Parallel Auction Algorithm

Haolin Guo (hg2691), Yuanqing Lei (yl5457), Ava Penn (ap4315)

Dec. 16, 2024

1 Introduction

In this project, we implemented the sequential and parallel versions of an auction algorithm in Haskell. The auction algorithm is an optimization technique used for solving linear assignment problems, where the goal is to match agents to tasks in a way that minimizes or maximizes a total cost. The graph implementation of this problem is given a bipartite graph $G = (V, E)$ with bipartition (A, B) and weight function $w : E \to \mathbb{R}$, find a matching of maximum weight, where the weight of a matching M is given by $w(M) = \sum_{e \in M} w(e)$.

The sequential implementation of this algorithm is inspired by economic principles where agents bid for items (similar to a second-price auction), leading to iterative improvements in the set of prices until the total payoff is maximized. We chose to focus on this algorithm because it becomes computationally infeasible at a large number of bidders and items, and because steps 2 and 3 as indicated below are suitable for running in parallel–that is, they are mostly done independent of other tasks.

We focus on two approaches: the Jacobi implementation and the Guass-Seidel implementation, and compare their runtime efficiencies. These approaches are adapted from Jin [1], which implements a similar algorithm in C.

2 The Assignment Problem

Consider the following example. There are three bidders (B_1, B_2, B_3) and three items (I_1, I_2, I_3). The payoffs of assigning each item to each bidders are represented in the following payoff matrix:

$$\begin{bmatrix} 2 & 4 & 3 \\ 3 & 2 & 1 \\ 3 & 4 & 10 \end{bmatrix}$$

Here, the entry in row i and column j (e.g., 4 in the top left) represents the payoff of assigning item I_i to bidder J_j. In the context of auctions, payoff is similar to utility, or the value that the bidder assigns to a specific item (how much they are willing to pay).

The goal is to find an assignment where each item is assigned to exactly one unique bidder, such that the total payoff is maximized.

To maximize the total payoff in the example above:

- Assign I_1 to B_2 (payoff 4),
- Assign I_2 to B_1 (payoff 3),
- Assign I_3 to B_3 (payoff 10).

The total payoff of this assignment is:

$$4 + 3 + 10 = 17$$

This optimal solution can be obtained using algorithms like the auction algorithm. This algorithmic approach has applications in many types of allocation/linear assignment problems.

3 The Auction Algorithm

3.1 Brute-force Implementation

Using a brute-force sequential approach involves generating all possible permutations of assignments and calculating the total pay-off for each permutation to identify the maximum. This approach is exponential in the size of the input matrix, and is thus intractable. However, we used this brute-force approach to test the correctness of our sequential implementation on small matrices. What follows is our implementation in Haskell.

```haskell
optimalAssignment :: PayoffMatrix -> Assignment
optimalAssignment matrix = maximumBy (comparing totalPayoff) assignments
  where
    bidders = [0 .. length matrix - 1]
    items = bidders -- assume square matrix
    assignments = [Map.fromList (zip items perm) | perm <- permutations bidders]
    totalPayoff assignment = sum [matrix !! b !! i | (i,b) <- Map.toList assignment
       ]
```

3.2 Algorithm

The auction algorithm is taken from Jin [1]. It is pseudo-polynomial in that it also depends on the largest element of the payoff matrix. The worst-case performance is $O(n^3)$ or $O(C \cdot n^2)$, but on average, it is expected to perform in $O(n^2 \log n)$. The $O(n^3)$ Hungarian algorithm is more difficult to implement in parallel, however, and it has been found in practice that the auction algorithm often outperforms the Hungarian algorithm.

1. Start with a set U of all bidders. U denotes the set of all unassigned bidders. Initialize a set of prices to zero and any structure that stores the current tentative (partial) assignment.

```haskell
initialUnassigned = [0 .. numBidders - 1]
initialPrices = Map.fromList [(j, 0) | j <- [0 .. numItems - 1]]
```

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

```haskell
-- calculate net payoffs for all items
netPayoffs = [(j, netPayoff i j prices) | j <- [0 .. numItems - 1]]

-- find the best and second-best items
(bestItem, maxPayoff) = maximumBy (comparing snd) netPayoffs
secondMaxPayoff = if length netPayoffs > 1
                  then maximum [ p | (j,p) <- netPayoffs, j /= bestItem ]
                  else maxPayoff - epsilon
```

3. Update the price p_j of item j as:

$$p_j \leftarrow p_j + (A_{ij} - p_j - (A_{ik} - p_k)) \,. \tag{1}$$

This update ensures that the updated prices satisfy $A_{ij} - p_j = A_{ik} - p_k$ (it makes it so that the bidder is indifferent to buying the two items).

```haskell
-- update the price of the best item
newPrice = (prices Map.! bestItem) + (maxPayoff - secondMaxPayoff + epsilon)
updatedPrices = Map.insert bestItem newPrice prices
```

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

```
-- handle previous assignment of the item
(newAssignment, remainingUnassigned) =
  case Map.lookup bestItem assignment of
    Just prevBidder ->
      -- since bestItem was assigned to prevBidder, remove that assignment and
          add prevBidder back into U
      let updatedAssignment = Map.insert bestItem i assignment -- reassign item
            to current bidder i
          updatedUnassigned = prevBidder : unassignedBidders
      in (updatedAssignment, updatedUnassigned)
    Nothing ->
      (Map.insert bestItem i assignment, unassignedBidders)
in go remainingUnassigned updatedPrices newAssignment
```

5. If U becomes empty, the algorithm terminates; otherwise, return to Step 2.

4 Parallelization: Gauss-Seidel

The Gauss-Seidel version focuses on parallelizing step 2 of the auction algorithm, where each bidder searches for the best and second-best items to bid on. This parallelization divides the items among p threads, allowing each thread to search its partition independently.

This seems like the most obvious and intuitive way to implement parallelization, since the bid on an item i does not affect the bid on an item j. However, there is a loss of efficiency to overhead for the following reasons:

- **Synchronization Costs:** After the individual searches, the results must be merged to determine the overall best and second best items.
- **Load Imbalance:** If the item partitions are not evenly distributed, or if the complexity varies due to the variation in item values, some threads may finish earlier, leaving others idle.

4.1 Parallelization Choices

We implemented the algorithm in parallel using Haskell's `Control.Parallel.Strategies` library. Specifically, the `parMap` function is used to divide the workload across multiple threads, with each thread independently processing a partition of items.

- `rpar` sparks the evaluations in parallel. `parMap` starts the evaluation of each chunk in the list in parallel.
- `chunkItems` function: Items are split into p chunks, where each chunk is processed by a separate thread. This distributes independent computations of selecting the best bidder per item.

Here is a snippet of code that uses parallelization. Please see the appendix for the entirety of the code:

```
-- parallelize the search for best and second-best items
partitions = chunkItems 1600 netPayoffs -- change this number iteratively to find
    the best size chunk
-- for chunks: tested 2, 4, 8, 20, 100, 400, 1600, 6400, 10000, 20000
partialResults = parMap rpar findBestAndSecond partitions
(bestItem, maxPayoff, secondMaxPayoff) = mergeResults partialResults epsilon
```

We tested the code with varying numbers for p, seeing which number of chunks resulted in the best sparks output (i.e. wanting to keep garbage collected and fizzled sparks low). We ended up choosing 1600 chunks because we thought it offered a good balance between the number of threads and the number of work done on each thread, like the painters on the wall analogy. Also observe that the total number of sparks doesn't increase after 1600–because of the problem size. Since we tested this first with matrices of 1000x1000 and didn't observe a great speedup, we didn't try to optimize on larger problems and thus didn't end up needing to adjust this variable for larger problems. Here are the results of testing p chunks on four cores:

Number of Threads	Total Sparks	Converted	Overflowed	GC'd	dud	Fizzled
2	48305	5879	0	0	12468	29958
4	96611	15163	0	0	28712	52736
8	193203	42612	0	0	41620	108971
20	483003	100362	0	0	164934	217707
100	2415022	611727	0	0	1103680	699615
400	9660052	4538608	0	0	4081373	1040071
1600	24150254	24014601	0	0	2796	132857
6400	24150416	23863965	0	0	8463	277988
10000	24150184	24065539	0	0	3140	81505
20000	24150248	23986340	0	0	4805	159103

4.2 Gauss-Seidel Runtime/speedup analysis

We only tested this parallel approach on a 1000x1000 matrix (with randomly generated doubles between 0 and 100) due to realizing the Jacobi version offered more interesting results.

The following analysis shows that even though the GS version is simple, its merging overhead bottlenecks the speed.

Number of Cores	Runtime (s)	Speedup
1	176.316	1.00×
2	256.689	0.69×
3	301.140	0.59×
4	298.948	0.59×
5	267.741	0.66×
6	298.785	0.59×
7	298.913	0.59×
8	196.961	0.90×

Table 1: Runtime and speedup across different numbers of cores.

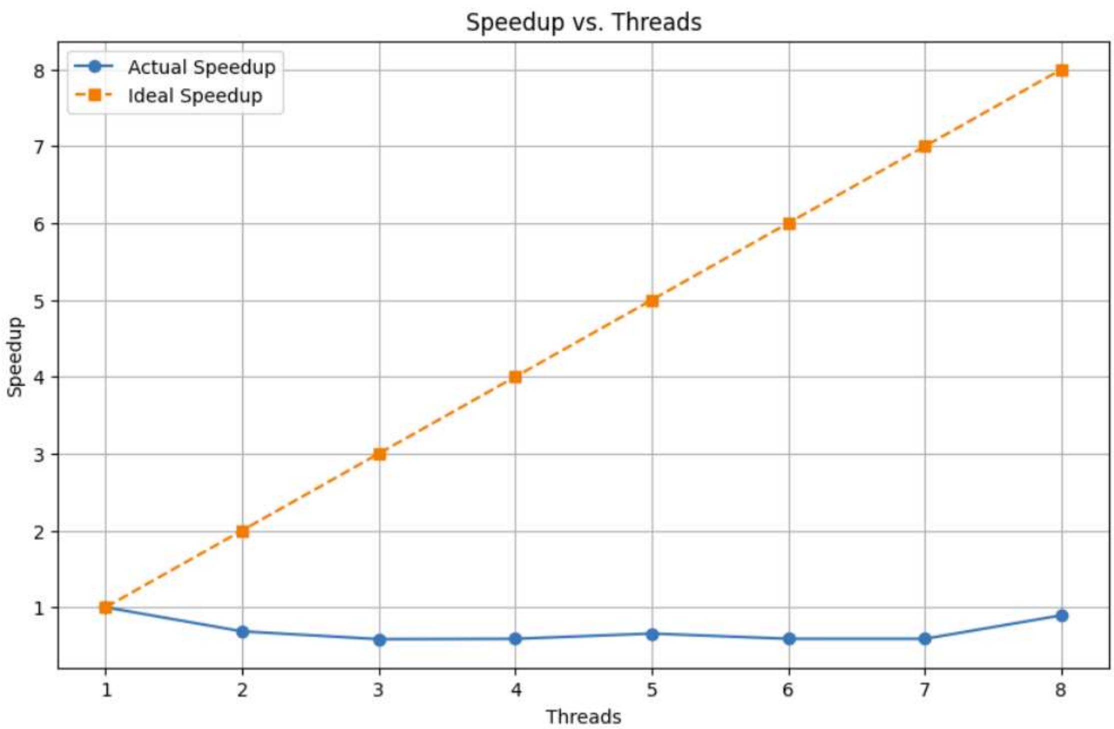
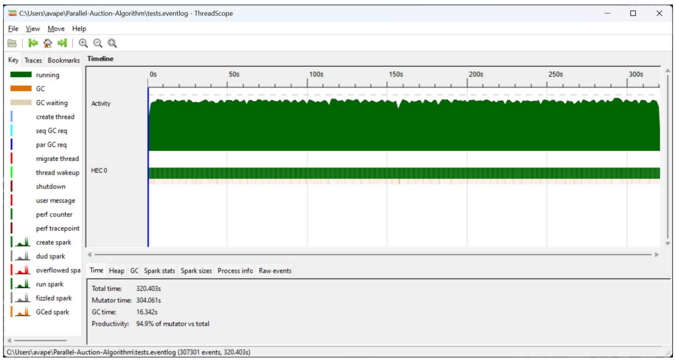
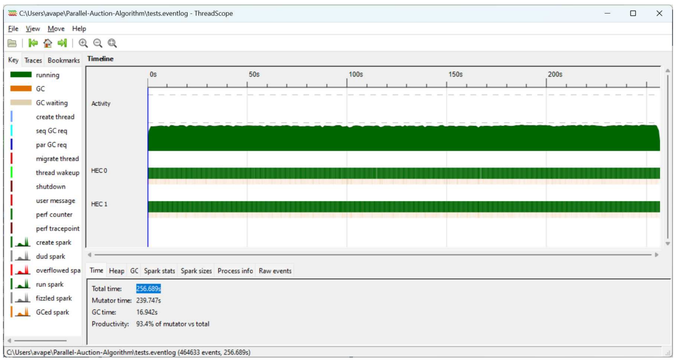


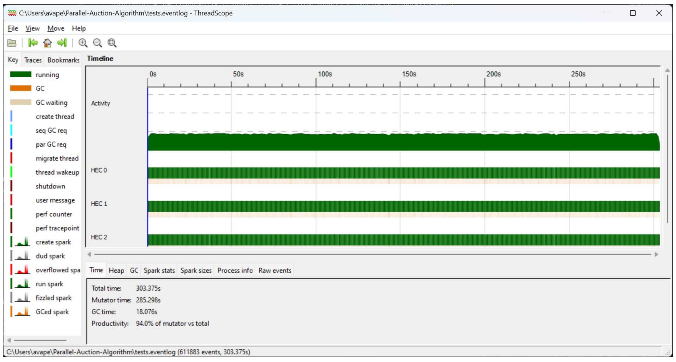
Figure 1: Guass-Seidel speedup for 1000x1000 matrix



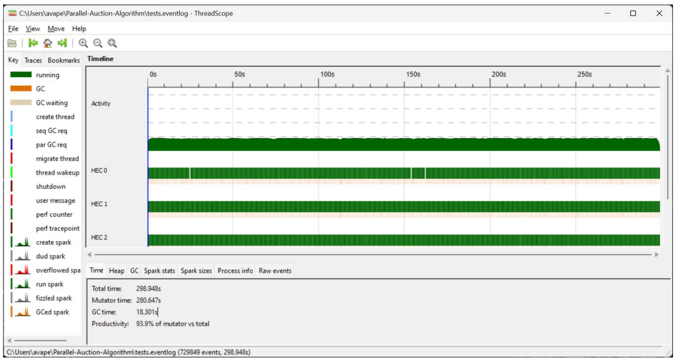
(a) 1 core: 176.316s



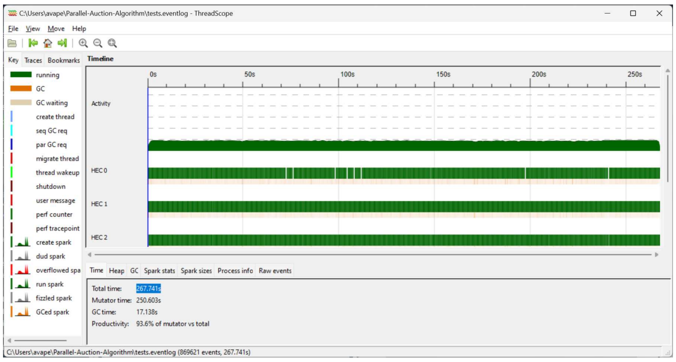
(b) 2 cores: 256.689s



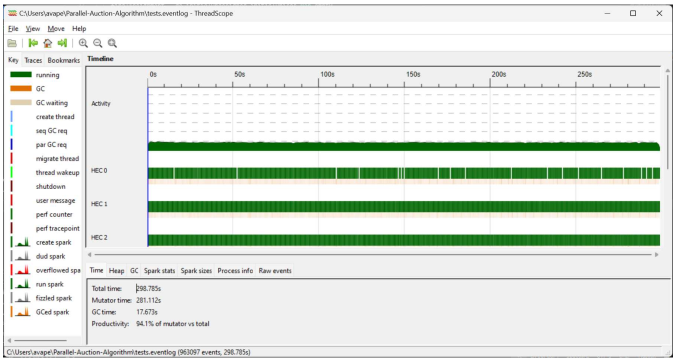
(c) 3 cores: 301.140s



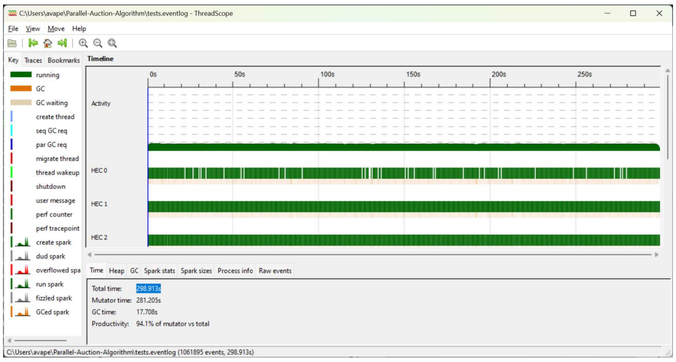
(d) 4 cores: 298.948s



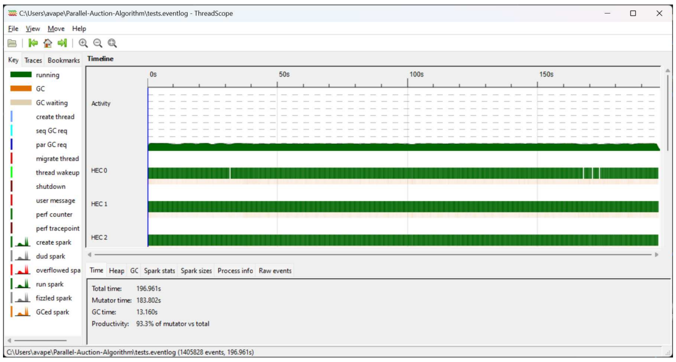
(e) 5 cores: 267.741s



(f) 6 cores: 298.785s



(g) 7 cores: 298.913s



(h) 8 cores: 196.961s

Figure 2: Execution times and event logs for different core counts

5 Parallelization: Jacobi

The Jacobi version parallelizes step 3 of the algorithm. It allows multiple bidders to search for their bids simultaneously. Through parallelization, each core handles a portion of the total bidders awaiting, reducing the runtime. Each thread handles one bidder. It may happen that two or more bidders make bids for the same item in parallel; in this case, we can only make one of them the tentative owner of the item. 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. It has been proven though that even with outdated prices during the search, updating the price as long as the new price is higher than the original (but latest) price is still correct.

By focusing on bidders rather than items, the Jacobi version avoids the merging overhead present in the Gauss-Seidel version, offering more interesting results.

5.1 Parallelization Choices

We implemented the algorithm in parallel using Haskell's `Control.Parallel.Strategies` library. Specifically, the `parMap` function is used to divide the workload across multiple threads, with each thread independently processing a partition of items.

Here is a snippet of code that uses parallelization. Please see the appendix for the entirety of the code:

```
synchronizedParallelBidding :: [Bidder] -> Prices -> [(Bidder, Item, Double)]
synchronizedParallelBidding bidders prices =
  map (bestBid prices) bidders `using` parList rdeepseq
```

- `using` applies the parallel evaluation strategy (`parList rdeepseq`) to a list of bidders.
- The `parList` strategy evaluates each element of a list in parallel.
- The `rdeepseq` strategy ensures that each element in the list is fully evaluated to normal form before being returned–it's used because the bid computation must be fully carried out before results can be merged.

Essentially, what this does is it creates a spark for each element in the list returned by map (bestBid prices) bidders. This is the same as each spark corresponding to finding the best item and bid price for a single bidder. This level of granularity was chosen because it was just the first implementation we tried and it happened to distribute the workload well. The total number of sparks however changes problem to problem, since the number of bidders in the subset U at any given iteration is variable depending on the payoff matrix. It changes even more drastically when the size of the matrix changes. For some measure of the problem size and how well it parallelizes we include the sparks information for a 1000x1000 matrix and a 3000x3000 matrix:

Size of matrix	Total sparks	Converted	Overflowed	GC'd	dud	Fizzled
1000x1000	1411	1407	0	0	0	4
3000x3000	4324	0	0	0	0	0

5.2 Jacobi Runtime/speedup analysis (1000x1000)

The table below summarizes the runtime of the auction algorithm executed on different numbers of cores, for a test case of 1000x1000.

Number of Cores	Runtime (s/ms)	Speedup
1	3.840 s	1.00×
2	3.709 s	1.04×
3	2.230 s	1.72×
4	2.351 s	1.63×
5	1.356 s	2.83×
6	1.292 s	2.97×
7	1.406 s	2.73×
8	969.87 ms	3.96×

Table 2: Runtime and speedup across different numbers of cores.

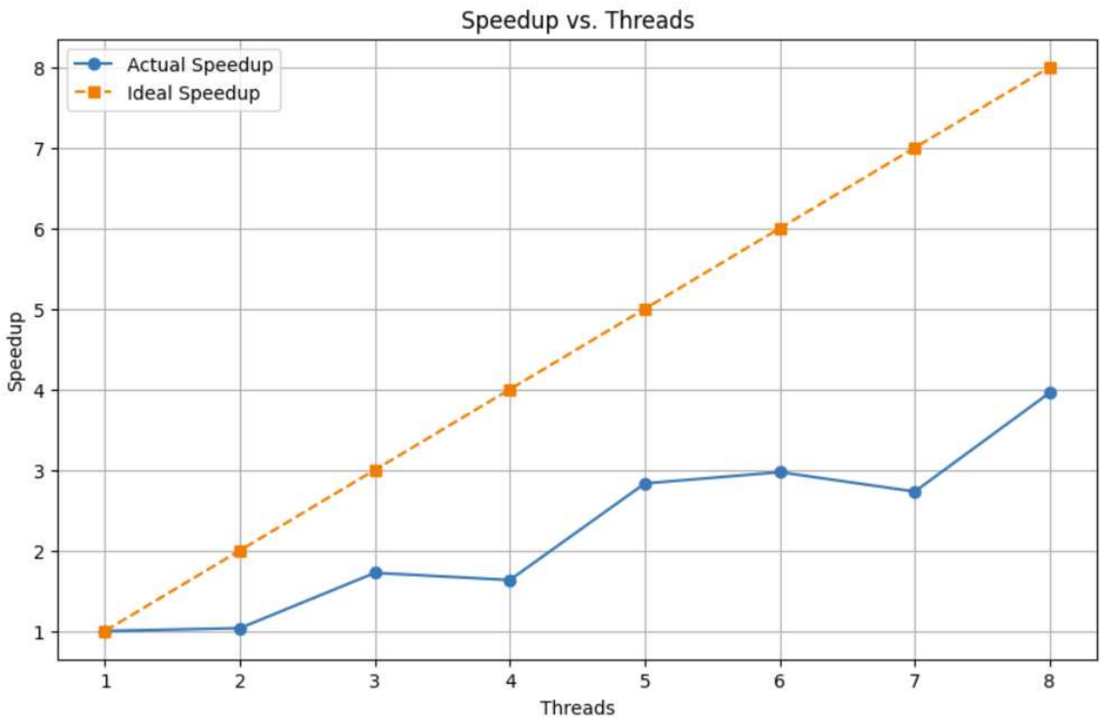
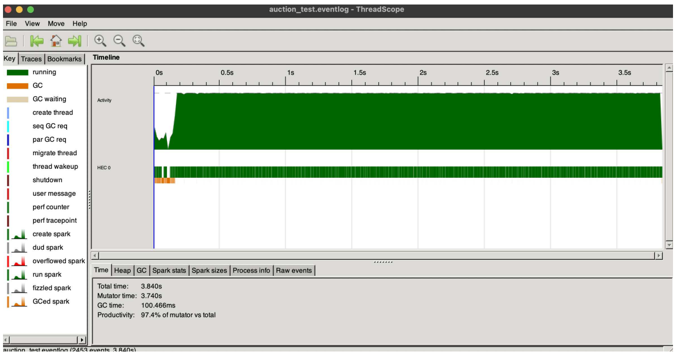
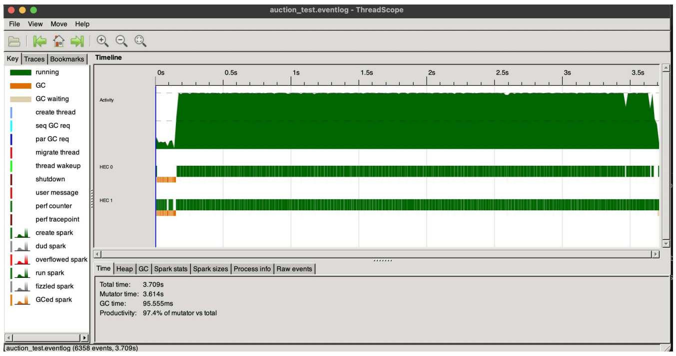


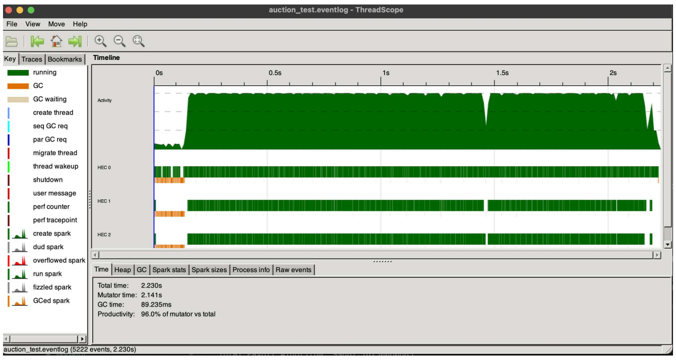
Figure 3: Actual speedup and ideal speedup



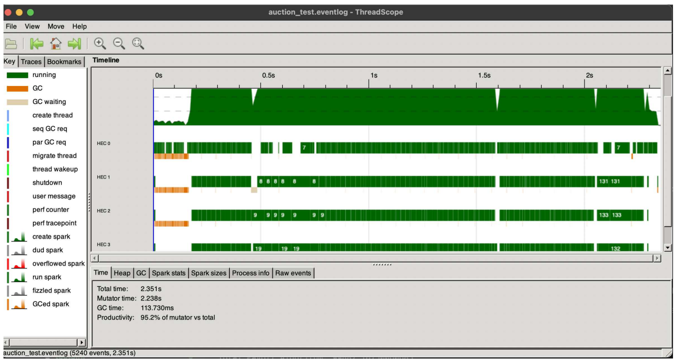
(a) -N1 eventlog



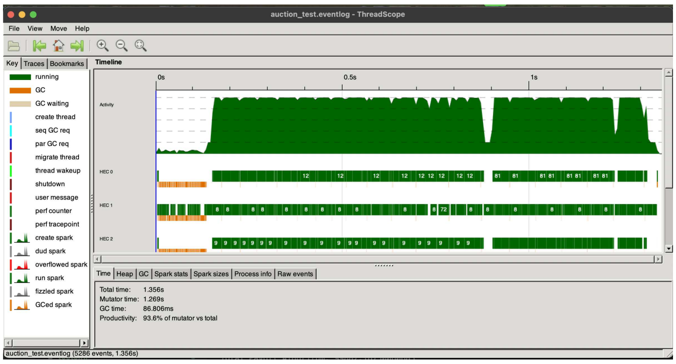
(b) -N2 eventlog



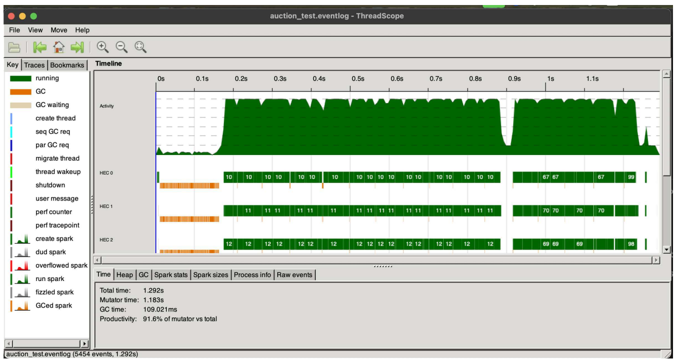
(c) -N3 eventlog



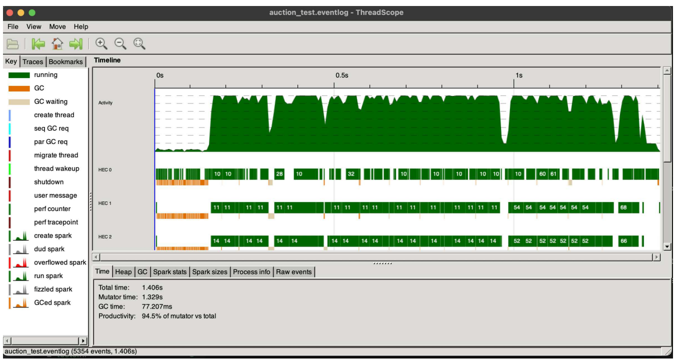
(d) -N4 eventlog



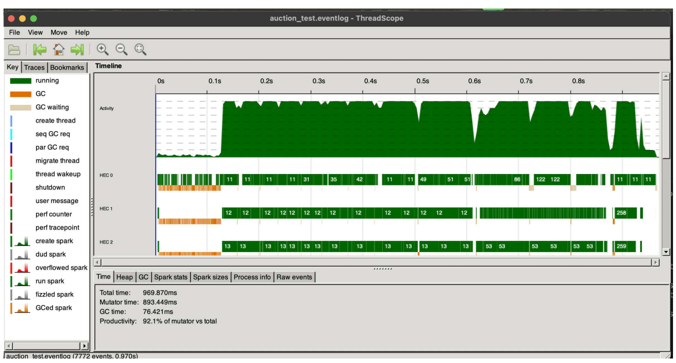
(e) -N5 eventlog



(f) -N6 eventlog



(g) -N7 eventlog



(h) -N8 eventlog

Figure 4: Jacobi Algorithm Eventlog for 1000 x 1000 matrix

As one can observe, the productivity measures for all numbers of cores are above 90%, signaling efficient core usage. However, the speedup is not ideal, since the test dataset is not large enough. We will test a larger use case of 3000 x 3000 to demonstrate the parallel algorithm's speedup ability.

5.3 Jacobi Runtime/speedup analysis (3000x3000)

The table below shows the runtime and speedup of the auction algorithm for a larger test case with a 3000x3000 matrix.

As the matrix size becomes larger, the speedup is more apparent. In this test case of 3000x3000, where runtime takes a measure of minutes, the speedup is closer to perfect.

Number of Cores	Runtime (s/ms)	Speedup
1	151.657 s	1.00×
2	82.560 s	1.83×
3	53.064 s	2.85×
4	40.421 s	3.75×
5	30.587 s	4.95×
6	27.942 s	5.42×
7	24.039 s	6.31×
8	21.226 s	7.14×

Table 3: Runtime and speedup across different numbers of cores for a 3000 x 3000 matrix.

As one observes, the speedup is more ideal as the matrix size gets larger. This is because as the matrix grows larger, the computations become more significant than the spark overhead. The speedup diagram demonstrates that we can achieve near-ideal speedup using the Jacobi algorithm.

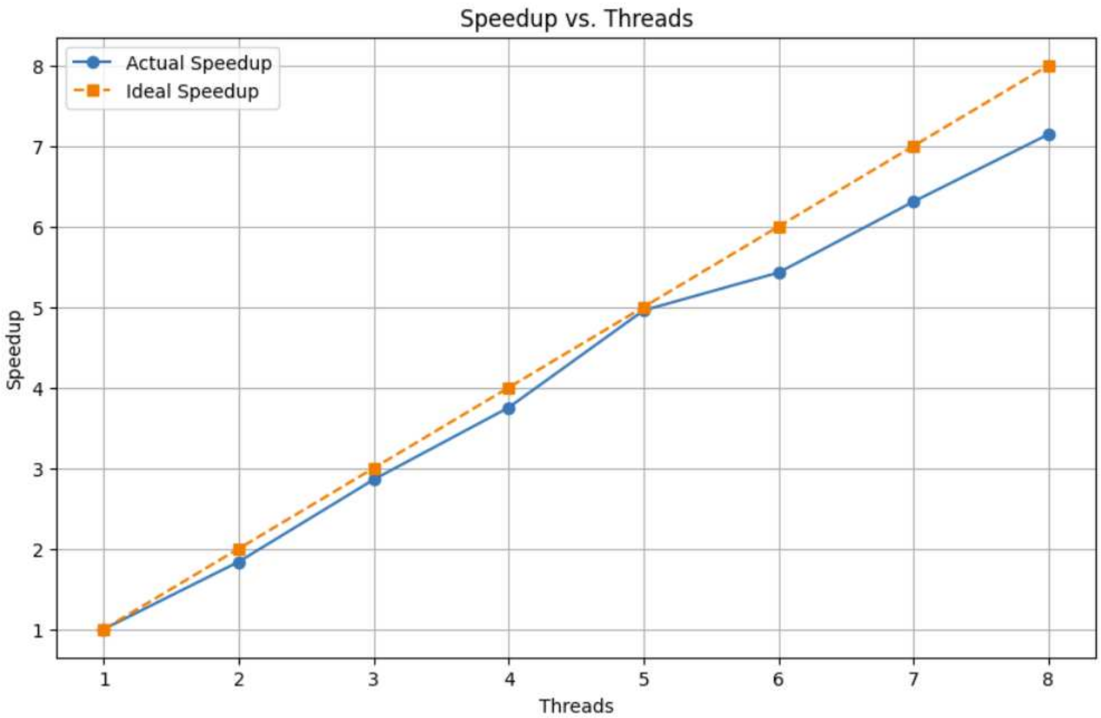
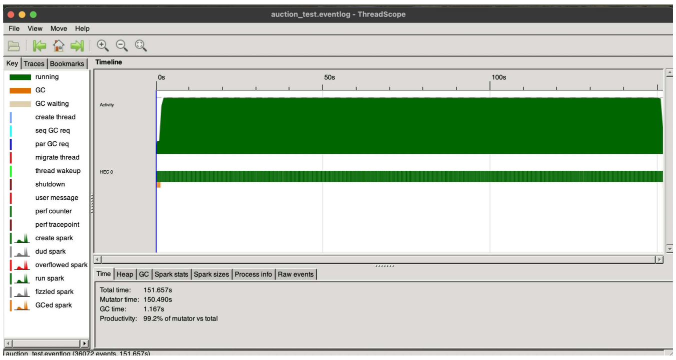
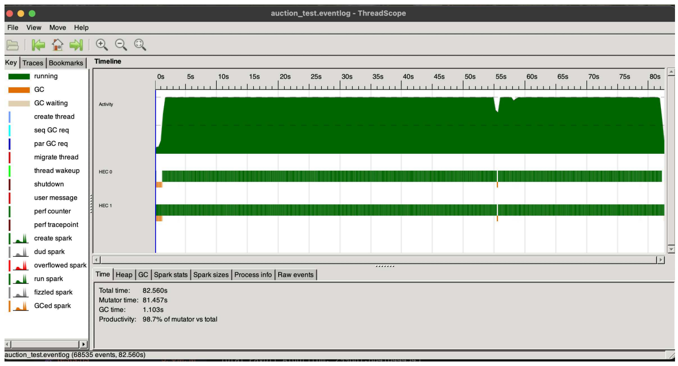


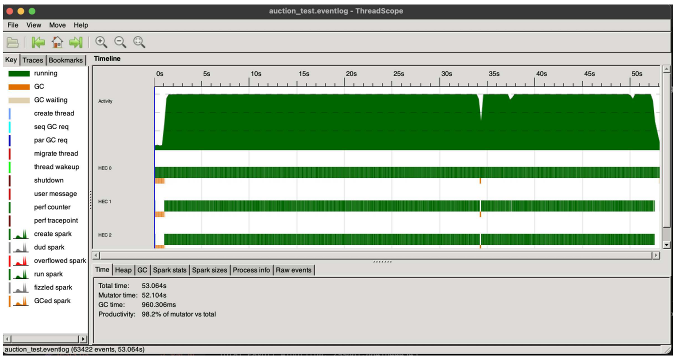
Figure 5: Actual speedup and ideal speedup



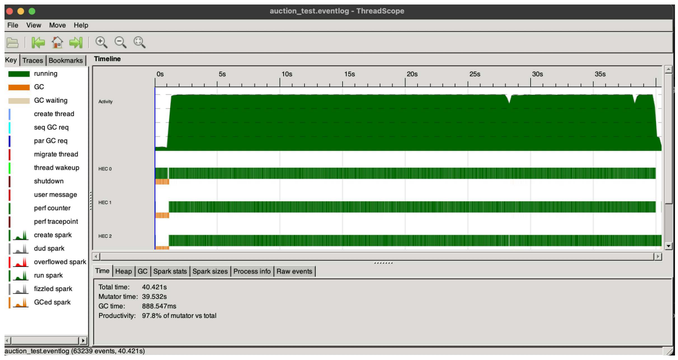
(a) -N1 eventlog



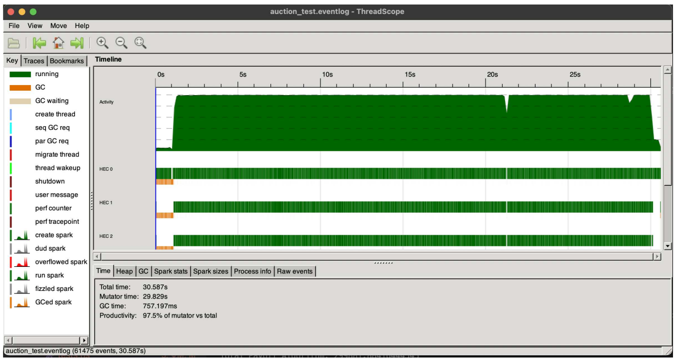
(b) -N2 eventlog



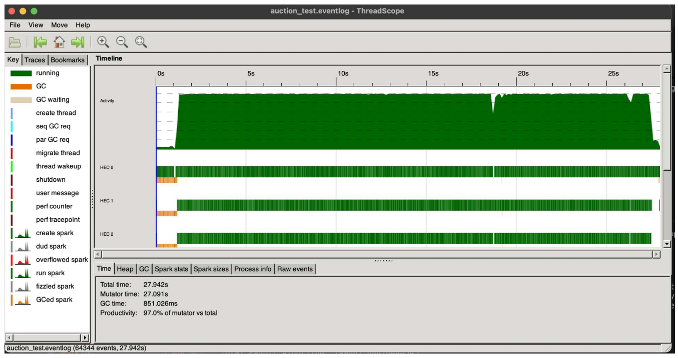
(c) -N3 eventlog



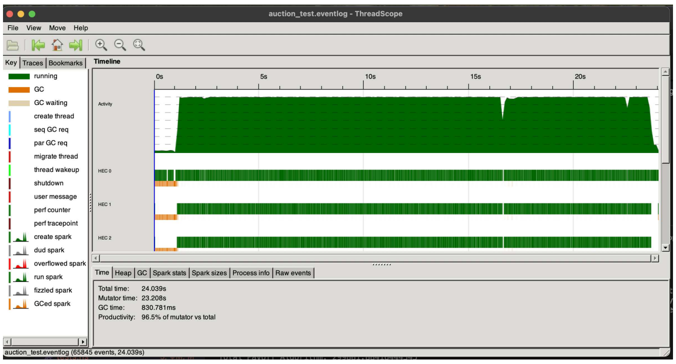
(d) -N4 eventlog



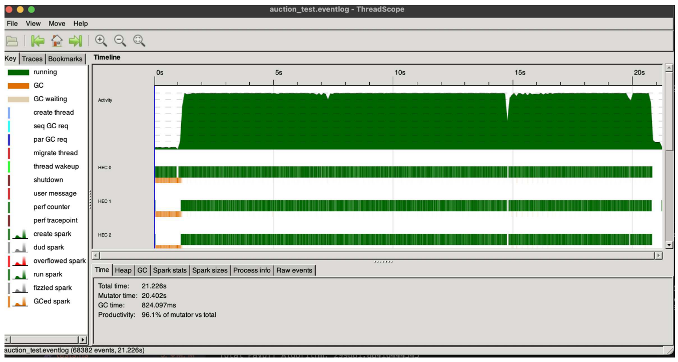
(e) -N5 eventlog



(f) -N6 eventlog



(g) -N7 eventlog



(h) -N8 eventlog

Figure 6: Jacobi Algorithm Eventlog for 3000 x 3000 matrix

6 Conclusion

- The assignment problem becomes more computationally practical in parallel!
- The Gauss-Seidel implementation faces significant synchronization overhead and load imbalance issues, resulting in substantially slower performance compared to the Jacobi version.

- As the data size increases, the Jacobi algorithm demonstrates near-ideal scalability, making it a highly effective approach for parallelization.

Note about testing: Testing was initially done with seven small matrices (with dimensions less than 6x6) to verify correctness. Once the algorithm was verified, random generation was introduced with the ability to adjust the size of the matrix through a command-line argument. Please see the test file and README.md for usage.

7 References

[1] Jin, J. (2016). Parallel Auction Algorithm for Linear Assignment Problem.

8 Appendix

gs_auction.hs

```haskell
module GSAuction (gsAuctionAlgorithm) where

import Control.Parallel.Strategies
import qualified Data.Map as Map
import Data.List
import Data.Maybe
import Data.Ord (comparing, Down(..))

type PayoffMatrix = [[Double]]
type Bidder = Int
type Item = Int
type Prices = Map.Map Item Double
type Assignment = Map.Map Bidder Item

gsAuctionAlgorithm :: Double -> PayoffMatrix -> (Assignment, Double)
gsAuctionAlgorithm epsilon inputMatrix = (finalAssignment, totalPayoff)
  where
    numItems = length (head inputMatrix)
    initialUnassigned = [0 .. length inputMatrix - 1]
    initialPrices = Map.fromList [(j, 0) | j <- [0 .. numItems - 1]]

    -- get the resulting assignment and also the total payoff, to return
    finalAssignment = go initialUnassigned initialPrices Map.empty
    totalPayoff = sum [inputMatrix !! bidder !! item | (item, bidder) <- Map.toList finalAssignment]

    go :: [Bidder] -> Prices -> Assignment -> Assignment
    go [] _ assignment = assignment
    go (i : unassignedBidders) prices assignment =
      let
        -- calculate net payoffs for all items
        netPayoffs = [(j, netPayoff i j prices) | j <- [0 .. numItems - 1]]

        -- parallelize the search for best and second-best items
        partitions = chunkItems 1600 netPayoffs -- change this number iteratively to find the best size chunk
        -- for chunks: tested 2, 4, 8, 20, 100, 400, 1600, 6400, 10000, 20000
        partialResults = parMap rpar findBestAndSecond partitions
        (bestItem, maxPayoff, secondMaxPayoff) = mergeResults partialResults epsilon

        -- update price according to the auction algorithm description
        newPrice = (prices Map.! bestItem) + (maxPayoff - secondMaxPayoff + epsilon)
        updatedPrices = Map.insert bestItem newPrice prices

```

```haskell
        -- handle previous assignment of the item
        (newAssignment, remainingUnassigned) =
          case Map.lookup bestItem assignment of
            Just prevBidder ->
              let updatedAssignment = Map.insert bestItem i assignment
                  updatedUnassigned = prevBidder : unassignedBidders
              in (updatedAssignment, updatedUnassigned)
            Nothing ->
              (Map.insert bestItem i assignment, unassignedBidders)
      in go remainingUnassigned updatedPrices newAssignment

    -- calculate net payoff for a bidder for a specific item
    netPayoff :: Bidder -> Item -> Prices -> Double
    netPayoff i j prices = inputMatrix !! i !! j - (prices Map.! j)

    -- find the best and second-best items in a partition
    findBestAndSecond :: [(Item, Double)] -> (Item, Double, Maybe Double)
    findBestAndSecond payoffs =
      let (bestItem, maxPayoff) = maximumBy (comparing snd) payoffs
          secondMaxPayoff = if length payoffs > 1
                            then Just $ maximum $ map snd (filter ((/= bestItem) . fst) payoffs)
                            else Nothing
      in (bestItem, maxPayoff, secondMaxPayoff)

    -- merge results from all partitions
    mergeResults :: [(Item, Double, Maybe Double)] -> Double -> (Item, Double, Double)
    mergeResults results epsilon =
      let
        allPayoffsWithItems = concatMap (\(item, p, ms) -> [(item, p), (item, fromMaybe (-1 / 0) ms)]) results
        sortedPayoffsWithItems = sortBy (comparing (Down . snd)) allPayoffsWithItems
        (bestItem, maxPayoff) = head sortedPayoffsWithItems
        secondMaxPayoff = if length sortedPayoffsWithItems > 1
                          then snd (sortedPayoffsWithItems !! 1)
                          else maxPayoff - epsilon
      in (bestItem, maxPayoff, secondMaxPayoff)

    -- split items into equal-sized chunks for parallel processing
    chunkItems :: Int -> [a] -> [[a]]
    chunkItems n items = let (q, r) = length items `quotRem` n
                         in goChunks q r items
      where
        goChunks _ 0 [] = []
        goChunks q r xs = let (chunk, rest) = splitAt (q + if r > 0 then 1 else 0) xs
                          in chunk : goChunks q (max 0 (r - 1)) rest
```

jacobi_auction.hs

```haskell
module JacobiAuction (jacobiAuctionAlgorithm) where

import Control.Parallel.Strategies (parList, rdeepseq, using)
import Data.List (maximumBy, foldl')
import Data.Ord (comparing)
import qualified Data.Map as Map

type Bidder = Int
type Item = Int
type Prices = Map.Map Item Double
type Assignment = Map.Map Item Bidder -- mapping from item to bidder (to correspond to implementation in paper)
type PayoffMatrix = [[Double]]
```

```haskell

jacobiAuctionAlgorithm :: Double -> PayoffMatrix -> (Assignment, Double)
jacobiAuctionAlgorithm epsilon inputMatrix = (finalAssignment, totalPayoff)
  where
    numItems = length (head inputMatrix)
    initialUnassigned = [0 .. length inputMatrix - 1]
    initialPrices = Map.fromList [(j, 0) | j <- [0 .. numItems - 1]]

    -- get the resulting assignment and also the total payoff, to return
    (finalAssignment, _) = runSynchronizedAuction initialUnassigned initialPrices Map.empty
    totalPayoff = sum [inputMatrix !! bidder !! item | (item, bidder) <- Map.toList finalAssignment]

    runSynchronizedAuction :: [Bidder] -> Prices -> Assignment -> (Assignment, [Bidder])
    runSynchronizedAuction [] _ assignment = (assignment, [])
    runSynchronizedAuction unassignedBidders prices assignment =
      let
        bidResults = synchronizedParallelBidding unassignedBidders prices
        updatedPrices = foldl' updatePrices prices bidResults
        (newAssignment, newUnassigned) = resolveConflicts bidResults assignment
      in
        if null newUnassigned
        then (newAssignment, newUnassigned)
        else runSynchronizedAuction newUnassigned updatedPrices newAssignment

    synchronizedParallelBidding :: [Bidder] -> Prices -> [(Bidder, Item, Double)]
    synchronizedParallelBidding bidders prices =
      map (bestBid prices) bidders `using` parList rdeepseq

    -- find the best item and second-best payoff for a bidder
    bestBid :: Prices -> Bidder -> (Bidder, Item, Double)
    bestBid prices i =
      let
        netPayoffs = [(j, netPayoff i j prices) | j <- [0 .. numItems - 1]]
        (bestItem, maxPayoff) = maximumBy (comparing snd) netPayoffs
        secondMaxPayoff = if length netPayoffs > 1
                          then maximum $ map snd (filter ((/= bestItem) . fst) netPayoffs)
                          else maxPayoff - epsilon
        bidPrice = (prices Map.! bestItem) + (maxPayoff - secondMaxPayoff + epsilon)
      in (i, bestItem, bidPrice)

    -- resolve conflicts: only one bidder can win an item
    -- paper states that this will still result in the optimal assignment, even if prices are outdated
    resolveConflicts :: [(Bidder, Item, Double)] -> Assignment -> (Assignment, [Bidder])
    resolveConflicts bids assignment =
      let
        groupedBids = Map.fromListWith (++) [(item, [(bidder, bidPrice)]) | (bidder, item, bidPrice) <- bids]
        resolvedAssignments =
          Map.mapWithKey (\_ bidders -> fst $ maximumBy (comparing snd) bidders) groupedBids
        newAssignment =
          foldl' (\acc (item, bidder) -> Map.insert item bidder acc) assignment (Map.toList resolvedAssignments)
        unassignedBidders =
          [bidder | (_, bidders) <- Map.toList groupedBids, (bidder, _) <- bidders, bidder `notElem` Map.elems newAssignment]
      in (newAssignment, unassignedBidders)
```

```

    -- update prices for items based on the winning bids
    updatePrices :: Prices -> (Bidder, Item, Double) -> Prices
    updatePrices prices (_, item, bidPrice) =
      let currentPrice = Map.findWithDefault 0 item prices
      in Map.insert item (max currentPrice bidPrice) prices

    -- calculate net payoff for a bidder for a specific item
    netPayoff :: Bidder -> Item -> Prices -> Double
    netPayoff i j prices = inputMatrix !! i !! j - (prices Map.! j)
```

sequential_auction.hs

```
module SequentialAuction (auctionAlgorithm, optimalAssignment) where

import Data.List (maximumBy, permutations)
import Data.Ord (comparing)
import qualified Data.Map as Map

type Bidder = Int
type Item = Int
type Prices = Map.Map Item Double

-- item is the key, bidder is the value, for consistency with the algorithm from
    the paper
type Assignment = Map.Map Item Bidder
type PayoffMatrix = [[Double]]


auctionAlgorithm :: Double -> PayoffMatrix -> (Assignment, Double)
auctionAlgorithm epsilon inputMatrix = (finalAssignment, totalPayoff)
  where
    numItems = length (head inputMatrix)
    numBidders = length inputMatrix

    initialUnassigned = [0 .. numBidders - 1]
    initialPrices = Map.fromList [(j, 0) | j <- [0 .. numItems - 1]]

    finalAssignment = go initialUnassigned initialPrices Map.empty
    totalPayoff = sum [inputMatrix !! bidder !! item | (item, bidder) <- Map.toList
        finalAssignment]

    go :: [Bidder] -> Prices -> Assignment -> Assignment
    go [] _ assignment = assignment
    go (i : unassignedBidders) prices assignment =
      let
        -- calculate net payoffs for all items
        netPayoffs = [(j, netPayoff i j prices) | j <- [0 .. numItems - 1]]

        -- find the best and second-best items
        (bestItem, maxPayoff) = maximumBy (comparing snd) netPayoffs
        secondMaxPayoff = if length netPayoffs > 1
                          then maximum [ p | (j,p) <- netPayoffs, j /= bestItem ]
                          else maxPayoff - epsilon

        -- update the price of the best item
        newPrice = (prices Map.! bestItem) + (maxPayoff - secondMaxPayoff + epsilon
            )
        updatedPrices = Map.insert bestItem newPrice prices

        -- handle previous assignment of the item
        (newAssignment, remainingUnassigned) =
          case Map.lookup bestItem assignment of
            Just prevBidder ->
              -- since bestItem was assigned to prevBidder, remove that assignment
```

```haskell
                  and add prevBidder back into U
              let updatedAssignment = Map.insert bestItem i assignment -- reassign item to current bidder i
                  updatedUnassigned = prevBidder : unassignedBidders
              in (updatedAssignment, updatedUnassigned)
            Nothing ->
              (Map.insert bestItem i assignment, unassignedBidders)
      in go remainingUnassigned updatedPrices newAssignment

    netPayoff :: Bidder -> Item -> Prices -> Double
    netPayoff i j prices = inputMatrix !! i !! j - (prices Map.! j)

-- find the optimal assignment by brute force (adjusted to return item->bidder)
optimalAssignment :: PayoffMatrix -> Assignment
optimalAssignment matrix = maximumBy (comparing totalPayoff) assignments
  where
    bidders = [0 .. length matrix - 1]
    items = bidders -- assume square matrix
    assignments = [Map.fromList (zip items perm) | perm <- permutations bidders]
    totalPayoff assignment = sum [matrix !! b !! i | (i,b) <- Map.toList assignment]
```

tests.hs

```haskell
module Main (main) where

import SequentialAuction (auctionAlgorithm)
import JacobiAuction (jacobiAuctionAlgorithm)
import GSAuction (gsAuctionAlgorithm)
import qualified Data.Map as Map
import Control.Monad (unless)
import System.Random (mkStdGen, randomRs, StdGen, split)
import System.Environment (getArgs, getProgName)
import System.Exit (die)
import System.IO.Error (catchIOError)

type Bidder = Int
type Item = Int
type PayoffMatrix = [[Double]]
type Assignment = Map.Map Item Bidder

roundToTenths :: Double -> Double
roundToTenths x = fromIntegral (round (x * 10)) / 10

printMatrix :: PayoffMatrix -> IO ()
printMatrix m = do
    putStrLn "Price␣matrix:"
    mapM_ (putStrLn . formatRow . map roundToTenths) m
  where
    formatRow :: [Double] -> String
    formatRow row = "[" ++ unwords (map show row) ++ "]"


printAuctionResults :: PayoffMatrix -> Assignment -> Double -> IO ()
printAuctionResults matrix assignment totalPayoff = do
    putStrLn "\nAssignments␣and␣payoffs:"
    let payoffBreakdown = [(item, bidder, matrix !! bidder !! item) | (item, bidder) <- Map.toList assignment] -- !! is same as matrix[bidder][item]
    mapM_ (\(i, b, p) -> putStrLn $ "Item␣" ++ show i ++ "␣->␣Bidder␣" ++ show b ++ ":␣" ++ show (roundToTenths p)) payoffBreakdown
    putStrLn $ "\nTotal␣payoff:␣" ++ show (roundToTenths totalPayoff)

runAlgorithm :: (Double -> PayoffMatrix -> (Assignment, Double)) -> PayoffMatrix -> IO (Assignment, Double)
runAlgorithm algorithm matrix = do
```

```
    let (assignment, totalPayoff) = algorithm 0.01 matrix -- always assume 0.01 is sufficient for epsilon
    return (assignment, totalPayoff)

main :: IO ()
main = runProgram `catchIOError` \_ ->
    die "ERROR, try making sure the command-line arguments are formatted correctly"

runProgram :: IO ()
runProgram = do
    args <- getArgs
    case args of
        [sizeStr, algStr] -> do
            let maybeSize = reads sizeStr :: [(Int, String)] -- read the input, and cast as Int, String tuple
            case maybeSize of
                [(n, "")] -> do
                    algFunc <- case algStr of
                                    "seq"    -> return auctionAlgorithm
                                    "gs"     -> return gsAuctionAlgorithm
                                    "jacobi" -> return jacobiAuctionAlgorithm
                                    _        -> die "ERROR, please enter 'seq', 'gs', or 'jacobi'"
                    let seed = 100 -- causes generated matrix to stay the same if file isn't reloaded
                        gen = mkStdGen seed
                        (matrix, _) = generateMatrix gen n n
                    (assignment, totalPayoff) <- runAlgorithm algFunc matrix

                    -- show the results when the matrix is resonably small
                    -- assume that this case is used for testing correctness by hand
                    if n < 6
                        then do
                            printMatrix matrix
                            printAuctionResults matrix assignment totalPayoff

                        -- assume that this case is used for testing runtime on large matrices
                        else do
                            putStrLn $ "Total payoff: " ++ show (roundToTenths totalPayoff)
                _ -> do
                    pn <- getProgName
                    die $ "ERROR, Invalid command line arguments. Usage: " ++ pn
        _ -> do
            pn <- getProgName
            die $ "ERROR, Usage: " ++ pn


generateMatrix :: StdGen -> Int -> Int -> (PayoffMatrix, StdGen)
generateMatrix gen rows cols = (matrix, finalGen)
    where randomNumbers = take (rows * cols) $ randomRs (0.0, 100.0) gen
          (finalGen, _) = split gen
          matrix = chunksOf cols randomNumbers

-- splits the randomNumbers list into chunks of size cols
chunksOf :: Int -> [a] -> [[a]]
chunksOf _ [] = []
chunksOf n xs = let (ys, zs) = splitAt n xs in ys : chunksOf n zs
```