIndex then
23:                 secondMax ← a_ij − prices_j
24:             j ← j + 1
25:         bidPrice ← prices_maxIndex + max − secondMax + ϵ
26:         state ← DECIDED
27:         myObj ← maxIndex
28:         myObjPrice ← bidPrice
29:         return state, myObj, myObjPrice
```

The person instance now iterates over all objects and tries to find the object which maximizes the value according to the given criteria in Equation 2. If it finds that not even one object generates a valid assignment, then it marks itself as *TERMINATED* as object prices can only increase in subsequent rounds. Therefore, there is no chance that it will be able to find a valid assignment in a round after this if it is not able to find a valid assignment now. However, if a valid assignment exists, then it finds the object which ranks second in its preferences. It marks its best preference as its current assignment and increases the price of it by the difference in value of the best and second best as given by Equation 3. Finally it returns the object chosen by it and the updated price of the object to the controller instance by means of an RMI call.

Now, the Controller has three options after the person is no longer in *PENDING* state. If the state of the person is *TERMINATED*, it simply decrements $nSet$ and removes the person from the $lSet$ queue. In our algorithm, the removal from the queue is simulated by implementing the $lSet$ data structure as an array and polling the $nSet$ index of the array as top of the queue. If the person was able to find a valid object and the object was not previously assigned to someone

else, then the controller reduces $nSet$ by one. However, if the object was assigned to someone else before, $nSet$ remains the same as that person will not have a matching object now. That displaced person is now supposed to replace the current person at the top of the queue and the next round will start from this displaced person.

IV. Asynchronous-Parallel Algorithm

One disadvantage that one can notice with the algorithm is that it is not completely distributed in the sense of the word. The controller has to keep track of the persons who are not assigned any items and ensure that the bidding moves forward. To resolve this issue, we moved towards a totally asynchronous approach. The idea was that both the bidding and the selection processes are parallelizable and hence we can do away with a controller or even a token-based approach. Drawing parallels from the stable matching problem, we now consider each person and item as a separate thread, with persons bidding(/proposing) for their most preferred item. We also extend the algorithm to be applicable even if the number of items and persons do not match (which is normally the case).

A. Implementation Details

Threads are spawned for persons and objects at the start of the method and RMI calls are used communicate between them as before. Initially all the persons are set to *PENDING* state suggesting that they are yet to assigned to an object. Each person independently computes the bidding price for their most preferred object (with the same algorithm as discussed in Section II). The person waits after sending the bid for the response.

Algorithm 3 Controller

```
1: objectSet ← null
2: personSet ← null
3: function REPORT(personId, objId, price)
4:     if personId ∉ personSet then
5:         personSet ← personSet ∪ {personId}
6:     if objId ∉ objectSet then
7:         objectSet ← objectSet ∪ {objId}
8:     if |objectSet| = min(nPersons, nObjects) ∧
       |personSet| = nPersons then
9:         TERMINATEALL
```

The object can receive multiple bids simultaneously and hence is protected by mutex to prevent any critical section problem. The object checks the proposed price and sends a *REJECT* if it is lesser than its current price. If the bid is in fact higher, the object broadcasts the updated price with *UPDATE* message to all the persons except the bidder. Once that is done, it sends back *ACCEPT* to the bidder.

Algorithm 4 Person i

```
1: state ← PENDING
2: myObj ← null
3: myObjPrice ← null
4: function START
5:     while state ≠ TERMINATED do
6:         if state = PENDING then
7:             BID
8:         else
9:             SLEEP(30)
10: function BID
11:     max ← 0.0
12:     maxIndex ← −1
13:     j ← 0
14:     for j < numberOfObjects do
15:         if a_ij − p_j > max then
16:             max ← a_ij − p_j
17:             maxIndex ← j
18:         j ← j + 1
19:     if maxIndex = −1 then
20:         State ← Terminated
21:         REPORT(Unmatched, i)
22:     else
23:         secondMax ← 0.0
24:         j ← 0
25:         for j < numberOfObjects do
26:             if a_ij − p_j > max ∧ j ≠ maxIndex then
27:                 secondMax ← a_ij − p_j
28:             j ← j + 1
29:         bidPrice ← p_maxIndex + max − secondMax + ϵ
30:         response ← BIDOBJECT(maxIndex, i, bidPrice)
31:         if response = ACCEPT then
32:             state ← DECIDED
33:             myObj ← maxIndex
34:             myObjPrice ← bidPrice
35:             REPORT(i, myObj, myObjPrice)
36: function UPDATE(objectId, price)
37:     if objectId = assignedObject then
38:         state ← PENDING
39:         assignedObjectPrice ← null
40:         assignedObjectId ← null
```

If the person receives *REJECT* from the bid, it will iterate back to find the next object to bid to. On the other hand, if its bid is successful (i.e. it received *ACCEPT*), the person changes its state to *DECIDED* and reports to the controller (*REPORT*) that it is currently assigned, along with its object and the price.

When the person receives an *UPDATE* message from an object, it updates the price of the object in its local array. It also checks if it is currently assigned to that object, in which case the object has been reassigned to another bidder and the person changes his state back to *PENDING* and checks for the next object to bid.

As hinted at the start of the section, while handling the condition where the number of persons and item are not matching (specifically when *nPersons* $>$ *nObjects*) every person cannot be assigned to objects. To handle this condition another state *TERMINATED* is introduced and will be used if the person does not have any more objects to bid to. The person also updates the controller (*REPORT*) about the termination.

Algorithm 5 Object i

```
1:  assignedPerson ← null
2:  currentPrice ← null
3:  function BIDOBJECT(personId, price)
4:      if currentPrice = null ∨ price > currentPrice then
5:          currentPrice ← price
6:          assignedPerson ← personId
7:          j ← 0
8:          for j < numberOfPersons do
9:              if j ≠ personId then
10:                 UPDATE(j, i, currentPrice)
11:             j ← j + 1
12:         return ACCEPT
13:     else
14:         return REJECT
```

The controller handles the termination detection, by keeping track of the state of persons (*personSet*) and objects(*objectSet*). As we wanted to minimize the message complexity, there is no interaction between object and the controller. Controller, in turn uses the information sent by the persons (*UPDATE* message) to document which objects have been assigned. As an object once bid will remain bid (explained in Sec II), the *objectSet* will never reduce in size. If you notice the algorithm given, there is no update message sent by a person if its assigned object is successfully bid by another person and it went back to *PENDING* state. This is because the controller utilizes the *objectSet* to keep track of all the objects that are assigned and its size wouldn't increase when the report message comes from the new successful bidder.
The controller decides that the auction process is terminated when

- all the persons have reported to the controller (citing termination or assignment) - In auction all persons should get the chance to bid
- the number of assigned objects is the minimum of *nPersons* and *nObjects* - This is true because once an object is bid, it always stays assigned to one person or another. Hence, once the required amount of objects are bid we can be assured of complete assignment and hence termination

V. EXPERIMENTS

We perform various experiments by changing the number of persons and the number of objects in the problem. We compare the time taken by the system for various runs. We give a brief outline of the methodology used to carry out these experiments.

For each set of parameters, we randomly initialize the weights on the edges(a_{ij}). Then we run the system on this input and record the time taken. We repeat this experiment 10 times for different random initializations of the weights and average out the time taken to give a fair estimate.

A. Varying number of persons and objects

1) Centralized algorithm: For the first set of experiments, we vary the *problem size*, which is the number of persons (and objects). The results of this experiment are given in Figure 1.

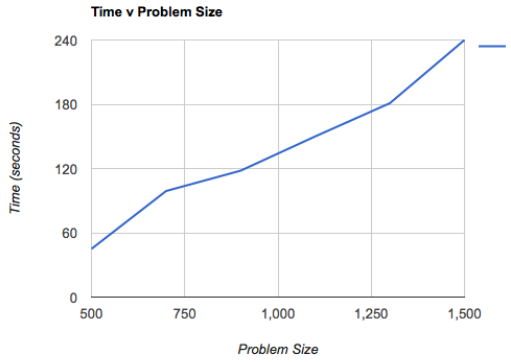

Fig. 1.

As expected the time taken for termination increases proportionally to the *problem size* due to increased amount of bidders and objects.

For our next experiment, we vary the number of objects, keeping the number of persons fixed to 500. The results are shown in Figure 2.

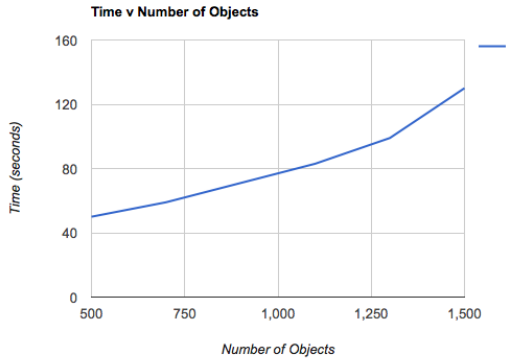

Fig. 2.

Now, we vary the number of persons, keeping the number of objects fixed. The results are shown in Figure 3.

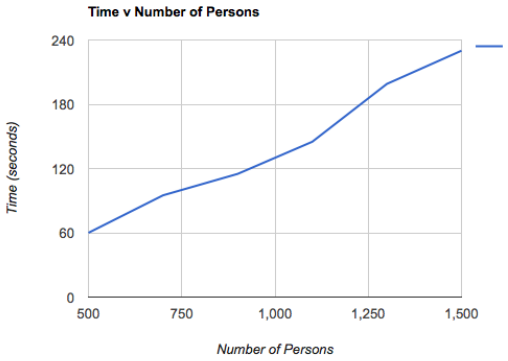

Fig. 3.

As expected, the time taken shows a steady increase as we increase the number of persons.

2) Asynchronous-Parallel algorithm: As the implementation involves spawning of different threads for each person and object, we have limited their number due to computational constraints. Below we have a graph comparing the performance of the two algorithms when the number of objects and persons (both equal) are ranging from 5 to 25.

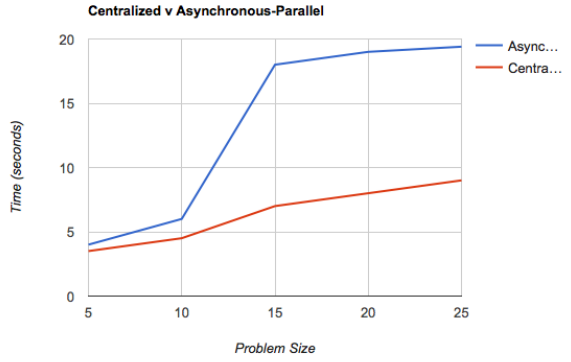

Fig. 4.

As expected the centralized algorithm is much faster due to the lack of conflicting bids coming from multiple persons (as it is synchronized). But, the asynchronous-parallel model makes up for its computational disadvantage by reduced message complexity.

B. ϵ scaling

Now we analyze the impact of ϵ scaling in the time taken for termination for both the algorithms. As discussed in the subsection II-A, we start off with a big value of epsilon and then gradually gradually decrease to it less than the critical value (equal to 1/n). In our implementation, we identify the maximum entry in the benefit matrix, a_{ij}, and used that to get an idea of what the initial value of ϵ should be. We then reduce ϵ value by a factor of $\frac{2}{3}$ after every bidding cycle until it goes below the critical value. From then on we keep it constant till termination.

Algorithm 6 ϵ scaling

$\epsilon \leftarrow \frac{\max_{ij} weight_{ij}}{4}$
while $\neg TERMINATED$ **do**
  BID
  if $\frac{2}{3}\epsilon > \frac{1}{N1}$ **then**
    $\epsilon \leftarrow \frac{2}{3}\epsilon$

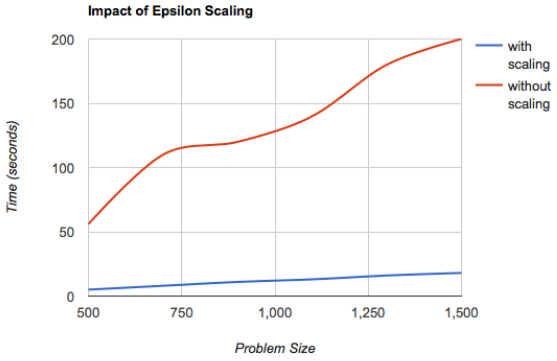

Fig. 5. Centralized algorithm

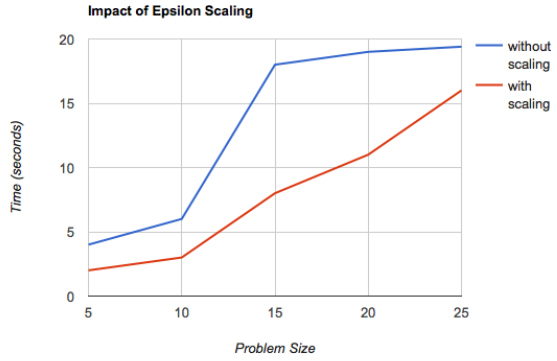

Fig. 6. Asynchronous-Parallel Algorithm

As expected, ϵ scaling helps in terminating the process quickly. As the number of people increases, the gap is starkly visible (as can be seen in Fig. 6).

VI. Conclusion

In this paper, we have introduced the auction algorithm for the assignment problem in a bipartite graph. We first provide a centralized algorithm where the controller organizes synchronous rounds and carries out the auction process. We then relax the centralized aspect of the solution where the controller, persons, and objects run on separate threads and decide on the final assignment process asynchronously

and in parallel. A termination detection algorithm is used to identify when the final assignment is made. We analyzed the computational performance of both the algorithms with and without ϵ scaling, and found the performance is vastly improved by its introduction.

The codes for project can be found at https://github.com/raltgz/BertsekasAuction

REFERENCES

[1] D.P. Bertsekas, *A distributed algorithm for the assignment problem*. Lab. for Information and Decision System Working Paper, M.I.T., March 1997.

[2] D.P. Bertsekas, *A new algorithm for the assignment problem*. Math. Program. 21 (1981) 152-171.

[3] D.P. Bertsekas, *A distributed asynchronous relaxation algorithm for the assignment problem*. Proc. 24th IEEE Conf. on Dec. and Control, 1985, 1703-1704.

[4] D.P. Bertsekas, *The auction algorithm: a distributed relaxation method for the assignment problem*. Ann. Oper. Res. 14 (1988) 105-123.

[5] D.P. Bertsekas, and D. Castanon, *Parallel Synchronous and Asynchronous Implementations of the Auction Algorithm*. Parallel Computing 17 (1991) 707-732, North Holland.