Parallelization of Gale and Shapley Algorithm using CUDA

Hemant Gaur
Computer Science
IIT Bhilai
Bhilai, Chhattisgarh
hemantgaur@iitbhilai.ac.in

Hrishik Kanade
Computer Science
IIT Bhilai
Bhilai, Chhatisgarh
hrishikrajeshkanade@iitbhilai.ac.in

Parth Khandenath
Computer Science
IIT Bhilai
Bhilai, Chhattisgarh
khandenathparth@iitbhilai.ac.in

***Abstract*—In this research paper we are exploring the parallelization of the stable matching algorithm, also called Gale-Shapley [GS62] algorithm, on Graphics Processing Units (GPUs). GPUs has ability to do large computation parallelly, thus reducing the execution time and computation overhead. We are aiming to enhance the efficiency and speed of the matching process, that is the most computationally expensive task, by leveraging the massively parallel architecture of GPUs. We are proposing a novel parallelized algorithm written in NVIDIA CUDA, tailored to the characteristics of the Stable-Matching algorithm, expecting significant performance improvements compared to traditional CPU-based implementations. Our experimental results and algorithm designed will showcase the scalability and effectiveness of the GPU based approach, paving the way for accelerated stable matching in various application domains including Job Matching, College Matching(for admissions eg. JOSAA counselling), etc. We look at the Gale and Shapley algorithm for computing stable matchings in a parallel setting. We present our attempts to parallelize in CUDA along with observations.**

I. Introduction

The stable matching problem is one of the classic problems in combinatorial optimization which are extensively studied in both theoretical and practical contexts. The problem has various applications in various domains, including college admissions, job recruitment, and kidney exchange programs. It was first introduced by Gale and Shapley in their seminal paper in 1962 [GS62].

The Gale and Shapley algorithm provides an interesting solution to the stable matching problem by guaranteeing a stable matching between two sets of agents/ genders(in case of stable marriage problem) with preferences over each other. The computational complexity of finding a stable matching increases as the size of the problem grows which then leads to longer execution times for sequential implementations.

In this proposal, we propose to parallelize the Gale and Shapley algorithm with the help of CUDA which is a parallel computing platform and programming model developed by NVIDIA. By using the parallel processing capabilities of GPUs, our aim is to accelerate the computation of stable matchings, which are suitable particularly for large-scale instances.

The research on stable marriage problems has mainly focused on theory and mathematical rigor previously. Several recently published algorithms for computing greedy matchings were in fact special cases of classical algorithms for stable marriage problems. Earlier work proposed a new approach called Parallel Iterative Improvement (PII), where algorithm is implemented in OpenMP using n^2 processing elements (PEs), to obtain an O(n log n) running time compared to the sequential Gale and Shapley algorithm, which is O(n^2). However their implementation required O(n^2) PEs, its not scalable in a GPU context since we have limited hardware resources.

Algorithm 1 Gale and Shapley algorithm [1]

```
1:  Input : $G = (\mathcal{M} \cup \mathcal{W}, E)$
2:  Start with all $m \in \mathcal{M}$, $w \in \mathcal{W}$ as free, $M = \phi$
3:  do
4:      if $\exists m \in \mathcal{M}, M(m) = \phi$, $\exists w \in \mathcal{W}_m$, s.t. $w$ is most preferred & not applied by $m$ then
5:          if $h$ is unmatched then
6:              $M = M \cup \{(m, w)\}$
7:          else
8:              Let $m'$ be the current partner of $w$
9:              if $m' >_w m$ then
10:                 $m$ continues applying to next woman in his list (if it exists)
11:             else
12:                 $M = (M \cup \{(m, w)\}) \setminus \{(m', w)\}$
13:                 $m'$ continues applying to next woman in his list (if it exists)
14:     else
15:         Exit the loop.
16: while true
17: Return $M$
```

In a stable marriage problem we have two finite sets of players, conveniently called the set of men (M) and the set of women (W). We assume that every member of each set has strict preferences over the members of the opposite sex. In the rejection model, the preference list of a player is allowed to be incomplete in the sense that players have the option of declaring some of the members of the opposite sex as unacceptable; in the Gale-Shapley model we assume that preference lists of the players are complete. A matching is just a one-to one mapping between the two sexes; in the rejection model, we also include the possibility that a player may be unmatched, i.e. the player's assigned partner in the matching is himself/herself. The matchings of interest to us are those with the crucial stability property, defined as follows: A matching μ is said to be unstable if there is a man-woman pair, who both prefer each other to their (current) assigned partners in μ; this pair is said to block the matching μ, and is called a blocking pair for μ. A stable matching is a matching that is not unstable. [TST99]

II. Approach

A. Sequential

Sequential code solves the Stable Marriage Problem using the Gale-Shapley algorithm. This algorithm allows us to find stable pairs for men and women based on their preference arrays. The problem is solved by reading prefs arrays, forming queue proposals, and processing them one by one compared with current partners. The considered code contains functions for dynamic memory allocation and the most basic operations with data. The code efficiently solves the problem and forms stable pairs for all individuals.

B. Parallel approach

The idea behind parallel implementation was to have a separate thread for each man, which can apply to women independently of other men at the same time. Thus we are expecting a increase in the overall process use and decrease the execution time. For our preliminary purpose, one kernel call would perform exactly one proposal per unmatched man. We repeatedly perform the kernel call, until all the men are matched (this is guaranteed since we have equal number of men and woman and a complete bipartite setting. As explained earlier in the parallelization intuition, we need to serialize the proposals happening to the same woman in a given time-step. For this purpose, we also implemented a lock using atomicCAS function.

1. Initialization Function (`initialize`):

- This function sets up the initial state for the matching process. Think of it as preparing the stage before the main performance begins.
- It assigns starting values to various arrays used to keep track of preferences and matches for men and women.
- For instance, men and women are initially set as unmatched, and men start with their first preference while women are unlocked.

2. CUDA Kernel Function (`stable_matching`):

- Imagine this as the heart of our matchmaking process. Each thread represents a hopeful suitor (a man) trying to find their best match among the available women.
- Threads run in parallel, each exploring a different man's preferences simultaneously, speeding up the overall process.
- To ensure fairness and prevent conflicts, we use special techniques like atomic operations and locks, which act like bouncers making sure only one suitor talks to a woman at a time.
- The process continues until all matches are stable, meaning no man or woman wants to change their partner.

3. Main Function:

- Here, we handle the administrative tasks of our matchmaking event.
- We start by gathering information about the number of participants (men and women) from our audience (the user).
- Then, we allocate memory both on the host (CPU) and the guest of honor (GPU) to store everyone's preferences and match status.
- After setting the stage, we collect each participant's preferences and rankings, creating the perfect environment for matchmaking.
- With everything set, we launch our matchmaking process using CUDA, letting the GPU do the heavy lifting.
- Once the matches are made, we bring the results back to the audience for everyone to see.

4. Performance Measurement:

- Like keeping an eye on the clock during a speed-dating event, we measure how long it takes to gather preferences and make matches.
- We use CUDA events, like setting a stopwatch at the beginning and end of our matchmaking process, to precisely measure time.

In the above code snippet, the matching process is handled entirely within a single CUDA kernel function (stable_matching). Each thread corresponds to a man, and the matching process iterates within the kernel until all matches are stable. This approach aims for simplicity by encapsulating the entire matching logic within the kernel.

On the other hand, we proposed a second code snippet which employs a different strategy. It separates the initialization, input handling, and matching process into distinct functions and stages. While CUDA is still used for parallel execution, the matching process is driven by a while loop in the main function. This loop iterates until all men are matched, invoking the stable_matching kernel function repeatedly.

This difference in organization affects the control flow and how parallelism is leveraged. In the first snippet, all matching logic is contained within the kernel, which might simplify code maintenance but could potentially limit flexibility and make debugging more challenging. The second snippet, with its iterative approach in the main function, provides more explicit control over the matching process and allows for easier integration with other parts of the program.

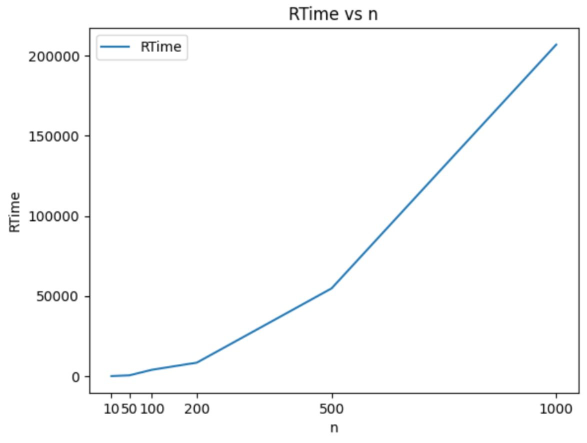
III. Evaluation Methodology

Here our aim was to compare the performance of serialized gale-shapley algorithm with that of our parallel implementation. So to do this, we generated data points for parameters like number of persons involved, number of proposals made while execution, execution time of code on gpu(device), and overall execution time. These collected data points are then used for graph plotting of parallel vs serial implementation on various parameters vs n(number of participants). We used google collab with CPU: Intel Xeon @ 2.20 GHz, RAM: approximately 13 GB and GPU: A Tesla K80 accelerator with 12 GB GDDR5 VRAM . We generated easy and hard datasets for evaluation. We used the runtimes obtained from the parallel versions implemented and compared them with runtimes obtained from sequential versions. We plotted the speedups for both the parallel algorithms.

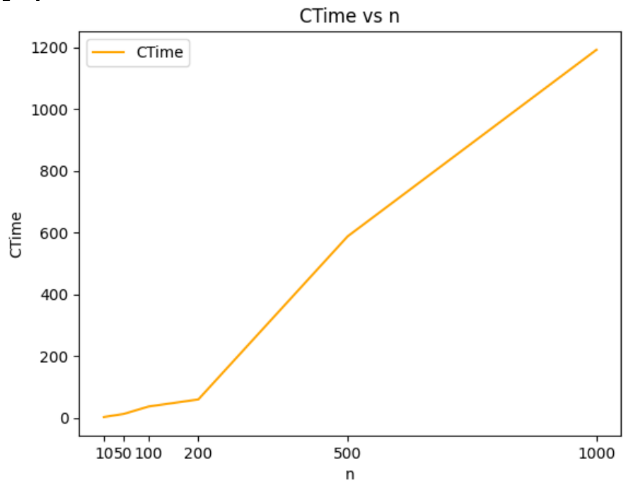
IV. Results

We have generated two datasets by running code for both sequential and parallel implementation. On running the code, we have measured the reading time of data transfer from file to arrays. After reading, the execution time for the given N is calculated. We have varied N from 10, 50, 100, 200, 500, 1000. Here are the results snippets. for both sequential and parallel implementation.

A. *Sequential*

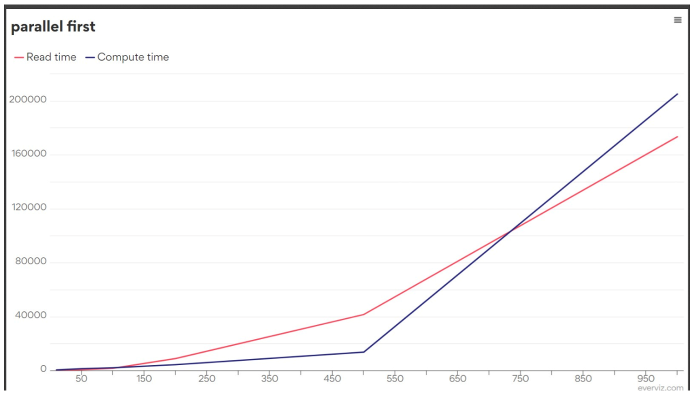


graph for Read time (in microseconds) vs n.

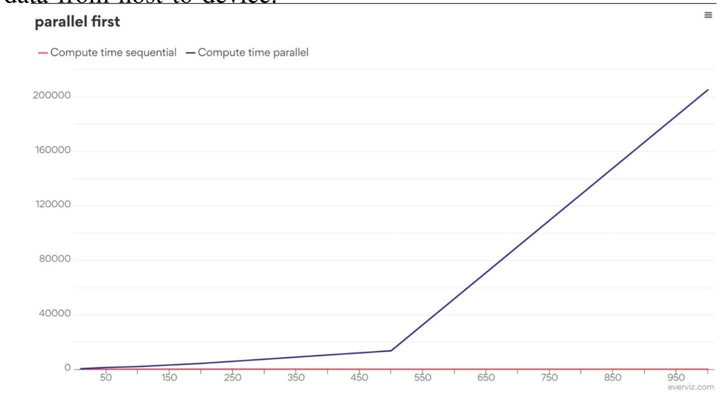


The above graph is Computation time (in microseconds) vs n. Here we can see that roughly the computation time is quadratic, which is inline with the O(n^2) time complexity of the original Gale-Shapley proposed Stable Matching Problem.

Now comparing the sequence with the performance of sequential with parallel implementation. Below are the output graphs.



the above graph shows that when huge input is required to be used in the gpu, a lot of time is spent in transferring the data from host to device.



Here we can we see that the parallel implementation doesn't performed better as compare to sequential which was the finding of the larsen et. al. The parallel approach doesn't works better compared to sequential because of the increased overhead due to communication. [Lar97]

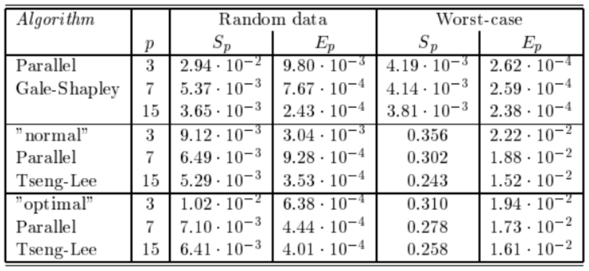
V. Conclusion

Beacuse of the massive architecture used in gpu for computation, we were expecting to develop a technique that actually outperforms the sequential code but our finding were quite in line with previous works. Earlier Jesper et.al highlighted that the parallel implementation cannot outperform sequential one's because of increased communication overhead [Lar97]. While Quinn[Qui87] argued that we cannot expect a large speedup from a parallel implementation of Gale-Shapley style algorithms in practice as the algorithm cannot run faster than the maximal number of proposals made by any one man. Thus medium sized problem can be solved by parallelizing the algorithm execution on multiple processors, as in the findings of Jasper et.al [Lar97] or another approach can be proposed in future that have improved lookup in the preference list as originally proposed. Thus, we can conclude that 'The Stable Marriage Problem' is an inherently sequential problem, and attempts to parallelize and solve it in lesser time have been mainly compromised by high communication overhead.

VI. Related Work

- A parallel approach to stable marriage problem (Jesper Larsen DIKU Department of Computer

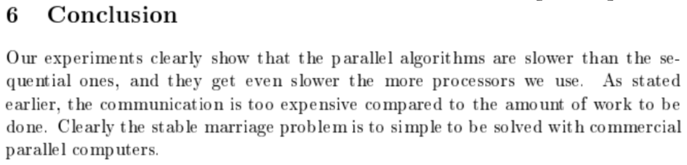
Science University of Copenhagen , 1997)



Algorithm		Random data		Worst-case	
	p	S_p	E_p	S_p	E_p
Parallel	3	$2.94 \cdot 10^{-2}$	$9.80 \cdot 10^{-3}$	$4.19 \cdot 10^{-3}$	$2.62 \cdot 10^{-4}$
Gale-Shapley	7	$5.37 \cdot 10^{-3}$	$7.67 \cdot 10^{-4}$	$4.14 \cdot 10^{-3}$	$2.59 \cdot 10^{-4}$
	15	$3.65 \cdot 10^{-3}$	$2.43 \cdot 10^{-4}$	$3.81 \cdot 10^{-3}$	$2.38 \cdot 10^{-4}$
"normal"	3	$9.12 \cdot 10^{-3}$	$3.04 \cdot 10^{-3}$	0.356	$2.22 \cdot 10^{-2}$
Parallel	7	$6.49 \cdot 10^{-3}$	$9.28 \cdot 10^{-4}$	0.302	$1.88 \cdot 10^{-2}$
Tseng-Lee	15	$5.29 \cdot 10^{-3}$	$3.53 \cdot 10^{-4}$	0.243	$1.52 \cdot 10^{-2}$
"optimal"	3	$1.02 \cdot 10^{-2}$	$6.38 \cdot 10^{-4}$	0.310	$1.94 \cdot 10^{-2}$
Parallel	7	$7.10 \cdot 10^{-3}$	$4.44 \cdot 10^{-4}$	0.278	$1.73 \cdot 10^{-2}$
Tseng-Lee	15	$6.41 \cdot 10^{-3}$	$4.01 \cdot 10^{-4}$	0.258	$1.61 \cdot 10^{-2}$

Table 6: The speedup and efficiency for the parallel Gale-Shapley and Tseng-Lee algorithms

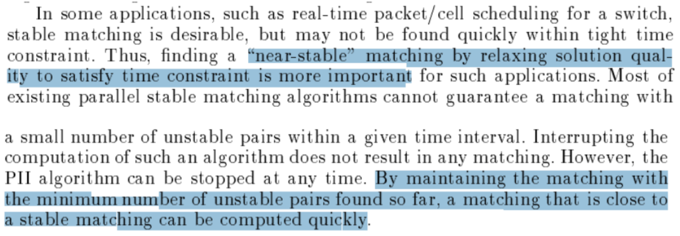
The finding of their experiments highlights that the parallel approach doesn't works better compared to sequential because of the increased overhead due to communication. [Lar97]



6 Conclusion

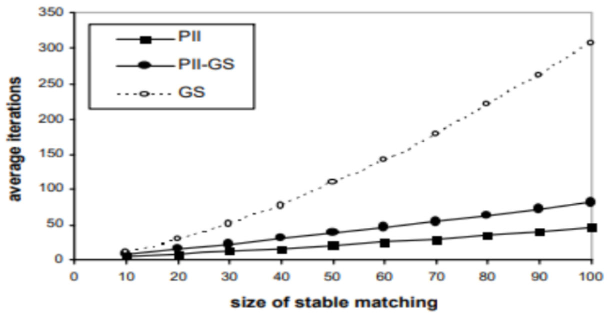
Our experiments clearly show that the parallel algorithms are slower than the sequential ones, and they get even slower the more processors we use. As stated earlier, the communication is too expensive compared to the amount of work to be done. Clearly the stable marriage problem is to simple to be solved with commercial parallel computers.

- A parallel iterative improvement of Stable Matching Algortihm (Enyue Lu and S. Q. Zheng Department of Computer Science, University of Texas at Dallas, 2003)[LZ03]



In some applications, such as real-time packet/cell scheduling for a switch, stable matching is desirable, but may not be found quickly within tight time constraint. Thus, finding a "near-stable" matching by relaxing solution quality to satisfy time constraint is more important for such applications. Most of existing parallel stable matching algorithms cannot guarantee a matching with

a small number of unstable pairs within a given time interval. Interrupting the computation of such an algorithm does not result in any matching. However, the PII algorithm can be stopped at any time. By maintaining the matching with the minimum number of unstable pairs found so far, a matching that is close to a stable matching can be computed quickly.



References

[GS62] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[Lar97] Jesper Larsen. *A parallel approach to the stable marriage problem*. Datalogisk Institut, Københavns Universitet, 1997.

[LZ03] Enyue Lu and SQ Zheng. A parallel iterative improvement stable matching algorithm. In *International Conference on High-Performance Computing*, pages 55–65. Springer, 2003.

[Qui87] Michael J Quinn. *Designing efficient algorithms for parallel computers*. McGraw-Hill, Inc., 1987.

[TST99] Chung-Piaw Teo, Jay Sethuraman, and Wee-Peng Tan. Gale-shapley stable marriage problem revisited: strategic issues and applications. In *Integer Programming and Combinatorial Optimization: 7th International IPCO Conference Graz, Austria, June 9–11, 1999 Proceedings 7*, pages 429–438. Springer, 1999.