CS6023: Matching with Preferences

J. Prem Krishnaa*, R. Meenakshi†
Indian Institute of Technology Madras
Roll No: *CS14B049, †AE15B051

***Abstract*—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. The Stable Matching Problem

The stable matching problem has been very well studied in literature since its inception in the seminal paper by Gale and Shapley [1] in 1962. The stable matching problem has led to lots of variants and researchers keep finding more and more pristine structural results on it. On the practical application side, we are interested in the problem as it can model many real world processes like college applications, student elective allocation etc. The problem is also studied from many different domains, for eg., researchers from economics study it from a varied perspective, with their interest arising from the recent popularity of matching markets. In this section we formally define the stable matching problem.

A stable matching instance can be modelled as a bipartite graph $G(\mathcal{M} \cup \mathcal{W}, E)$, where $\mathcal{M}$ represents the set of men and $\mathcal{W}$ represents the set of women. Every $x \in \mathcal{M} \cup \mathcal{W}$ has a strict preference order over a subset of elements in the other set. An edge (m, w) is said to be acceptable, if w is present in m's preference list and vice-versa. The set of acceptable edges form the edge set E. Let $\mathcal{W}_m$ denote the set of women in m's preference list and $\mathcal{M}_w$ denote the set of men in w's preference list. We denote the strict preference order for m by $\succ_m$ on $\mathcal{W}_m$, and similarly for w by $\succ_w$ on $\mathcal{M}_w$.

Any arbitrary subset M of E is called a **matching**. For a given matching M, we denote the woman matched to m by $M(m)$, and the man matched to w by $M(w)$.

Definition 1: An edge $(m, w) \in E \setminus M$ is called a **blocking** pair if both the following conditions are satisfied: (i) m is unmatched or $w \succ_m M(m)$, and (ii) w is unmatched or $m \succ_w M(w)$. A matching is said to be **stable** if it does not admit any blocking pair.

For any stable matching instance, the Gale and Shapley algorithm, as described (Algorithm 1), always computes a stable matching in $O(|E|)$ linear time. A lot of algorithms for different variants of stable matchings draw inspiration from Gale and Shapley algorithm. So if we are able to parallelize Gale and Shapley algorithm on GPU, it is very likely that we could be able to parallelize other similar algorithms with appropriate modifications. To make our task simpler, we make the following assumptions: the graph is a complete bipartite graph, i.e., all men have all women in their preference list and vice versa.

Example 1.1: $m_1 : w_1, w_2$ $\quad$ $w_1 : m_1, m_2$
$m_2 : w_2, w_1$ $\quad$ $w_2 : m_2, m_1$

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 $w$ is unmatched then
6:             $M = M \cup \{(m, w)\}$
7:         else
8:             Let $m'$ be the current partner of $w$
9:             if $m' \succ_w m$ 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 the above example, we can verify that $M_1 = \{(m_1, w_1), (m_2, w_2)\}$ is a stable matching by our definition, whereas $M_2 = \{(m_1, w_2), (m_2, w_1)\}$ is not a stable matching.

Parallelization: A very interesting observation about Algorithm 1 is that in step-4, the order in which men make proposals is not specified. But irrespective of this order, the algorithm outputs the same final matching [2]. This property can be exploited for parallelization. So we can have men make proposals in parallel to their most preferred women, but a sequential bottleneck could occur, if all men apply to the same woman, in which case we need to process each man one by one using queue like structure. In terms of GPU implementation, we can consider it synonymous to an atomic operation.

II. Serial Code Bench-Marking

We had a public C++ repository (GraphMatching - Amit Rawat) containing stable matching serial code available to us. However we felt it would be harder to port the existing code with CUDA, we implemented our own serial code using C language. We generated stable matching instances using three models as follows. These generators were described in Amit's python implementation repository (HR). Notation: k is the fixed number of women in each man's preference list, in complete bipartite graph, $k = n$.

- **Master:** In this model we try to reflect a real world scenario that few women are more preferred among the men. Also the women have a master list to rank men. A geometric probabilistic distribution with $p = 0.10$ is set up among the women - this denotes the probability with which a woman is chosen to be included in a man's preference list. Each man samples k women from the above distribution and orders them arbitrarily. Every woman orders the men in her preference list according to the master list of men.
- **Shuffle:** This model is similar to Master model except that we do not consider the master list of men. The men chose their preference list as described in Master model. But every woman orders the men in her preference list arbitrarily. This model also reflects a real-world scenario that each woman can have her own precedent for assigning preferences to men.
- **Random:** This model is used to generate arbitrary/unstructured instances by randomly sampling the preference lists for men. Similarly every woman orders the men in her preference list arbitrarily.

In our case, both random and shuffle models are similar as we assumed a stable matching setting with complete bipartite graph / preference lists. Each model we considered the following values of n (number of men, number of women): 10, 50, 100, 200, 500, 1000. For each configuration, we measured the computation time and averaged it over five instances. As the algorithm has a running time of $O(|E|)$ and $|E| = n^2$, we expect to see a quadratic plot for n vs computation time. The experimental time taken is plotted as follows.

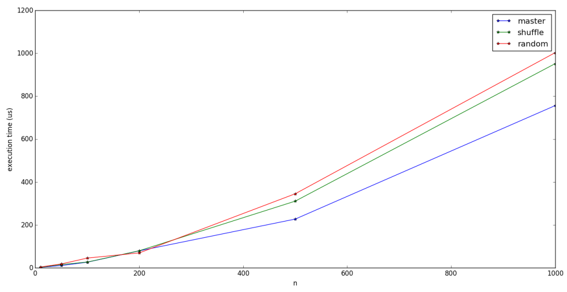


Fig. 1: Serial - Time taken vs n

The above time taken are obtained by running on a machine with the following configuration: 4 GB RAM, Ubuntu 16.04, gcc compiler, Intel i5 processor. It is practically difficult to obtain the time taken for larger values of n. For eg., the size of one input file for $n = 10^4$, would be 1 GB. For $n = 10^5$, it would be 100 GB, which is impossible to process normally. It might be possible to obtain the time taken for $n = 10^4$, by running on some cloud based service like AWS, since our current local configuration is incapable of processing that.

Though we do not have too many data points, its easy to observe that the slope keeps increasing, which is similar to that of a quadratic curve. We can also see that there is not much difference between the execution times of different models. Master model is relatively faster, which could be attributed to the fact that it utilizes a master list of residents.

III. Existing Work

The parallelizability of stable matching problem is not fully understood. Work done in the early 1980s ([3], [6]) suggested that parallel stable matching algorithms cannot be expected to provide high speedups on average. Hence designing such parallel algorithms that perform up-to mark for most cases is a challenging exercise. Early work on designing a parallel algorithm for computing a stable matching was done by Enyue Lu and S.Q. Zheng [4] where they propose a new approach called Parallel Iterative Improvement (PII). They also implemented their algorithm in OpenMP using n^2 processing elements (PEs), to obtain an $O(n \log n)$ running time for their PII algorithm. This was an improvement over the sequential Gale and Shapley algorithm, which is $O(n^2)$. However since they require $O(n^2)$ PEs, its not scalable in a GPU context since we have limited hardware resources. Later we present our approach to parallelize Gale and Shapley algorithm itself in a GPU, which only uses n PEs. In 2014, Paul Richmond [7] presented a parallel implementation of Gale and Shapley algorithm in FLAME GPU, which is an agent modelling simulation framework, and deals with a different objective. Hence it is not comparable with our task due to the difference in framework. Recently in 2016, Fredrik Manne et al. [5] presented a parallel OpenMP implementation of Gale and Shapley algorithm. They also address the challenges involved with implementing their approach using CUDA.

IV. CUDA Implementation Attempts

In this section, we present four different CUDA kernels that we attempted for parallelizing the Gale and Shapley algorithm. We compare the running time of each kernel with that of serial benchmark and also analyze the kernels using Nvidia profiling tools `nvprof, nvvp`.

A. *CUDA Kernel-1*

We implemented a basic CUDA kernel to compute a stable matching in a stable matching instance and verified its correctness by comparing with the output of serial implementation. We computed the time taken to run locally on a Nvidia GeForce 830M GPU card.

The idea behind parallel implementation was to have a separate thread for each man, which can apply to women independently of other men. 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. We also have a global counter which we update using `atomicAdd` function to identify the end of the algorithm. These could lead to performance degradation, as can be seen in the following plot - parallel implementation is almost 30 times slower than our serial implementation. We suspected that this is because of the latency due to global atomic operations.

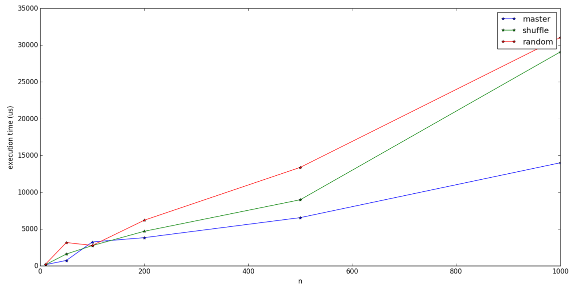
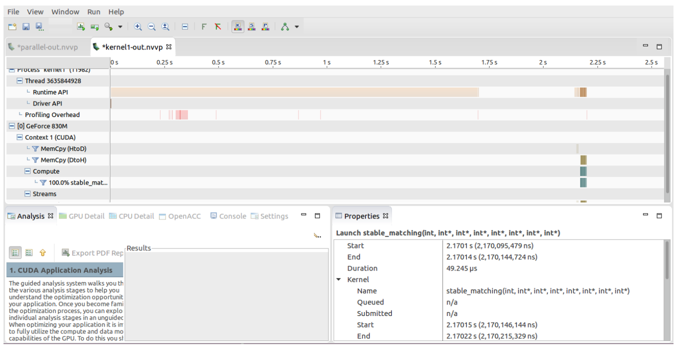


Fig. 2: Kernel-1 : Time taken vs n

We also used Nvidia's profiling and visualization tools, `nvprof`, `nvvp` to check the critical time taking components of the CUDA implementation.



Launch stable_matching(int, int*, int*, int*, int*, int*, int*, int*)

Start	2.1701 s (2,170,095,479 ns)
End	2.17014 s (2,170,144,724 ns)
Duration	49.245 µs
Kernel	
Name	stable_matching(int, int*, int*, int*, int*, int*, int*, int*)
Queued	n/a
Submitted	n/a
Start	2.17015 s (2,170,146,144 ns)
End	2.17022 s (2,170,215,329 ns)

Fig. 3: Kernel-1 Profiling

The above metrics were obtained for an instance with $n = 10^3$. The total compute time given by `cudaEvents` was 28 ms. However on checking the visual profiler, we found that the kernel was executing for only 10 ms. And the overhead due to `cudaMemcpy` from device to host in every iteration accounted for 1.5 ms. It turns out that the major part of the overhead is due to kernel launches (as can be seen in Figure 3).

B. *CUDA Kernel-2*

We suspected a bottleneck in Kernel-1 due to atomic add operations that we use to maintain the size of the matching, which could lead to inherent serialization. So we tried to remove this add operation and replace it with a boolean variable instead (non-atomic operation, to keep track of end of the algorithm). However this variable has to be set and checked before and after kernel call respectively, in every iteration, thus requiring two `cudaMemcpy` operations, as compared to just one in Kernel-1. We verified the correctness of this kernel by comparing its output with that of serial code. Though we expected this kernel to perform better as we removed the atomic add operation, but in reality this was two times slower than Kernel-1, implying that the `cudaMemcpy` operations are much costlier.

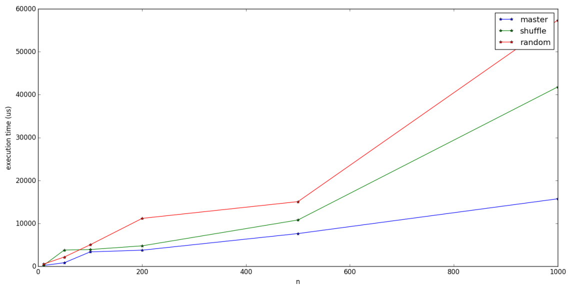
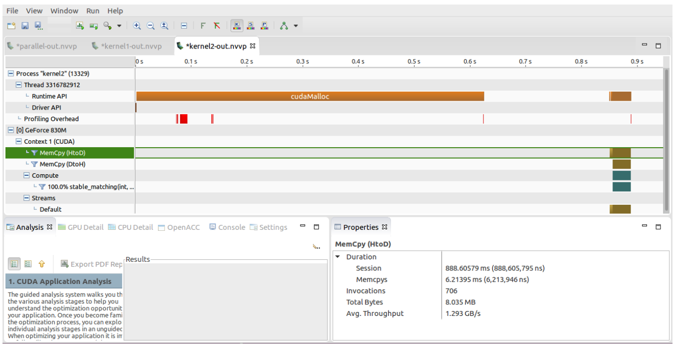


Fig. 4: Kernel-2 : Time taken vs n

We used profiling tools `nvprof`, `nvvp` to check different components of our CUDA implementation.



MemCpy (HtoD)

Duration	
Session	888.60579 ms (888,605,795 ns)
Memcpys	6.21395 ms (6,213,946 ns)
Invocations	706
Total Bytes	8.035 MB
Avg. Throughput	1.293 GB/s

Fig. 5: Kernel-2 Profiling

Turns out, atomic add overhead is negligible, and we are only adding more overheard due to extra `cudaMemcpy`. Further we still have the multiple kernel invocation overhead, carrying over from Kernel-1.

C. *CUDA Kernel-3*

We wanted to overcome the limitation of Kernel-2, by avoiding `cudaMemcpy` operations, which led us to think that if the main kernel call was done from the GPU in another kernel, then we need not have to do the `cudaMemcpy` operations since the global variable is available in both the kernels. Hence we decided to try out **dynamic parallelism** by calling a `__global__` function from another `__global__` driver function. The driver function is called from the main with just one block and one thread configuration. Here we expected the runtime to be better than that of both Kernel-1 and Kernel-2, but however the runtime was almost similar to that of Kernel-1. The reason we believe is because of `cudaDeviceSynchronize` function needed to be placed in the driver kernel in order to start the child kernel, which is leading to an additional overhead.

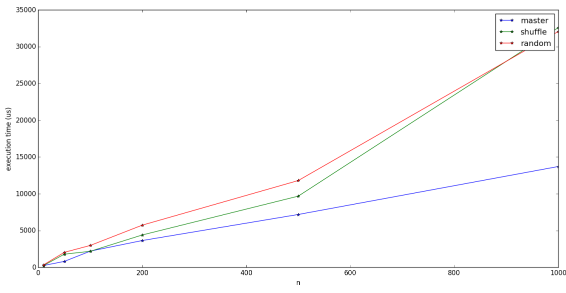
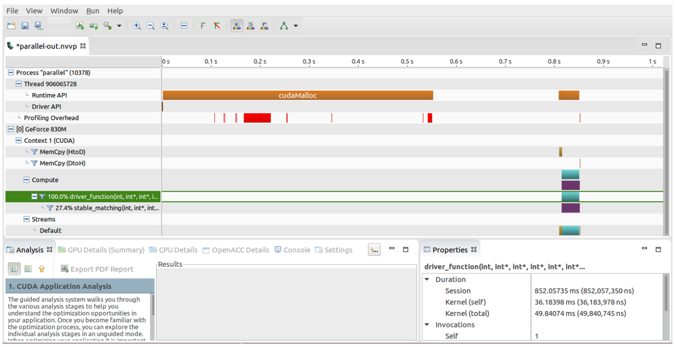


Fig. 6: Kernel-3 : Time taken vs n

We used profiling tools `nvprof`, `nvvp` to check different components of our CUDA implementation.



```
Process "parallel" (10378)
  Thread 906065728
    Runtime API
    Driver API
  Profiling Overhead
[0] GeForce 830M
  Context 1 (CUDA)
    MemCpy (HtoD)
    MemCpy (DtoH)
    Compute
      100.0% driver_function(int, int*, int*, i...
        27.4% stable_matching(int, int*, int...
    Streams
      Default
```

1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode.

driver_function(int, int*, int*, int*, int*, int*...

Duration	
Session	852.05735 ms (852,057,350 ns)
Kernel (self)	36.18398 ms (36,183,978 ns)
Kernel (total)	49.84074 ms (49,840,745 ns)
Invocations	
Self	1

Fig. 7: Kernel-3 Profiling

As we can observe from Figure 7, the actual kernel is taking up only 27% of the total time taken by the driver kernel. And the major part of the driver kernel is also taken up by `cudaEventSynchronize` command, as we predicted. This seems to be a drawback of using dynamic parallelism for our task.

D. CUDA Kernel-4

The major time-taking factor in all three previous kernels was the initialization overhead for multiple kernel calls. We tried to overcome this in Kernel-4 by having a single kernel call. We have a loop which runs inside the kernel and terminates once all men get matched. Note that we have used `__syncthreads()` inside the kernel to synchronize all threads. This works because we have a single block which contains all threads (our maximum value of $n = 10^3$). If we need to scale it up for larger n, then we would have to use multiple blocks since there can be at most 1024 threads in a block. Therefore we need a grid level synchronization for our method to work similarly in such case. This concept actually exists in CUDA as *cooperative-groups*. As we can see in Figure 8, this kernel is much faster than all the previous kernels. However, it is still 4-5 times slower than our serial implementation.

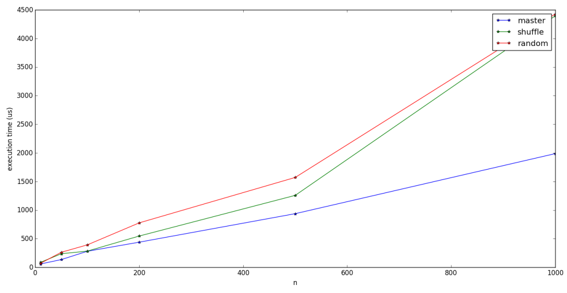
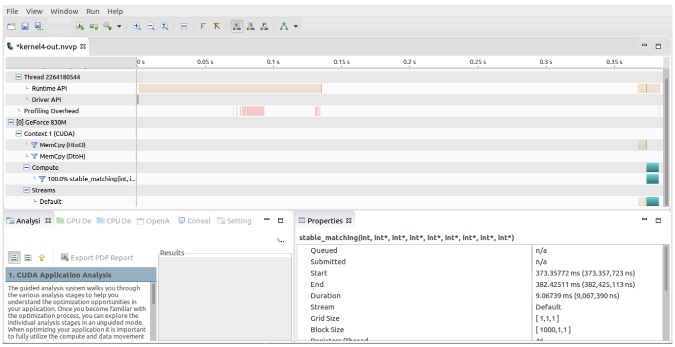


Fig. 8: Kernel-4 : Time taken vs n

We used profiling tools `nvprof`, `nvvp` to check different components of our CUDA implementation.



```
Thread 2264180544
  Runtime API
  Driver API
Profiling Overhead
[0] GeForce 830M
  Context 1 (CUDA)
    MemCpy (HtoD)
    MemCpy (DtoH)
    Compute
      100.0% stable_matching(int, i...
    Streams
      Default
```

1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement

stable_matching(int, int*, int*, int*, int*, int*, int*, int*, int*)

Queued	n/a
Submitted	n/a
Start	373.35772 ms (373,357,723 ns)
End	382.42511 ms (382,425,113 ns)
Duration	9.06739 ms (9,067,390 ns)
Stream	Default
Grid Size	[1,1,1]
Block Size	[1000,1,1]

Fig. 9: Kernel-4 Profiling

As we can observe from Figure 9, we have gotten rid of overheads due to multiple kernel calls. We can infer that the slow-down compared to serial implementation is now completely because of global memory `atomicCAS` operations.

V. Summary of Observations

Firstly we were limited by the maximum input size ($n = 10^3$) we could take due to compute limitation, as explained earlier. Our main focus in CUDA implementation was to speed up the compute time as compared to that of the serial implementation, hence we did not take the `cudaMemcpy` calls preceding the kernel call into account. Also as far as we observed, there is no scope for pipe-lining the `cudaMemcpy` and kernel computation operations, since the entire input data is required before starting any kernel. Therefore asynchronous memory copies using streams would be of no use here. For large values of n, we can possibly improve the compute-memcpy ratio and use the host-to-device bandwidth effectively by using asynchronous memory copies, but we finally need to synchronize all corresponding streams, since only then we can proceed with our kernel call.

There is no global memory coalescing possible either, since the preference lists of men and woman are randomly generated. Inherently there are also lot of unavoidable `if` conditions in the algorithm, leading to control divergence between threads of a warp. We thought of using shared memory on top of our Kernel-1, but however the overhead to initialize the shared memory for each launch might exceed the gain obtained over shared memory `atomicCAS` operation latency, also its not scalable for multiple blocks since privatization is not clearly evident.

In Kernel-3, we used dynamic parallelism, where we intuitively wanted to implement some sort of a master-slave model. But even there, we were unable to overcome the overhead due to `cudaDeviceSynchronize` calls, and ultimately leading to similar time taken as that of Kernel-1. Finally in Kernel-4, we use a single kernel call but still we are unable to get speedup, mainly due to `atomicCAS` overhead.

VI. Conclusion

We were theoretically correct about our parallelization intuition - the number of effective iterations required by parallel implementation is significantly lower than that of serial implementation. We have verified this and the comparison can be seen in the following Figure 10.

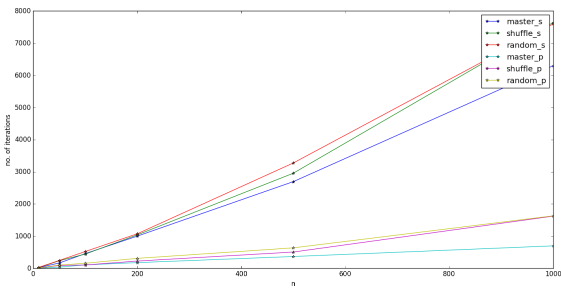


Fig. 10: No. of iterations vs n

We tried to parallelize the Gale and Shapley algorithm [1] in CUDA, and we did so using four different kernel implementations. However despite our best efforts, the time taken for the best parallel implementation turned out to be $\approx$ 4-5 times slower than that of our most efficient serial implementation. Even though we believed at the start that our parallelization intuition could lead to an average case speed-up, it turned out not to be the case, due to the inherent overheads associated with the CUDA implementation such as kernel launch, atomic operations and `cudaDeviceSynchronize` calls in case of dynamic parallelism.

Hence with our attempts and experiments as evidence, we conclude that the Gale and Shapley algorithm is difficult to parallelize for the input data under our consideration, due to the GPU overheads involved. However since our input size is not large, in future we can try to compare the time taken for $n = 10^4$ in serial and parallel implementation (Kernel-4 with cooperative-groups).

References

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

[2] Dan Gusfield and Robert W. Irving. *The Stable marriage problem - structure and algorithms*. MIT Press, 1989.

[3] Deepak Kapur and Mukkai S. Krishnamoorthy. Worst-case choice for the stable marriage problem. *Inf. Process. Lett.*, 21(1):27–30, 1985.

[4] Enyue Lu and S. Q. Zheng. A parallel iterative improvement stable matching algorithm. In *High Performance Computing - HiPC 2003, 10th International Conference, Hyderabad, India, December 17-20, 2003, Proceedings*, pages 55–65, 2003.

[5] Fredrik Manne, Md. Naim, Håkon Lerring, and Mahantesh Halappanavar. On stable marriages and greedy matchings. In *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing, CSC 2016, Albuquerque, New Mexico, USA, October 10-12, 2016.*, pages 92–101, 2016.

[6] Michael J. Quinn. A note on two parallel algorithms to solve the stable marriage problem. *BIT*, 25(3):473–476, 1985.

[7] Paul Richmond. Resolving conflicts between multiple competing agents in parallel simulations. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I*, pages 383–394, 2014.