Distributed Stable Marriage with Incomplete List and Ties using Spark

Yilong Geng
gengyl08@stanford.edu

Mingyu Gao
mgao12@stanford.edu

1 Introduction

The classic stable marriage (SM) problem can be stated as below: Given n men and n women, where each person has ranked all members of the opposite sex with a unique number between 1 and n in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are "stable". Gale and Shapley gave their famous deferred acceptance algorithm in 1962 which always finds a stable matching [4].

When the preference lists are incomplete and/or have ties, the problem becomes more challenge (called *Stable Marriage with Incomplete List and Ties*, SMTI). For SMTI, the GS algorithm still finds a stable matching. However, in this case, we are usually interested in finding a stable matching with maximum size. While the matching size of the GS algorithm is at least half of the optimal matching size, a modified version of the GS algorithm was proposed to guarantee 2/3 of the optimal matching size [3].

Algorithms for finding solutions to the stable marriage problem have applications in a variety of real-world situations. A well known example is in the assignment of graduating medical students to their first hospital appointments. In 2012, the Nobel Prize in Economics was awarded to Lloyd S. Shapley and Alvin E. Roth for *the theory of stable allocations and the practice of market design* [1].

In this report we will introduce the modified version of the GS algorithm and our implementation using Spark [2].

2 Stable Marriage and the GS Algorithm

The GS (Gale and Shapley) algorithm involves a number of iterations of proposals, acceptance and rejection. In each iteration,

1. each unengaged man proposes to the most-preferred woman to whom he has not yet proposed (regardless of whether the woman is already engaged), and then

2. each woman gets engaged to the proposer she most prefers (her existing fiance or someone else) and rejects the rest (perhaps including her current fiance).

The runtime complexity of this algorithm is $O(n^2)$ where n is number of men or women.

Below is the pseudocode for the GS algorithm.

Algorithm 1 The Gale Shapley Algorithm

```
Initialize all m and w to free
while there is a free m who has non-empty list do
    w = top in the list
    if w is free then
        mw engage
    else
        Some m'w already engaged
        if w prefers m to m' then
            mw engage, m' becomes free
            Remove w from m''s list
        else
            m'w remain engaged, remove w from m's list
        end if
    end if
end while
```

3 MAX-SMTI and Modified GS Algorithm

When the preference lists of the men and women can be incomplete and can have ties, not everybody will necessarily get married at the end of the GS algorithm. In fact, the GS algorithm takes $O(|E|)$ time complexity to complete on a single machine where E is the set of edges in the bipartite graph of men and women. And the GS algorithm guarantees the resulting matching size is at least half of the optimal matching size.

A modified GS algorithm was proposed to achieve better matching size with the same time complexity. The details can be found in [3]. Here we only outline the main modifications to the basic GS algorithm:

- The preference list of each un-engaged man will be traversed twice before the man stops proposing.
- A man will prefer un-engaged women to engaged women if they have the same priority in his preference list.
- If a women's fiance has a women in his preference list who he actually prefers to her, she will always accept a new proposal. And when her current fiance gets rejected from her, he will not remove her from his list in this case.

With these modifications, the new algorithm can actually guarantee at least 2/3 of the optimal matching size [3].

4 Distributed Algorithm for SMTI

Given the single-machine algorithms for solving SM and SMTI problems, now we propose their distributed versions. We begin with a description of the distributed data structure we use. As the Modified GS algorithm for SMTI is a variant of the basic GS algorithm, we first describe the distributed basic GS algorithm, and then we outline the difference of the modified GS algorithm. We will use the word "proposer" and "acceptor" instead of man and woman in the following sections.

4.1 Data Structures

To scale with the problem size, i.e., the number of proposers and acceptors, we need to represent their data structures in a distributed way, by using distributed collections (`RDD`s in Spark). The data for each proposer or acceptor include his/her preference list, and necessary status for engaged or single. Note that while the status is $O(1)$, the preference list can be long, in the worst case $O(n)$, and may not fit in one single machine. However, for most real SMTI problems, the preference list is incomplete and sparse, i.e., each person may only specify the rank for a few other people he is interested in. Thus the preference list is usually much shorter than n, and is able to fit in a single machine.

4.2 Distributed Basic GS Algorithm

4.2.1 Algorithm description

As previously described, the basic GS algorithm iterates over each active single proposer to propose to his current favorite acceptor. To make this algorithm parallel, first we let all the active single proposers send their proposals to their favorite acceptors simultaneously in each iteration. Each acceptor will then receive none, one, or more than one proposals. The acceptor picks up the highest-ranked proposal according to her preference list and compares with her current fiance, to choose the new fiance. The other proposals are all rejected.

Note that the original basic GS algorithm does not require a particular order for the proposers. This gives the freedom to the acceptors to handle the received proposals in any order, as long as they pick the highest-ranked one. Therefore proposing simultaneously is equivalent to proposing serially and will obtain the same result as the original algorithm.

After the proposing phase, the acceptors need to announce their decisions and current status to the proposers. This is being done in the responding phase, where each acceptor sends response to her current fiance (if there is one). The proposers who receive responses know they

are engaged, either by proposing successfully or by keeping the relation. The proposers who do not receive responses are single, either by breaking up or by failed proposing. Therefore they can update their status based on the responses.

Algorithm 2 sketches the complete algorithm scheme for the distributed GS algorithm.

Algorithm 2 Distributed GS algorithm scheme

```
Initialize proposers and acceptors
while there are active proposers do
    ▷ Proposing phase
    proposals = proposers.makeProposals()
    acceptors.handleProposals(proposals)
    ▷ Responding phase
    responses = acceptors.makeResponse()
    proposers.handleResponses(responses)
end while
```

4.2.2 Complexity analysis

It is easy to see that there are two rounds of communication happen in each iteration. As each proposer/acceptor can at most make one proposal/response, the communication cost is $O(n)$. Since engaged and inactive proposers do not make proposals, and single acceptors do not make responses, $O(n)$ is the upper bound. In fact as most proposers will become engaged after a few iterations, the communication cost drops rapidly after the first few iterations.

The maximum reduce size (the maximum number of proposals/responses each person receives) is limited by the length of the preference list. As we discussed before, the preference list is usually short, thus it will not become a bottleneck in real problems.

4.3 Distributed Modified GS Algorithm

The modified GS algorithm for SMTI problem uses the same scheme as in Algorithm 2. The only difference is in how proposers/acceptors updates their status and how they make proposals/responses, according to the algorithm difference highlights in Section 3. Among the three points, the first one is easy to modified as it only involves local operations. The other two require status cross-check between proposer and acceptor, so we need additional messages.

- For the second point, acceptor who is not engaged will send a special "response" to *all* proposers in her list to announce her singleness, in order to let proposers distinguish between single and engaged acceptors. This only adds moderate cost, as discussed before that most acceptors will be engaged after the first few iterations.

Listing 1: GS algorithm scheme

```
while ( hasActiveProposers ) {
        // Proposing phase
        val proposals = proposers
                .flatMap( makeProposal )
                .groupByKey()
                .cache()

        acceptors = acceptors
                .leftOuterJoin(proposals)
                .mapValues( handleProposal )
                .cache()

        // Responding phase
        val responses = acceptors
                .flatMap( makeResponse )
                .groupByKey()
                .cache()

        proposers = proposers
                .leftOuterJoin(responses)
                .mapValues( handleResponse )
                .cache()

}
```

- For the third point, proposer who satisfies the condition (i.e., has an acceptor in his list who he prefers to his fiancee) will also send a special "proposal" to his fiancee to announce this information. Note that in the basic algorithm he does not need to send proposal, so each proposer still sends at most one proposal.

To summarize, the distributed modified GS algorithm only adds moderate communication cost to the basic algorithm. In average the complexity keeps the same.

5 Implementation

We implement both the distributed basic GS algorithm and the distributed modified GS algorithms in Spark. The pseudocode is shown in Listing 1. It directly follows the algorithm scheme in Algorithm 2.

An interesting discussion is whether we can use Pregel API [5] to implement the GS algorithm scheme. The two groups of people form a (sparse) bipartite graph with edges as relations in their (incomplete) preference lists, and the proposing and responding phases are similar

to the scatter-gather communication pattern. Pregel provides a simple API to implement graph algorithms, and also it handles partitioning better to reduce communication between RDD partitions. However, we finally decide to write our own implementation due to the following reasons. First, proposers and acceptors have different attributes. Second, at most time each person (vertex) only sends message along one edge in each iteration, where Pregel by default supports sending messages along all edges, which is an overkill for us and may have more communication cost. Third, we have two rounds of message passing in one iteration in reverse direction, which is hard to distinguish in Pregel.

Another issue with the iterative algorithms in Spark is the long lineage of `RDD`s. Spark guarantees fault-tolerance for `RDD`s. As in each iteration we generate new `RDD`s from the previous `RDD`s, the dependency lineage keeps growing. To break this dependency, we use checkpointing. We empirically find that after about 20 iterations, the program becomes obviously slower due to the dependency lineage overhead. Thus, we checkpoint every 20 iterations. After the checkpoint, the program runs at regular speed.

6 Experiments

We evaluate our distributed modified GS algorithm locally on a dual-socket server machine, which has 12 physical cores and supports 24 threads in total. The dataset we use is a random-generated public dataset (1000 men vs. 1000 women) [6]. The algorithm takes about 700 iterations to complete. We run the algorithm for 10 iterations for 3 times, and report the average execution time.

Figure 1 shows the performance scaling with number of working threads using fixed 64 RDD partitions. This keeps the communication between partitions unchanged. We can see that the performance scales very well up to 8 to 16 threads.

Figure 2 shows the same performance numbers, but with number of partitions equal to number of working threads. We can see now as more partitions are used, the communication cost increases significantly and overweight the parallelism performance gain. The GS algorithm is a high-communication algorithm. To improve the scalability, better algorithms are needed.

7 Conclusion

Stable marriage is a famous problem with wide applications. When the preference lists are incomplete and/or have ties, it becomes more challenging, and algorithms must also optimize for maximum marriage size. Kiraly proposed an algorithm which modified the original Gale-Shapley algorithm, and gave the state-of-the-art quality as 2/3 of the optimal marriage size. We implement both the basic GS algorithm and Kiraly's algorithm in a distributed environment using Spark. The communication cost and the reduce size are both moderate. We also test the implementation locally and see reasonable scaling performance.

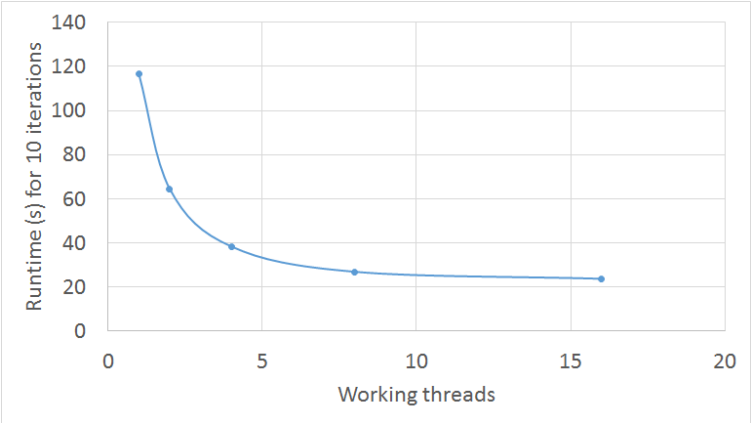

Figure 1: Performance scaling with fixed number of RDD partitions.

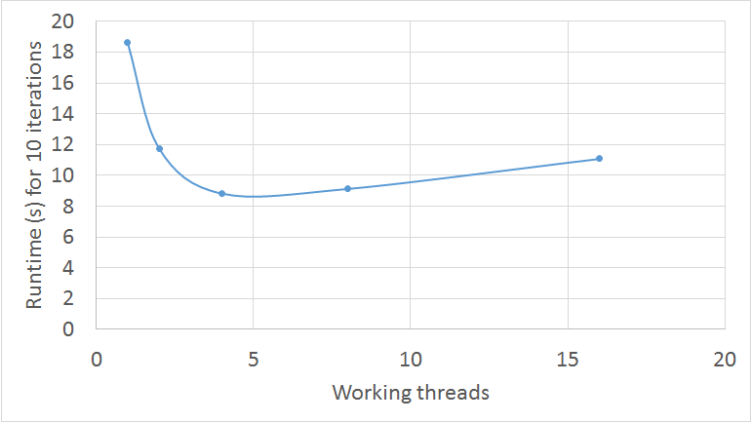

Figure 2: Performance scaling with same number of RDD partitions as number of threads.

References

[1] Stable Marriage Problem. http://en.wikipedia.org/wiki/Stable_marriage_problem.

[2] Apache Spark. https://spark.apache.org/.

[3] Kiraly, Z.. *Linear Time Local Approximation Algorithm for Maximum Stable Marriage*. Algorithms 2013.

[4] Gale, D., Shapley, L. S.. *College Admissions and the Stability of Marriage*. American Mathematical Monthly.

[5] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G.. *Pregel: A System for Large-Scale Graph Processing*. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (pp. 135-146). ACM.

[6] SMTI with Adaptive Search – support materials. http://cri-hpc1.univ-paris1.fr/smti/.