Parallel Auction Algorithm

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

12/18/2024

The Assignment Problem

The assignment problem is one of classic combinatorial optimization problems and is closely related to a wide range of important problems, such as min-cost network flow, weighted matching problem, and linear programming.

The problem can be described as follows:

- Given n persons and n items
- Given a person-item payoff matrix

$\Rightarrow$

What is the optimal assignment in which everyone gets exactly one item?

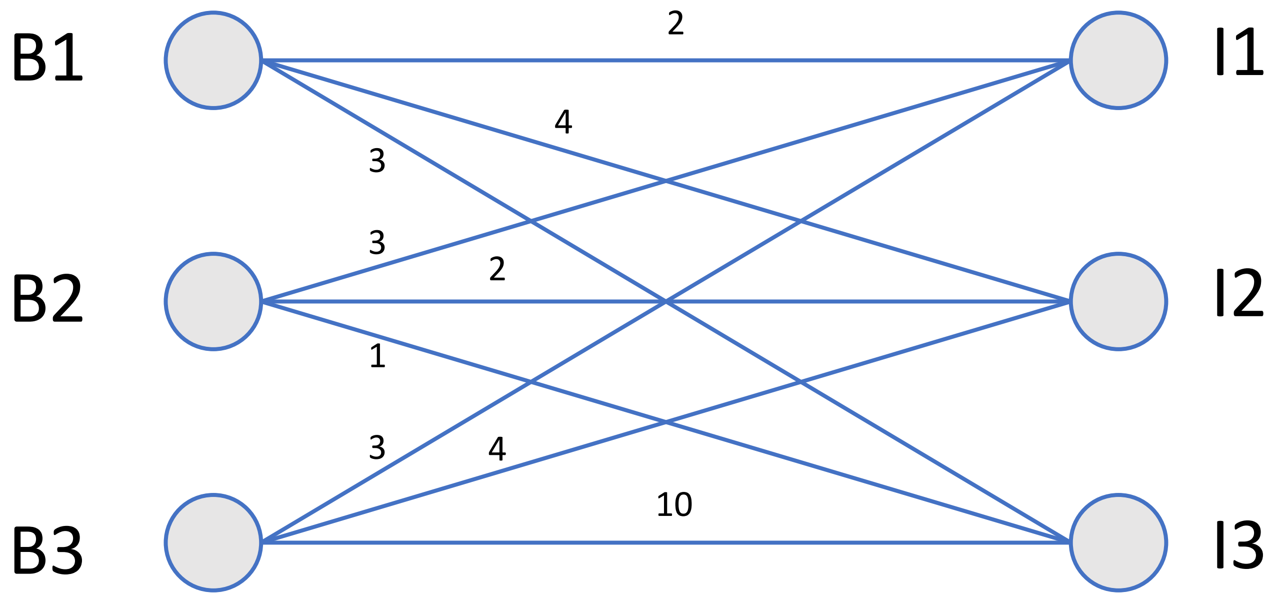
A simple example:

- 3 bidders (B1, B2, B3)
- 3 items (I1, I2, I3)
- 1 payoff matrix (each entry [i, j] represents the payoff of assigning item Ii to bidder Bj):

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

The Assignment Problem

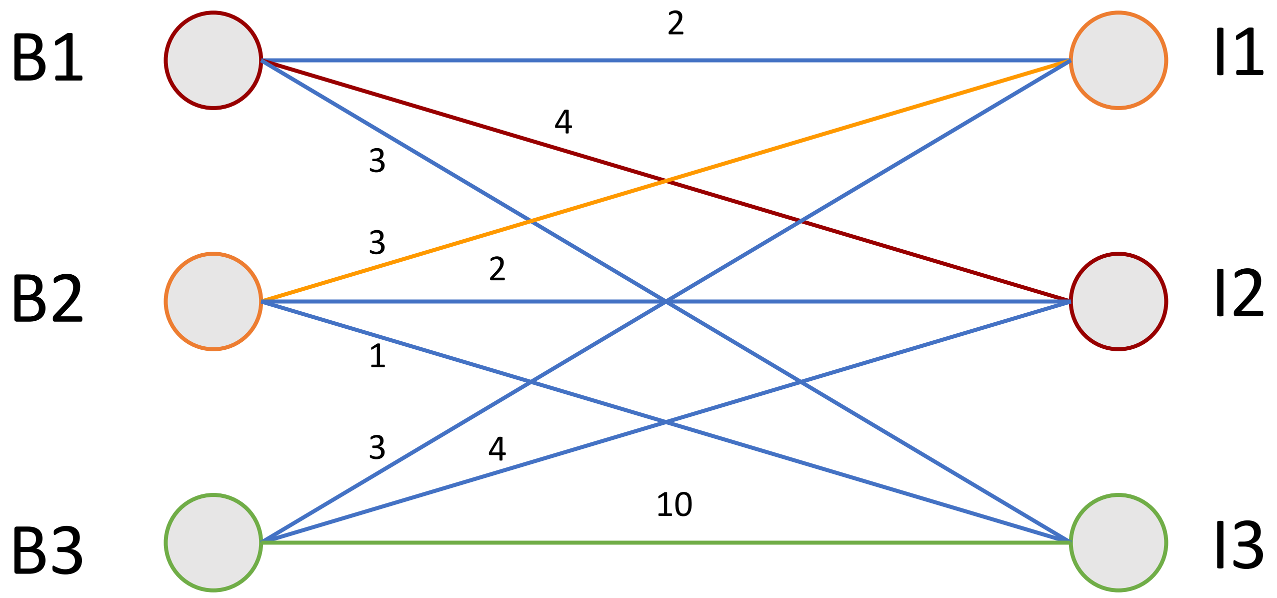
The graph implementation of this problem is: given a bipartite graph G = (V, E) with bipartition (A, B) and weight function $w : E \to \mathbb{R}$tching of maximum weight, where the weight of a matching M is given by $w(M) = \sum_{e \in M} w(e)$



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

The Assignment Problem

The graph implementation of this problem is: given a bipartite graph G = (V, E) with bipartition (A, B) and weight function $w : E \to \mathbb{R}$tching of maximum weight, where the weight of a matching M is given by $w(M) = \sum_{e \in M} w(e)$



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

The Assignment Problem

The assignment problem is one of classic combinatorial optimization problems and is closely related to a wide range of important problems, such as min-cost network flow, weighted matching problem, and linear programming.

The problem can be described as follows:

- Given n persons and n items
- Given a person-item payoff matrix

→

What is the optimal assignment in which everyone gets exactly one item?

A simple example:

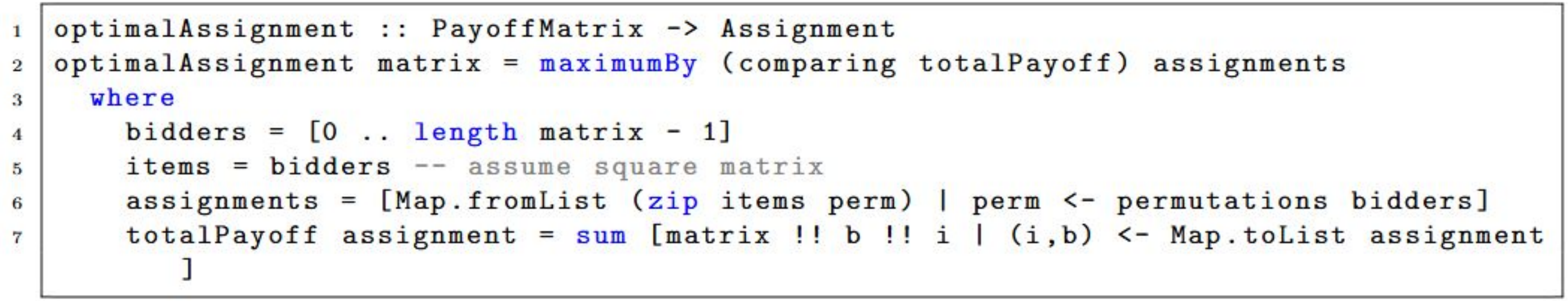
- 3 bidders (B1, B2, B3)
- 3 items (I1, I2, I3)
- 1 payoff matrix (each entry [i, j] represents the payoff of assigning item Ii to bidder Bj):

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

The Auction Algorithm

Brute-force Implementation:

- Generate all possible permutations of assignments
- Calculate the total pay-off for each permutation
- Identify the max total pay-off



```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
       ]
```

The Auction Algorithm

Algorithm from Jin[1]:

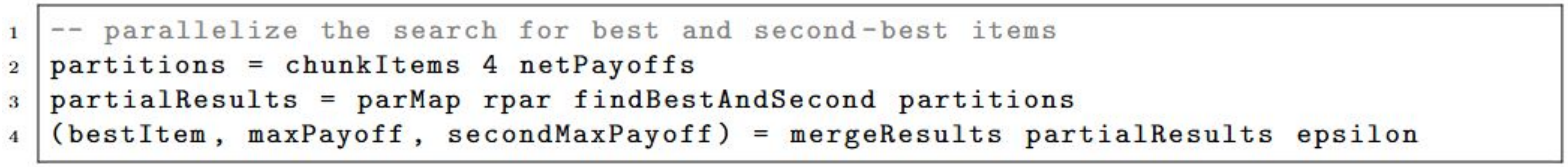
- Initialize unassigned bidders set *U* to all bidders and prices to 0
- Pick any bidder *i* from *U*. Search for the item *j* that gives the highest net payoff and second highest net payoff
- Update the price of item *j* as $p_j \leftarrow p_j + (A_{ij} - p_j - (A_{ik} - p_k))$
- Assign item *j* to bidder *i*. If *j* was previously assigned to another bidder *s*, remove the assignment and add *s* back to *U*
- If *U* becomes empty, the algorithm terminates. Otherwise, return to step 2

[1] https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/jin.pdf

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. The parallelization part include the following steps:

- Divides the items among p threads
- Each thread search its partition independently
- Merge the searching result

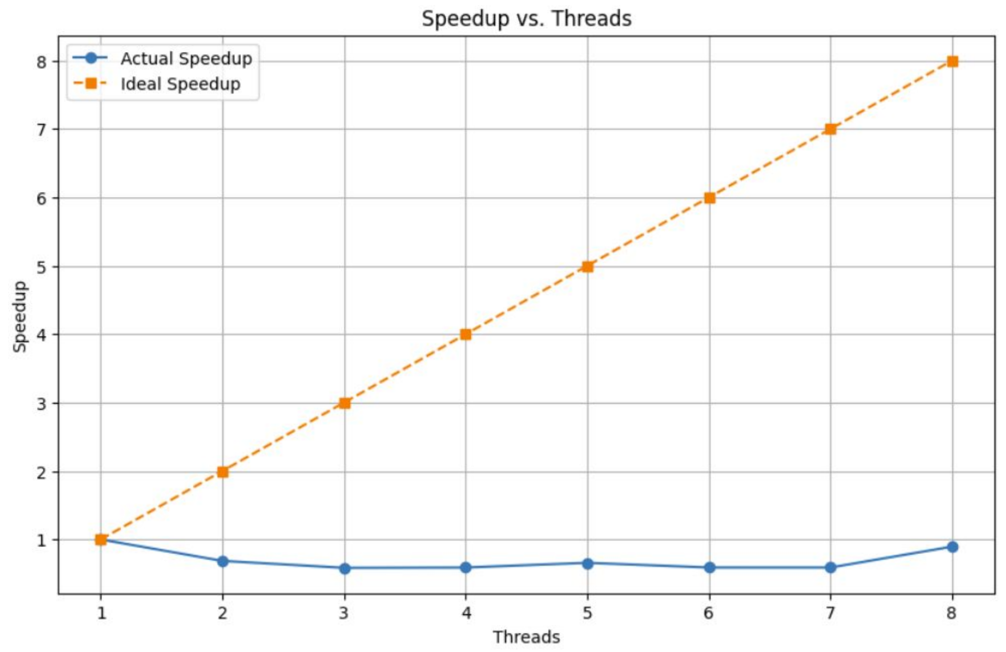


```haskell
-- parallelize the search for best and second-best items
partitions = chunkItems 4 netPayoffs
partialResults = parMap rpar findBestAndSecond partitions
(bestItem, maxPayoff, secondMaxPayoff) = mergeResults partialResults epsilon
```

Parallelization: Gauss-Seidel

This parallelization is intuitive. 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.



COLUMBIA | ENGINEERING The Fu Foundation School of Engineering and Applied Science

Parallelization: Jacobi

The Jacobi version parallelizes step 3 of the algorithm. Allows multiple bidders to search for their bids simultaneously.

What if two or more bidders make bids for the same item on one iteration?

- A synchronization stage is needed to make sure this conflict does not happen, since the prices used to search for the best item may be outdated
- Interestingly, it has been proven 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. Thus, the synchronization can be avoided

Parallelization: Jacobi

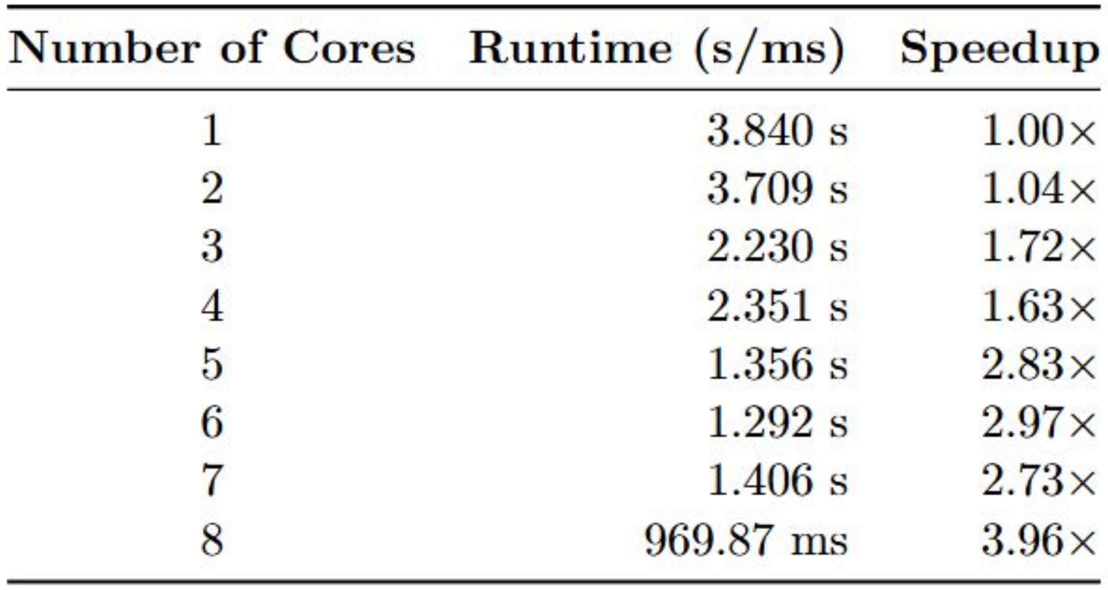
Here is how we implement the asynchronous Jacobi version:

- *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

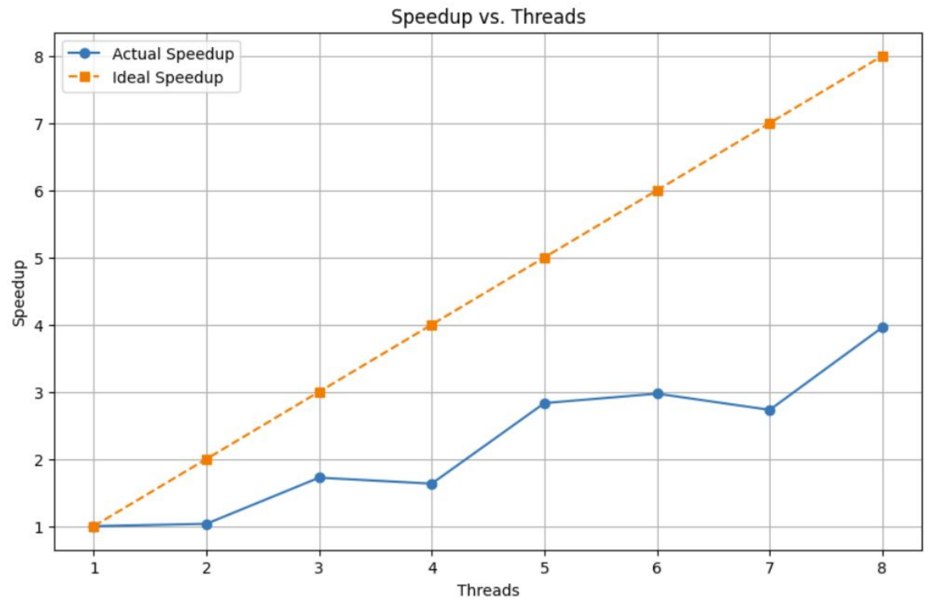
```
synchronizedParallelBidding :: [Bidder] -> Prices -> [(Bidder, Item, Double)]
synchronizedParallelBidding bidders prices =
  map (bestBid prices) bidders ‘using‘ parList rdeepseq
```

Parallelization: Jacobi

Since the Jacobi version avoids the merging overhead present in the Gauss-Seidel version, it offers more interesting results.
Here is the runtime/speedup of Jacobi implementation executed on different number of cores, for a test case of 1000x1000 payoff matrix

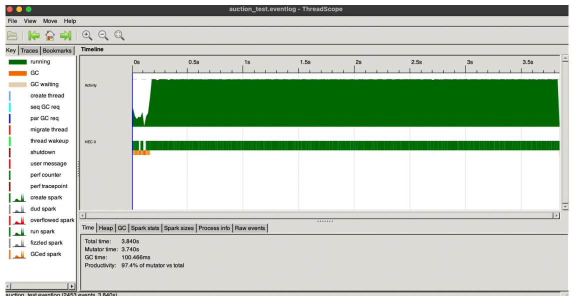


Number of Cores	Runtime (s/ms)	Speedup
1	3.840 s	1.00×
2	3.709 s	1.04×
3	2.230 s	1.72×
4	2.351 s	1.63×
5	1.356 s	2.83×
6	1.292 s	2.97×
7	1.406 s	2.73×
8	969.87 ms	3.96×

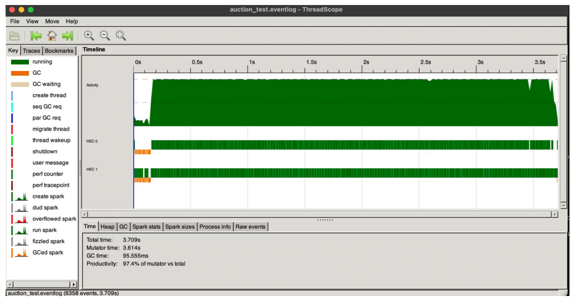


COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

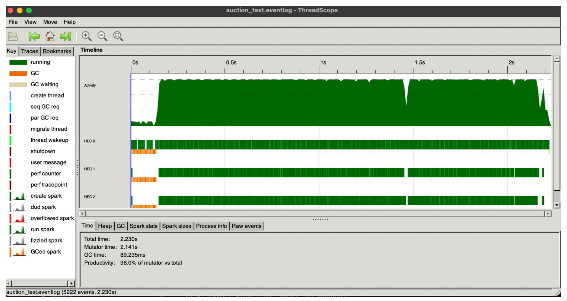
Parallelization: Jacobi



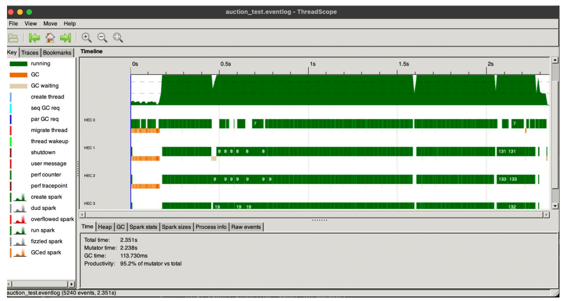
```
Total time:    3.840s
Mutator time:  3.740s
GC time:       100.466ms
Productivity:  97.4% of mutator vs total
auction_test.eventlog (2453 events, 3.840s)
```



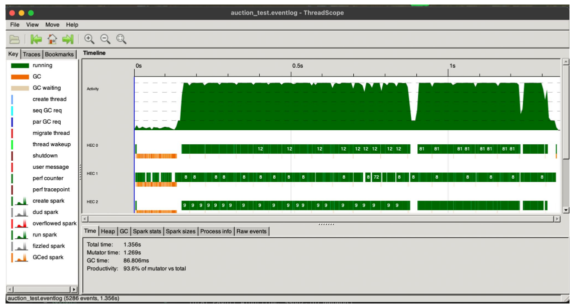
```
Total time:    3.709s
Mutator time:  3.614s
GC time:       95.555ms
Productivity:  97.4% of mutator vs total
auction_test.eventlog (6358 events, 3.709s)
```



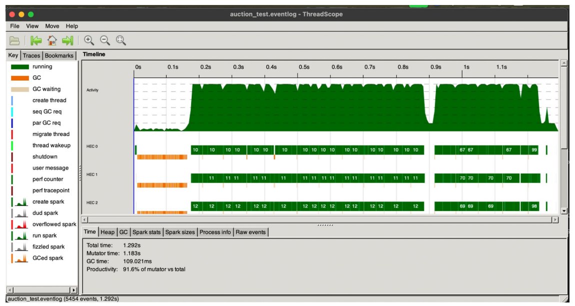
```
Total time:    2.230s
Mutator time:  2.141s
GC time:       89.235ms
Productivity:  96.0% of mutator vs total
auction_test.eventlog (5222 events, 2.230s)
```



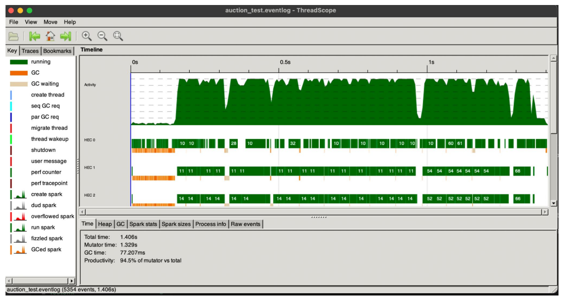
```
Total time:    2.351s
Mutator time:  2.238s
GC time:       113.730ms
Productivity:  95.2% of mutator vs total
auction_test.eventlog (5240 events, 2.351s)
```



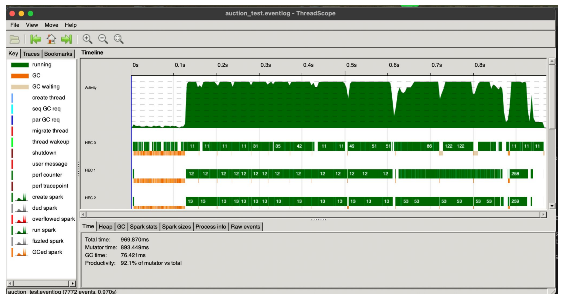
```
Total time:    1.356s
Mutator time:  1.269s
GC time:       86.806ms
Productivity:  93.6% of mutator vs total
auction_test.eventlog (5286 events, 1.356s)
```



```
Total time:    1.292s
Mutator time:  1.183s
GC time:       109.021ms
Productivity:  91.6% of mutator vs total
auction_test.eventlog (5454 events, 1.292s)
```



```
Total time:    1.406s
Mutator time:  1.329s
GC time:       77.207ms
Productivity:  94.5% of mutator vs total
auction_test.eventlog (5354 events, 1.406s)
```

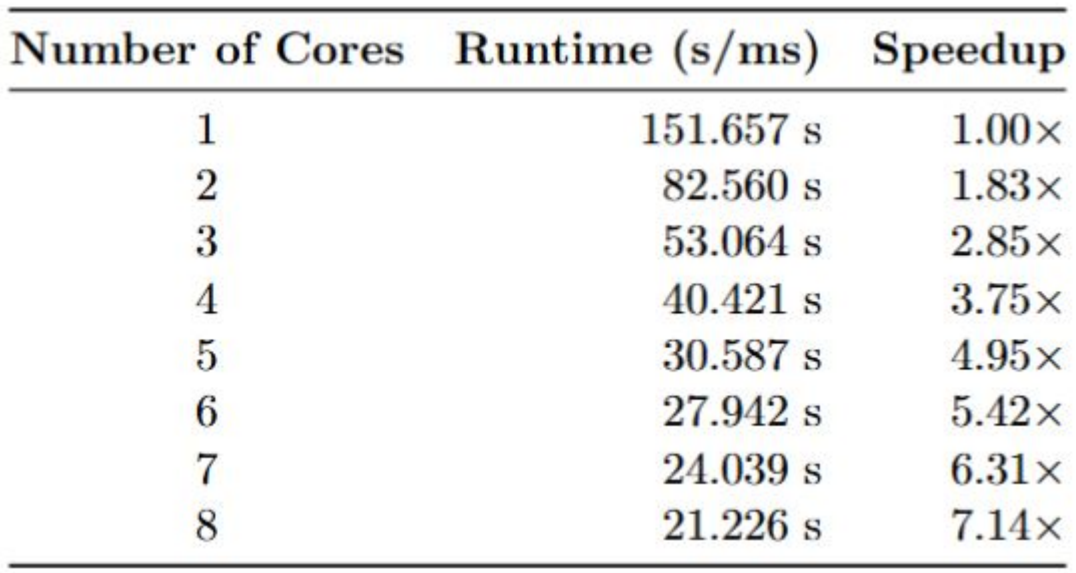


```
Total time:    969.870ms
Mutator time:  893.449ms
GC time:       76.421ms
Productivity:  92.1% of mutator vs total
auction_test.eventlog (7772 events, 0.970s)
```

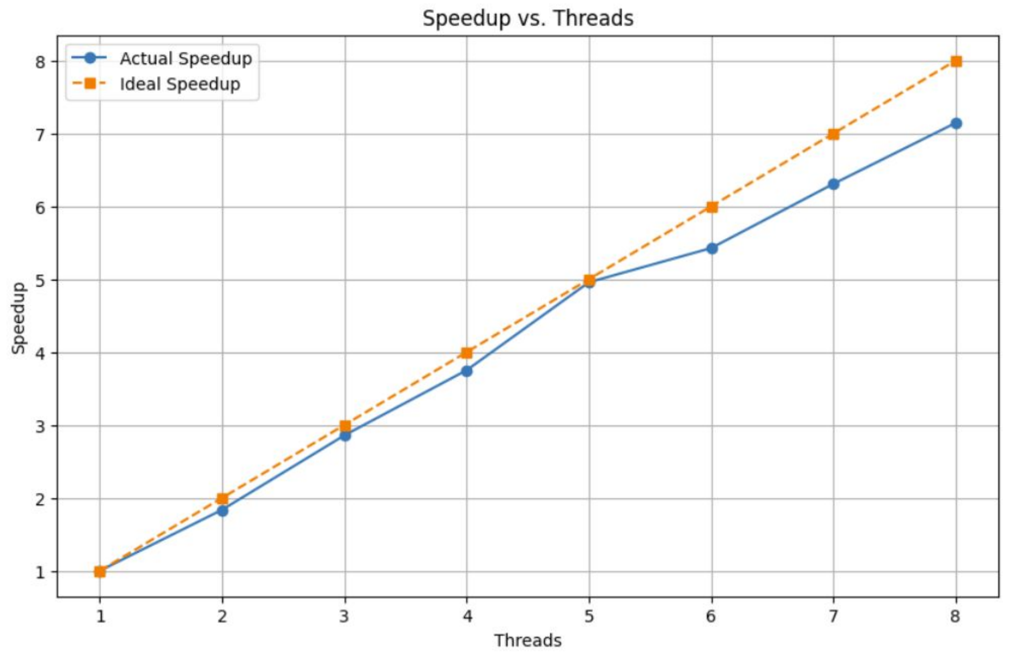
- The productivity measures for all numbers of cores are above 90%, signaling efficient core usage.
- However, the speedup is not ideal, since the test dataset is not large enough

Parallelization: Jacobi

To illustrate more of the benefit of this parallelization, we test it over a 3000x3000 test case (pseudo-random matrix generation with fixed seed). Here is the runtime/speedup of it executed on different number of cores. As the matrix size becomes larger, the speedup is more apparent



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×



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.

Parallel Auction Algorithm

Thanks!