FINAL REPORT

SUMMARY

We implemented parallel versions of the Gale-Shapley stable matching algorithm using OpenMP, OpenMPI, and CUDA. Each approach targets a different parallel architecture: shared memory across threads, message passing between processors, and GPU parallelism. We analyzed the scalability, performance bottlenecks, and synchronization trade-offs in parallelizing a sequential proposal-based algorithm. Our results reveal that while CUDA provides strong speedup for moderate problem sizes, MPI-based distributed memory parallelization delivers the best scalability for large data sets. Shared address space parallelization improves performance for small problems but severely suffers from synchronization and memory bottlenecks as size increases.

BACKGROUND

The Gale-Shapley algorithm solves the Stable Matching Problem, which involves finding a stable one-to-one matching between two equally sized groups (*n* men and *n* women). Each participant ranks all members of the opposite group in order of preference. A match is considered stable if no man and woman prefer each other over their assigned partner. In our implementation, we randomly generate these preference lists as input to the algorithm, which computes a stable matching by assigning each participant a partner. We then validate the stability of the resulting matching by checking all possible pairings.

Initially, all participants are unmatched. In each iteration of the algorithm, all unmatched men propose to the highest-ranked woman on their preference list to whom they have not yet proposed. During the response stage, each woman reviews her current match alongside any new proposals and tentatively accepts the suitor she prefers most, rejecting all others. This process repeats until no unmatched men remain. Because men initiate the proposals, the resulting matching is man-optimal, meaning each man is paired with the best possible partner he could achieve in any stable matching.

Our initial implementation uses a Participant structure for each individual, stored in a list of size *2n*. The first *n* Participants are men, and the rest are women. Each Participant has a field for its ID, its current match, and its preference list of size *n*. The current match field is initialized to -1 to indicate that the participant has no match. This is the only value that is changed while the algorithm is running. Another key data structure for male participants is a propose_next list of size *n* initialized to 0, which tracks the index of the next woman to whom the man will propose. For women, we maintain a proposal list, which stores a list of proposals each woman receives in the round. A proposal is considered received once the man's index is in the woman's proposal list. Finally, we track a dynamic free_males list, which contains the index of all unmatched men who are eligible to propose in the next round. If a woman rejects a man, his ID is added back to the free_males list for the next iteration.

The core computational bottleneck lies within each iteration, during the proposal and response stages. In each round, we must iterate through all unmatched men, update their proposal targets, and modify the corresponding woman's proposal queue. Next, each woman must examine her received proposals, determine the best offer, and add rejected men back into the free_males list. As the input size grows, the cost of proposal evaluation and memory accesses increases significantly, creating performance challenges. Although men and women interact with proposals and responses, a lot of the computation can be done

with little communication or locking overhead. Moreover, as *n* grows large, the bottleneck is often not pure computation but memory accesses. Multiple threads can potentially help hide memory latency by accessing different memory regions in parallel.

Within a single iteration, each man's or woman's work is independent of others in their group, providing an opportunity for data-parallelism. We can safely parallelize the proposal and response phases across all participants. However, contention may arise when multiple men propose to the same woman, leading to sequential access to the woman's proposal list. Similarly, updates to the shared free_males list must be managed serially or with atomic operations to avoid any lost entries. These synchronization points introduce overhead and potential bottlenecks.

Further performance considerations include locality and SIMD suitability. Preference list lookups and proposal list updates can lead to scattered memory accesses, reducing spatial locality. Optimizing data layout or access patterns could improve cache performance. Additionally, deploying the algorithm on SIMD architectures, such as on a GPU, can result in divergent control flow and contention in shared data structures, which may limit scalability. Overall, while the stable matching problem exhibits potential for data-parallelization, the inter-thread dependencies, synchronization challenges, and memory access patterns make writing an efficient parallel implementation a non-trivial problem.

DATA GENERATION

In the early stages of our project, we explored a Python script found during our research that generated preference lists for participants in the Gale-Shapley algorithm. The script provided flexibility, allowing us to specify partial preference lists of customizable lengths and even create "popular" women (women who received more proposals). However, this data generation method occasionally produced incorrect data, such as repeated values within a participant's preference list, which caused the algorithm to fail to converge to a stable matching solution. While this script was useful for basic tests, we chose not to invest additional time into fixing it, as the main focus of our project was on the parallelizability of the Gale-Shapley algorithm. Instead, we moved forward with generating cleaner, random data that better suited our goal of testing our parallel implementation. In the future, we hope to explore incorporating partial preference lists and introducing the concept of "popular" women, which could increase the complexity and variability of our testing scenarios.

To ensure reproducibility, we transitioned to generating standard data files to test our algorithm on. While this method proved to be reliable, it limited our ability to test on very large datasets due to disk space constraints. To address this, we switched to in-place data generation and validation. In this approach, data is generated dynamically during the testing phase, immediately before running the algorithm, and validated immediately after to ensure correctness. For reproducibility, we introduced a seed parameter that allows us to set a specific seed value for random number generation at the time of testing, ensuring consistent results across different numbers of threads.

SHARED ADDRESS SPACE

In this version of the stable matching algorithm, we implement shared address space parallelization to run the Gale-Shapley algorithm using OpenMP. This function takes in the participants array, the number of participants per group, and the number of threads to use as inputs. Within the participants array, men and

women are distinguished by assuming men's IDs range from [0, n) and women's range from [n, 2•n). Each participant's current_partner_id field is initially set to -1 to indicate that they are unmatched.

We begin by placing all male IDs into a free_males list, since all men start unmatched. We also initialize a propose_next vector of size n, which tracks the index of the next woman each man will propose to based on his preference list. The algorithm then enters a main while loop that repeatedly performs proposal and response stages until a stable matching is found – that is, until free_males becomes empty. In each iteration, we create a proposals vector of size n, where each entry corresponds to a woman and stores the list of proposals she receives during that round.

During the proposal stage, each free man proposes to the next woman on his preference list. Because each man's proposal is independent from others, this stage is parallelized using a #pragma omp parallel for. However, because multiple men may propose to the same woman simultaneously, updates to each woman's proposal list must be protected to avoid race conditions. Once all proposals have been made, we proceed to the response stage, where each woman independently selects her most preferred proposer from her list. If she prefers her new proposer over her current match, she accepts the new proposal and rejects the previous match. All rejected proposers are also added back into a new_free_men list for future iterations. This stage is also parallelized, with updates protected to prevent simultaneous writes to the shared new_free_men list. Once both stages are complete, the free_males list is updated, and the next iteration begins.

We explored several synchronization strategies to manage updates to shared resources. A poorly chosen approach could significantly degrade performance and limit scalability due to contention for resources, serialization, and inefficient parallel execution.

Our first approach was to use localized updates through thread-local buffers. We theorized that if we could reduce contention on shared data structures, we would improve parallel performance. To achieve this, we modified the algorithm so that each thread writes to its private copy of the proposals buffer during the proposal stage. This eliminates the need for synchronization during the parallel for loop, as each thread only accesses its own memory. Afterward, all thread-local buffers are sequentially merged into a single global proposals vector. In theory, this approach results in zero contention during the main parallel phases.

Our next approach was to use #pragma omp critical. This method effectively serializes access to critical sections by allowing only one thread to execute a protected block of code at a time. Because this is a general, compiler-supported method that abstracts away manual lock management, we hoped that the OpenMP runtime could optimize critical sections internally, potentially using lightweight locking mechanisms, local spinning, or contention management strategies to reduce overhead.

Finally, we experimented with using pthread mutexes to manually control locking at a finer granularity. Unlike OpenMP critical sections, pthread mutexes allow explicit lock and unlock operations, giving us full control over where and when locks are acquired and released. We wanted to see if manually minimizing the duration of locked regions could outperform the generalized locking behavior provided by OpenMP. Additionally, pthread mutexes can help manage lock contention and offer lower lock acquisition overhead. By trying pthreads, we hoped to achieve even finer-grained synchronization and potentially reduce waiting time compared to compiler-managed locking.

Synchronization Comparison (n = 10,000)					
seed = 10		1 Thread (s)	2 Threads (s)	4 Threads (s)	8 Threads (s)
local updates	Merge Array Time	0.096699	0.192050	0.383680	0.766539
	Computation Time	1.417344	1.808596	2.852463	5.175186
#pragma omp critical	Lock Waiting Time	0.004427	0.010094	0.026858	0.093435
	Computation Time	0.548644	0.391189	0.330062	0.320926
pthread mutexes	Lock Waiting Time	0.006152	0.009826	0.013919	0.018438
	Computation Time	0.607567	0.429886	0.341980	0.32364

The table above summarizes the performance results for each synchronization strategy across different thread counts, measured for n = 10,000 participants with random seed = 10. For each strategy, we report both the time spent in synchronization-related overhead (either merge time or lock waiting time) and the overall computation time.

At n = 10,000, it is immediately clear that the local updates approach with thread-local buffers performs poorly as the number of threads increases. The overall computation time grows dramatically, reaching over five seconds with eight threads. The timers around the sequential merge step clearly show that the additional overhead of merging thread-local proposals significantly outweighs any benefits of eliminating contention during the parallel phase. Since merging effectively doubles the work per iteration, the scalability of this approach is severely limited. Furthermore, the additional memory overhead introduced by allocating separate buffers for each thread likely exacerbates cache pressure, leading to further performance degradation.

In contrast, both #pragma omp critical and pthread mutexes scale more effectively compared to purely serial code. The computation time consistently decreases as the number of threads increases, with only a small penalty from lock waiting time, even as the thread count rises. Interestingly, when n = 10,000, the pthread version spends slightly less time waiting on locks compared to #pragma omp critical, but its overall computation time remains higher. This suggests that although pthreads reduce explicit lock contention, they incur additional hidden overheads. In particular, manual lock acquisition and release introduce function call and synchronization costs even when locks are uncontended, while stronger memory fences and cache coherence traffic create pipeline stalls that are not reflected in lock waiting measurements. Since the critical sections in our code are relatively short and accessed frequently, the simplicity of OpenMP's built-in synchronization ultimately outperforms the manually managed pthread synchronization.

Synchronization Comparison (n = 40,000)					
seed = 10		1 Thread (s)	2 Threads (s)	4 Threads (s)	8 Threads (s)
#pragma omp critical	Lock Waiting Time	0.025964	0.049118	0.135762	0.407523
	Computation Time	16.042568	11.801310	10.214647	10.192633
pthread mutexes	Lock Waiting Time	0.047206	0.060537	0.067088	0.097360
	Computation Time	18.157880	12.539671	10.623584	10.336868

When the problem size increases to n = 40,000, the trends become even clearer. #pragma omp critical continues to outperform pthread mutexes at all thread counts. Although both strategies benefit from increased parallelism, the relative scalability of #pragma omp critical remains stronger. Both strategies also exhibit diminishing returns beyond four threads. This behavior likely results from increased contention for shared data structures, reduced granularity of parallel work as the matching stabilizes, and growing synchronization overhead. Additionally, as the algorithm stabilizes and there are fewer free men, there is less work to do each iteration. Amdahl's Law predicts such a plateau in parallel speedup when the sequential portions of an algorithm, such as synchronization and locking, begin to dominate total runtime as problem size and thread count grow.

Overall, our results demonstrate that among the synchronization strategies explored, #pragma omp critical offers the best tradeoff between synchronization overhead and scalability. It is simple to implement, benefits from compiler and runtime optimizations, and remains portable across architectures. While thread-local updates theoretically eliminate contention, the merge overhead introduces a substantial sequential bottleneck. Pthread mutexes offer finer control but suffer from additional setup and management costs and excessive locking overhead in our fine-grained critical sections.

To further improve the performance of our shared address space implementation, we examined additional performance counters, particularly focusing on cache behavior. We measured cache misses and cache references using hardware counters, and the results revealed a surprisingly high cache miss rate. As shown in the table below, when n = 10,000, nearly 20% of all cache references result in cache misses. Although the miss rate improves slightly to approximately 13% at n = 40,000, the miss rates remain significant and indicate poor spatial and temporal locality in our memory accesses, resulting in worse computation times due to data stalls.

Cache Performance Counters With Normal Preference Lists					
		1 Thread	2 Threads	4 Threads	8 Threads
n = 10,000 (seed 10)	Cache Misses	118,114,195	119,308,999	121,398,733	120,386,322
	Cache References	565,893,420	573,803,041	543,366,623	539,028,542
	Miss / Total Refs	20.872 %	20.793 %	22.342 %	22.334 %
n = 40,000 (seed 10)	Cache Misses	2,699,615,087	2,720,682,859	2,699,169,955	2,687,192,230
	Cache References	20,723,661,094	20,726,132,414	21,328,131,930	19,693,631,184
	Miss / Total Refs	13.027 %	13.127 %	12.655 %	13.645 %

We realized that one of the main contributors to poor cache performance was the way women's preference lists were structured and accessed during proposals. In our implementation, each woman stored a preference list consisting of the IDs of men, ordered from most to least preferred. To determine a woman's ranking of a specific proposer, we had to loop through her entire preference list until the proposer's ID was found. This process resulted in frequent scanning of long arrays, poor cache usage, and repeated memory accesses, especially inside critical sections where decisions were being made.

To address this inefficiency, we redesigned how women's preferences were stored. Instead of keeping a list ordered by preference, we stored the rankings directly indexed by man ID. In this new format, each woman's preference array is organized such that index 0 holds her ranking of man 0, index 1 holds her ranking of man 1, and so on. This change allows us to directly access a woman's ranking of any proposer in constant time with a single indexed read, eliminating the need for loops.

Cache Performance Counters With Inverse Ranking					
		1 Thread	2 Threads	4 Threads	8 Threads
n = 10,000 (seed 10)	Cache Misses	62,064,890	63,352,987	63,925,039	63,771,529
	Cache References	449,226,439	460,769,454	443,057,451	429,934,275
	Miss / Total Refs	13.816 %	13.749 %	14.428 %	14.833 %
n = 40,000 (seed 10)	Cache Misses	1,356,937,327	1,365,632,804	1,392,096,225	1,430,066,743
	Cache References	18,162,225,234	18,437,303,815	17,437,842,915	16,912,823,514
	Miss / Total Refs	7.471 %	7.407 %	7.983 %	8.456 %

This redesign dramatically improves cache behavior. Instead of scanning across arrays, each thread now performs a single indexed memory access to retrieve a ranking within its preference list, improving spatial locality and reducing cache misses. Although the search loop through the preference list was outside the

critical region, it still introduced significant latency before threads reached the synchronization point. Because threads had to scan long preference lists to determine rankings, each thread's time spent on work varied unpredictably, leading to increased load imbalance and causing threads to arrive at critical sections at unpredictable times, potentially creating burst contention. By restructuring the preference lists to allow constant-time indexed lookups, we eliminated search overhead, resulting in faster, more regular access to critical sections and overall reduced synchronization delays. As can be seen in the table below, our computation times also significantly improved.

Inverse Ranking of Women Comparison					
n = 40,000 (seed 10)		1 Thread (s)	2 Threads (s)	4 Threads (s)	8 Threads (s)
Before	Computation Time	16.107	12.328	10.228	10.019
After	Computation Time	13.384	10.404	9.232	9.136

We also tested the impact of flattening the participants array, hoping to improve memory access patterns. Rather than storing each participant as a struct with nested fields, we reorganized the data into a flattened 1D array of size $2n \bullet (n+1)$. Each participant occupied $n+1$ consecutive indices, with the first index storing their current match and the next n indices storing their preference list. The participant's ID no longer needed to be explicitly stored, as it could be inferred from their position in the flattened array.

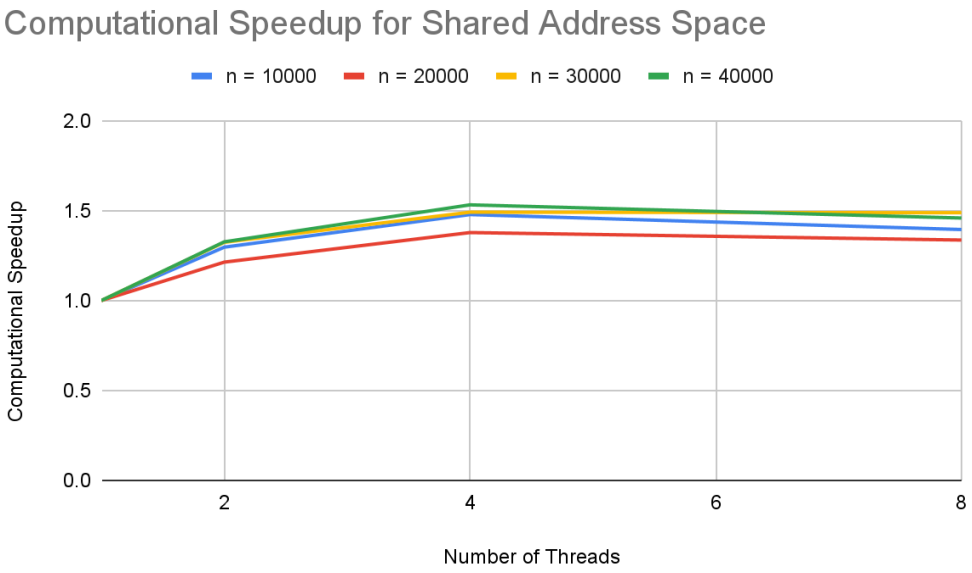
Performance Counters With Flattened Data				
n = 10,000 (seed 10)	1 Thread	2 Threads	4 Threads	8 Threads
Cache Misses	162,444,306	160,711,463	162,998,230	163,118,660
Cache References	1,141,723,380	1,183,680,814	1,192,830,029	960,829,770
Miss / Total Refs	14.228 %	13.577 %	13.665 %	16.977 %
Computation Time (s)	1.165	0.886	0.764	0.703

This restructuring aimed to improve spatial locality, ensuring that all data for each participant was stored contiguously in memory. We hoped to reduce cache misses by improving prefetching behavior and avoiding pointer dereferencing overheads associated with struct-based memory layouts. However, in practice, the total number of cache references and cache misses increased compared to the original layout, and the percentage of cache references that resulted in cache misses remained the same. Moreover, the overall computation time increased across all thread counts.

We attribute the increased computation time to several factors. Although the flattened layout improved spatial locality within a single participant's data, accessing different participants still required larger memory jumps, especially when proposals were made to random women in different parts of the array. Additionally, increased total memory accesses, combined with the larger contiguous array size, likely contributed to more pressure on cache hierarchies, offsetting any intended gains in locality, so we decided to revert to our previous implementation.

Finally, we theorized that as the stable matching algorithm progresses and the number of unmatched participants decreases, the overhead of parallelization might outweigh its benefits. We computed the time each 1,000th iteration takes, and found exponentially decreasing values. Later iterations compute faster because there is less work to do (fewer participants remain unmatched, so the number of proposals per round decreases), leading to load imbalance, leading to poor core utilization, and inefficient parallelism. To test this, we modified our parallel for loops to conditionally execute in serial if the number of free males fell below a user-defined threshold. However, in practice, we observed that the best threshold was effectively zero, meaning that parallel execution was always preferred. This could be because OpenMP incurs thread setup costs at the beginning of execution, and continuing parallel execution, even with small workloads, maintains better efficiency than switching to serial execution late in the algorithm.

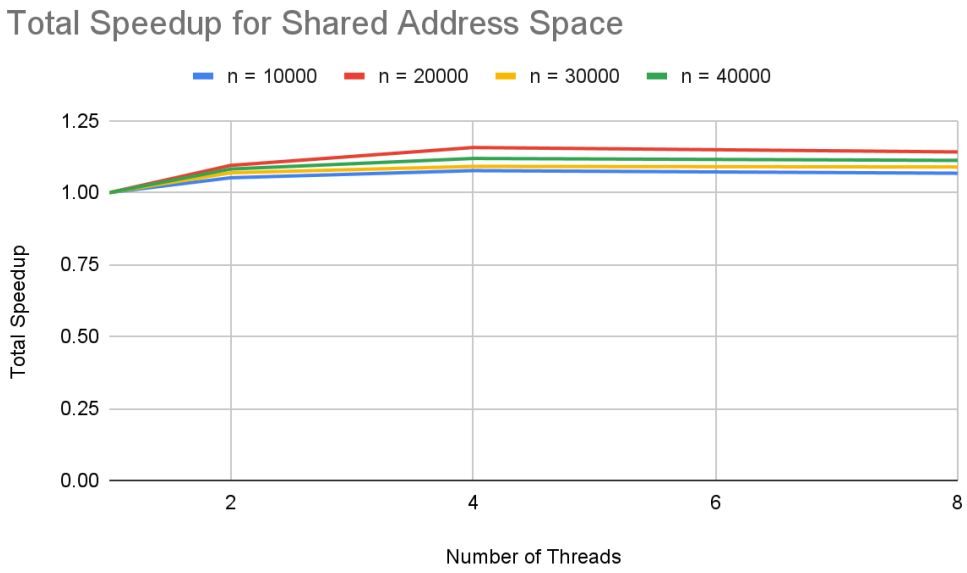
We evaluated our shared address space implementations across different synchronization strategies by measuring computational speedup and total speedup, calculated as the ratio of the computation time of one thread to the computation time for t threads. We ran experiments for different input sizes n=10,000, 20,000, 30,000, and 40,000, and varied the number of threads used (1, 2, 4, and 8).



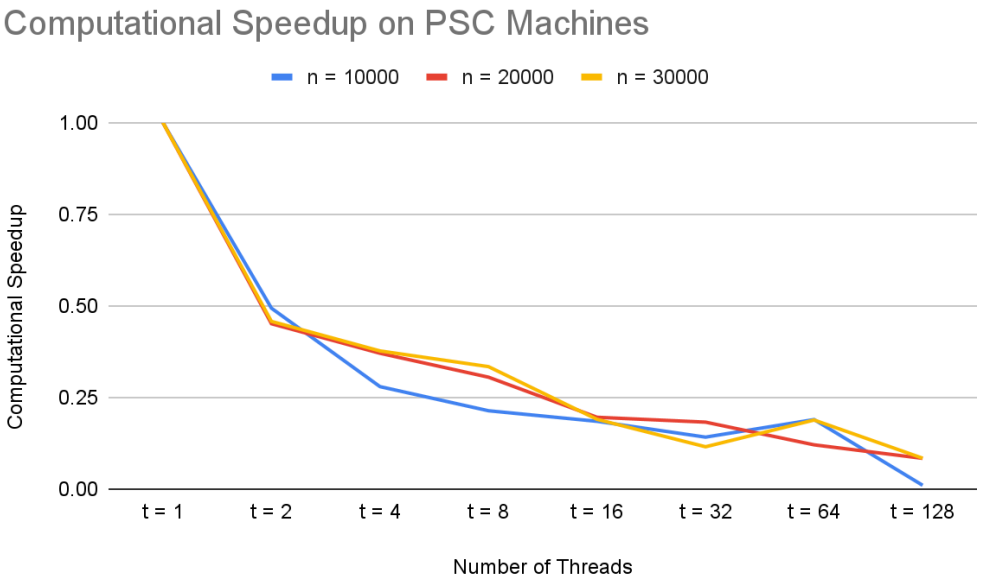
The graph above shows computational speedup. We observed that computational speedup improved steadily from 1 to 2 to 4 threads for all problem sizes. Speedup peaked around 4 threads and slightly declined at 8 threads. Larger workloads benefit slightly more from parallelism, as there is more work available to parallelize. For instance, with n =∫`¸ 40,00, we achieved a computational speedup of approximately 1.55x at 4 threads before seeing a small drop at 8 threads. This indicates that while the algorithm parallelized well initially, load imbalance and synchronization overhead diminish returns beyond 4 threads.

This trend can be directly related to the structure of the shared-memory code: although the main computational workload, such as proposal generation and preference list comparisons, is parallelized across threads, these threads must frequently access and update shared data structures (proposals and free_men lists). Because these updates are fine-grained and happen frequently, they likely cause cache line invalidations and increased memory bus traffic, leading to false sharing and memory bottlenecks. As

the number of threads increases, the overhead from coordinating these concurrent accesses grows faster than the computational benefits, ultimately causing the observed drop in computational speedup at higher thread counts. Therefore, even though the algorithm itself is parallel, the shared memory model and the code's frequent shared accesses create a natural scaling limit.



This graph shows total speedup (including all overhead and generating data). Here, the improvement was noticeably smaller. This demonstrates that sequential portions, such as the participant data creation phase, dominated the total runtime at higher thread counts, regardless of the size of *n*.



On the PSC machine, we observed poor computational scalability in our shared address space implementation. As shown in the graph, computation time actually increased as the number of threads grew, leading to progressively worse performance. Rather than achieving speedup, the system experienced slowdowns beyond a single thread. We believe this is due to significant synchronization

overhead, cache coherence costs, and memory bandwidth contention as more threads compete for shared data structures. Although the PSC machines offer higher raw parallelism (up to 128 threads), the Gale-Shapley algorithm's fine-grained synchronization needs and frequent critical section entry exacerbate contention across cores, especially when cache lines must constantly be invalidated and updated. In contrast, on GHC machines, which support a smaller number of threads (up to 8), the same algorithm achieves some meaningful speedup. This is because with fewer threads, synchronization and memory contention costs are relatively lower, critical sections remain short enough to be efficiently managed, and cache coherence traffic is less overwhelming. The workload per thread is also naturally larger, leading to better thread utilization and reducing the impact of idle time caused by load imbalance as matching progresses. Additionally, the Gale-Shapley algorithm exhibits load imbalance as matching stabilizes, causing some threads to become idle while others continue to work, further reducing efficiency. These effects compounded at higher thread counts, explaining the sharp deterioration in performance. Although parallelism improved computation time slightly at very small thread counts, scaling to large numbers of threads on the PSC machine ultimately led to diminishing returns and slower execution overall. This highlights the challenges of parallelizing fine-grained, synchronization-heavy algorithms in a shared memory environment.

While we do not achieve ideal linear speedup, likely due to memory hierarchy effects, synchronization overhead, and contention for shared resources such as cache lines and memory bandwidth, our shared address space parallelization still demonstrates solid scalability. Despite these bottlenecks, the observed speedup indicates that the stable matching problem contains meaningful parallelism that can be effectively exploited on modern multicore systems, especially as problem size and thread counts grow.

MESSAGE PASSING INTERFACE

Next, we implemented a distributed parallel version of the Gale-Shapley stable matching algorithm using MPI in C++. Our implementation targeted shared-memory multicore systems, running on standard CPU machines with OpenMPI installed. We used MPI to simulate distributed execution across multiple CPU cores on a single machine. Our goal was to leverage distributed data-parallelism to efficiently handle large numbers of participants and achieve good scalability as the problem size increased.

In our MPI implementation, the root processor (rank 0) generates all participant preference lists and initially stores participants as structs. Before distributing the data, the participants are flattened into a one-dimensional integer array to facilitate easier message passing. Each participant occupies n+1 consecutive integers: one for their current match (initialized to -1) and *n* for their ordered preference list. We use MPI_Bcast to send the entire participants array to all processes, followed by an MPI_Barrier to ensure synchronization. Strictly speaking, the barrier is not required for correctness because MPI_Bcast is already a collective operation. We still include it to synchronize the start of computation timing across all processes for more consistent performance measurements.

After the broadcast, each processor extracts its assigned chunk of participants based on its rank and the total number of processes. We assign contiguous chunks rather than interleaving participants to promote better spatial locality and simpler indexing. Each processor then builds its local men and women arrays from its assigned data. Additionally, each processor maintains a local free_males vector to track which of

its men are unmatched and a propose_next vector to track which woman each man will propose to next. Each participant's local index is mapped back to its global ID so that message passing during proposals and responses remains consistent across all processes.

At each iteration, similar to the shared address algorithm, the algorithm proceeds through a proposal stage, a proposal exchange, a proposal evaluation and response generation stage, a response exchange, and a local state update. During the proposal stage, each processor processes its free men and prepares proposals. Each proposal specifies the woman's global ID and the proposer's global ID. Proposals are grouped by the responsible processor that owns the targeted woman and packed into a send buffer. Proposals are exchanged among processors using MPI_Alltoallv, ensuring each processor receives only the proposals relevant to the women it owns. Each processor then processes the received proposals, choosing the most preferred suitor for each woman based on her preference list. Responses are generated for all proposers, indicating acceptance or rejection. These responses are again flattened and exchanged among processors using another MPI_Alltoallv. Upon receiving responses, each processor updates the states of its local men: accepted men are removed from the free list, while rejected men remain unmatched and continue proposing.

Before the end of the iteration, each processor sets a local_active flag if it still has free men who have not exhausted their preference lists. An MPI_Allreduce operation with logical OR (MPI_LOR) combines all local flags into a global_active flag. If global_active equals zero, the algorithm has converged, and the matching process terminates. After convergence, each processor builds its local list of matched (man_id, woman_id) pairs. Non-root processors send their matchings to the root processor using MPI_Send, and the root processor combines the results using MPI_Recv to produce the complete matching for validation.

During early versions of the implementation, we encountered a major scalability issue. Using MPI_Bcast meant that every processor received the entire participants array, which led to excessive memory usage and std::bad_alloc errors at larger n. To resolve this, we replaced MPI_Bcast with MPI_Scatterv, which distributes only the relevant chunks of the participant data to each processor. Each processor now receives only its assigned participants, reducing memory usage and improving scalability. Furthermore, rather than copying the data from the broadcasted array into local structures, processors now directly store the received scattered data into local memory, saving computation time and memory bandwidth.

After Switching to Scatterv				
n = 10,000	1 Thread	2 Threads	4 Threads	8 Threads
Initialization time	2.821336463	2.692275643	2.741566166	2.985921928
Computation time	0.661064091	0.4629830445	0.3719883885	0.397223852

To further improve memory access patterns, we introduced a reverse ranking optimization for women's preference lists. As described in our optimizations for our shared address space implementation, instead of storing ordered lists of preferred men, each woman's preference list was converted into a direct ranking lookup indexed by man ID. This change allowed ranking comparisons to be performed with a single indexed read rather than scanning through preference lists, improving memory access patterns during

proposal evaluation as threads would access contiguous memory regions. While the level of indirection in accessing rankings introduces some difficulty in prefetching, eliminating scanning the preference list with single indexed reads significantly reduces total memory work, leading to better overall memory efficiency and shorter critical path lengths during proposal evaluation.

Inverse Ranking of Women's Preference Lists				
n = 10,000	1 Thread	2 Threads	4 Threads	8 Threads
Initialization time	2.199358409	2.107044043	2.082914766	2.119075284
Computation time	0.557540586	0.331083733	0.2361809453	0.2052358054

Although the MPI_Alltoallv efficiently exchanges proposal and response messages among processors, communication costs became a larger portion of total execution time as the number of participants grew. In early iterations, when many free men remained, communication patterns were dense and balanced, leading to good scalability. However, as the number of unmatched participants shrank, the volume of communication decreased, while the fixed cost of MPI synchronization and message setup remained constant. As a result, communication overhead became more significant in later iterations. These effects illustrate a common tradeoff in distributed algorithms between fine-grained parallelism and communication cost.

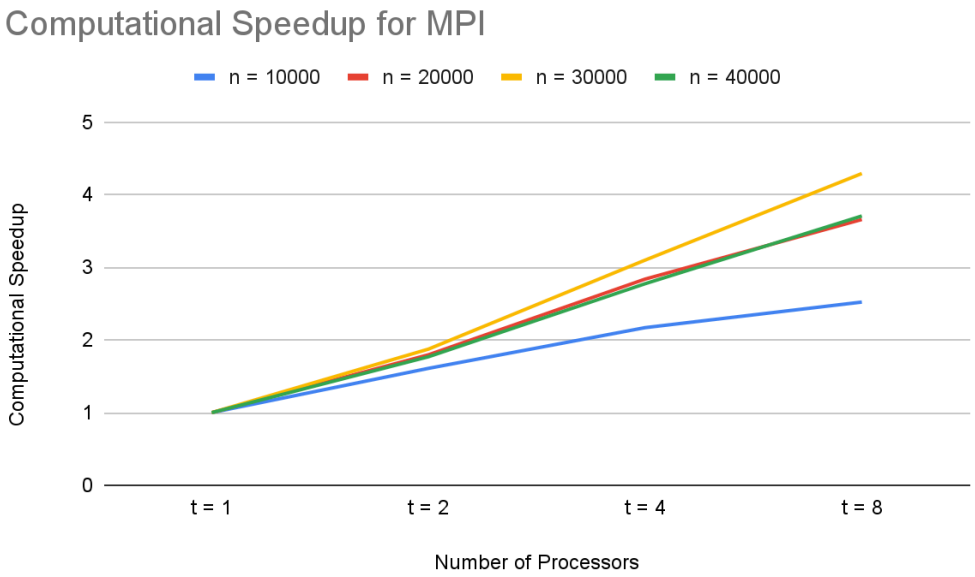
Despite the rising communication overhead near convergence, the ability of MPI to distribute both data storage and computation across processors remained critical. By avoiding centralized memory bottlenecks, MPI enabled us to scale to much larger problem sizes than would have been possible on a single processor, ensuring that parallelization remained worthwhile even as communication costs grew. To maximize these benefits, we eliminated the bottleneck of having a single root processor generate and distribute all participant data. Instead, each processor independently generated the preference lists for its assigned chunk of participants. This decentralized data generation removed the need for MPI_Bcast or MPI_Scatterv during initialization, avoiding bandwidth limitations and memory overhead associated with transmitting large participant arrays across processors.

Previously, the largest problem size we could compute was n = 30,000, limited by the computational burden and memory requirements placed on processor 0. With decentralized generation, we scaled to n = 45,000, achieving larger problem sizes without exhausting memory or overwhelming network communication. Decentralized generation also reduced memory bus contention during initialization, as each processor independently generated and wrote its assigned chunk of participant data, spreading memory traffic across cores and minimizing congestion compared to a single processor generating the entire dataset. Each processor locally constructed valid randomized preference lists for its men and women, maintaining consistency with the overall structure of the problem and enabling better scalability for large instances.

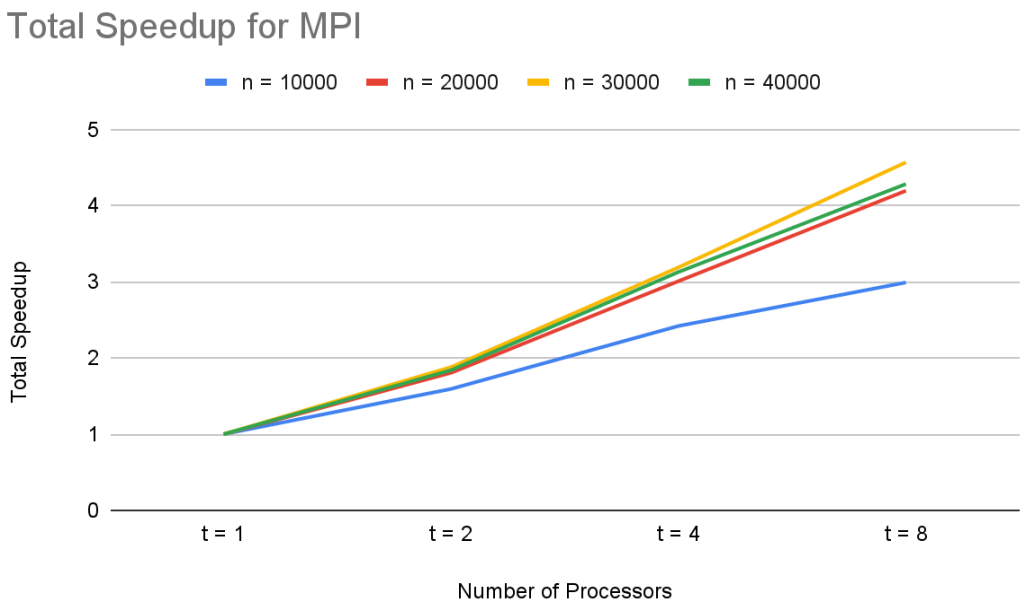
Experimental results demonstrated the impact of this optimization. For n = 45,000, both initialization time and computation time scaled nicely with increased processor counts and decentralized data generation.

Decentralized Data Generation				
n = 45,000 (seed 10)	1 Thread	2 Threads	4 Threads	8 Threads
Initialization time (s)	114.405	100.707	55.436	44.940
Computation time (s)	88.733	63.436	37.781	26.707

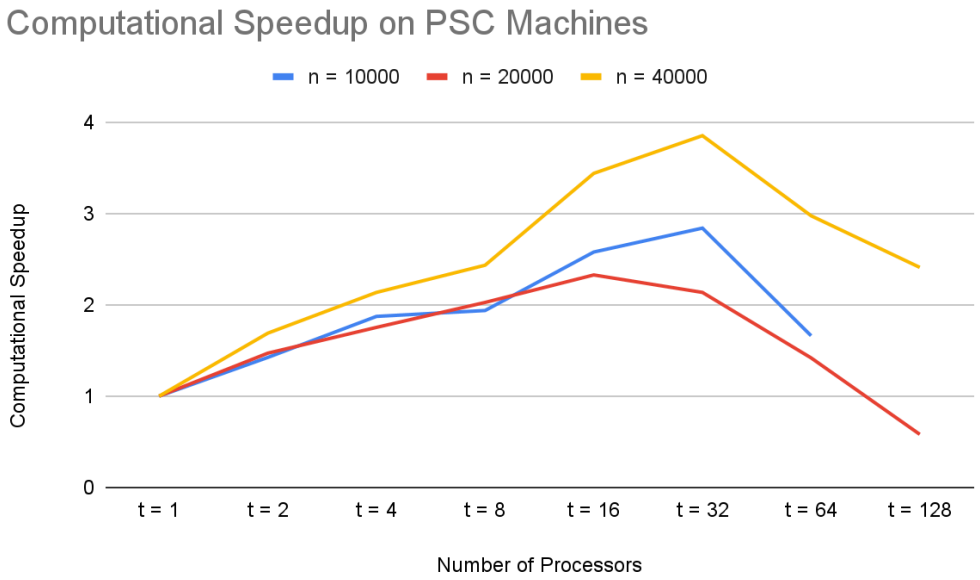
By decentralizing data generation, we eliminated the memory bottleneck that previously constrained our problem sizes. Earlier algorithm versions were memory-constrained and could not handle larger instances such as $n = 45{,}000$ due to centralized storage limitations. With decentralized generation, the limiting factor shifted: we became time-constrained rather than memory-constrained. Although total computation time naturally increases with problem size, this transition reflects a major improvement in parallelism, as we expanded the scale of the problems we could solve, even if direct timing comparisons to earlier versions are no longer meaningful because those versions could not feasibly run at larger n at all.



We evaluated our MPI-based distributed implementation of Gale-Shapley by measuring computation time across different numbers of processes (t = 1, 2, 4, and 8) and problem sizes ($n = 10{,}000$ to $40{,}000$). As shown in the graph above, we observed consistent speedup as the number of processes increased. Larger problem sizes achieved better scaling. For $n = 30{,}000$ and $n = 40{,}000$, we achieved speedups of approximately 4× on 8 processors, while smaller problem sizes ($n = 10{,}000$) experienced more modest gains. This behavior is expected, as communication and synchronization overheads in MPI become relatively less significant when the computational workload per process is larger. For small n, communication cost forms a larger fraction of the total time, limiting achievable speedup. Nonetheless, our distributed MPI implementation successfully handled very large problem sizes and scaled effectively across available cores, demonstrating the benefits of distributing both data and computation in memory-constrained environments.



Interestingly, in our MPI implementation, total speedup (including initialization time) closely mirrored computation speedup. This behavior is due to our decentralized data generation strategy. Each processor independently generates the preference lists for its local participants, eliminating any centralized initialization bottleneck. Therefore, as the number of processes increased, initialization time actually decreased slightly, contributing to a clean total speedup curve. This contrasts with the shared address space versions, where initialization remained sequential and became a relatively larger bottleneck as the problem scaled.



On the PSC machine, our MPI implementation achieved moderate computation speedup for smaller numbers of processes but exhibited diminishing returns at higher core counts. Speedup improved steadily

from 1 to 16 processes, but after reaching a peak at 32 cores, performance began to degrade. This trend reflects typical scaling challenges in distributed systems: as the number of processes grows, the communication overhead (especially from MPI_Alltoallv exchanges) increasingly dominates computation time. Additionally, at high process counts, load imbalance becomes more severe, since many men are matched early while others remain unmatched, causing processors to sit idle. Network contention and MPI synchronization costs further worsened scalability beyond 32 processes. Thus, while MPI parallelization enabled us to scale to larger problem sizes, ideal linear scaling was not achieved due to communication costs, increasing idle time, and synchronization barriers at high core counts.

CUDA

Our CUDA-based implementation of the Gale-Shapley stable matching algorithm aims to leverage the massive thread parallelism available on GPUs while addressing challenges such as synchronization, memory contention, and warp divergence. The algorithm was implemented in C++ with CUDA, targeting NVIDIA GPUs on a shared-memory system.

In our first design, each CUDA thread was mapped to a single man. Threads ran concurrently within a main loop until a globally stable matching was achieved. Each thread maintained a private propose_next variable, initialized to zero, that tracked which woman to propose to next. Inputs to the algorithm included flattened arrays for men's and women's preference lists, match arrays for men and women, a per-woman lock array, and stability flags. Flattened arrays improved memory locality and simplified copying data from the host to the device. Additionally, based on results from previous parallel implementations, we started with an inverse ranking approach for women’s preferences, allowing constant-time lookups of a man's rank rather than scanning preference lists.

To manage concurrent proposals safely, we implemented fine-grained locking where each woman has her own lock, acquired using an atomic compare-and-swap (CAS) operation and released using an atomic exchange operation. This preserved parallelism by allowing multiple proposals simultaneously, avoiding coarse-grained serialization of the entire women’s array. Fine-grained locks did introduce occasional contention when multiple men proposed to the same woman, but because preference lists were randomly generated, such cases were rare.

The algorithm proceeds through iterations within a while loop. At each iteration, threads were synchronized using __syncthreads() to ensure all previous proposal updates were complete, as iterations are dependent on previous ones. Free men attempted proposals by acquiring a woman's lock and checking if she was unmatched or preferred the new suitor over her current match. Match updates were made accordingly, and threads incremented their propose_next counters. A free thread must continue running in case its match will reject it for a better proposal in the future, so all threads must exit together. After the proposal phase, synchronization ensured that stability flags were correctly updated. If no proposals or rejections occurred, indicating a stable matching, threads exited together.

This approach replaces the round-based logic of the serial version with fine-grained proposal processing. Unlike in the serial algorithm, where women batch-process proposals at the end of each round, in the CUDA version, men propose and update matches immediately after acquiring locks. No threads are assigned to women; all threads represent men, minimizing thread role divergence and maintaining load

balance. However, this requires careful locking around shared memory structures since any thread may modify a woman's match at any time.

Initially, we restricted all threads to a single thread block, allowing for simple synchronization between threads in each iteration via __syncthreads(). However, this approach could not scale beyond $n = 1024$ due to CUDA's maximum threads-per-block limit and left many multiprocessors idle.

To address this, we explored alternative multi-block synchronization strategies. One approach used a global convergence flag combined with __threadfence() and atomic operations to attempt to synchronize across blocks. However, because CUDA lacks native global barriers, thread blocks drifted apart in execution and exited prematurely, leading to unstable or invalid matchings.

As a more robust alternative, we transitioned to a kernel-per-iteration approach. The CPU wrapped the main proposal loop and launched a new kernel for each proposal round. Within each kernel, threads would complete one iteration by proposing to one woman, setting a global "changed" flag if any match was updated. If no changes occurred after a kernel launch, the loop terminated, indicating stability. This method resolved synchronization issues between blocks because each kernel launch acted as an implicit global barrier. However, this design introduced new challenges, as kernel launch overhead accumulated over many iterations, particularly at larger n. Additionally, warp divergence worsened, as threads within the same warp exited at different times as they completed their matching process, leading to inefficient SIMT utilization.

Ultimately, we decided to revert to a single-block kernel approach using 1024 threads per block. In this design, each CUDA thread is assigned a subset of men for whom it makes proposals. We adopted an interleaved assignment strategy, where men are distributed round-robin among threads, rather than assigning contiguous blocks of men. This interleaving promotes memory coalescing, as threads within a warp access adjacent memory addresses during men's data lookups and updates, improving memory bandwidth efficiency and reducing scatter-gather overhead. Interleaving also improved SIMD efficiency by aligning thread memory accesses and minimizing stalls from uncoalesced memory operations.

Additionally, conditional branching was minimized to reduce warp divergence, allowing threads to progress uniformly. We also optimized stability checking. Instead of updating a global memory variable, is_stable was kept as a register-local variable within the block. After all threads have finished proposing and synchronized, the is_stable variable determines whether threads exit together or continue to the next iteration. This helped avoid expensive global memory accesses and atomic operations on the stability flag, further improving iteration performance.

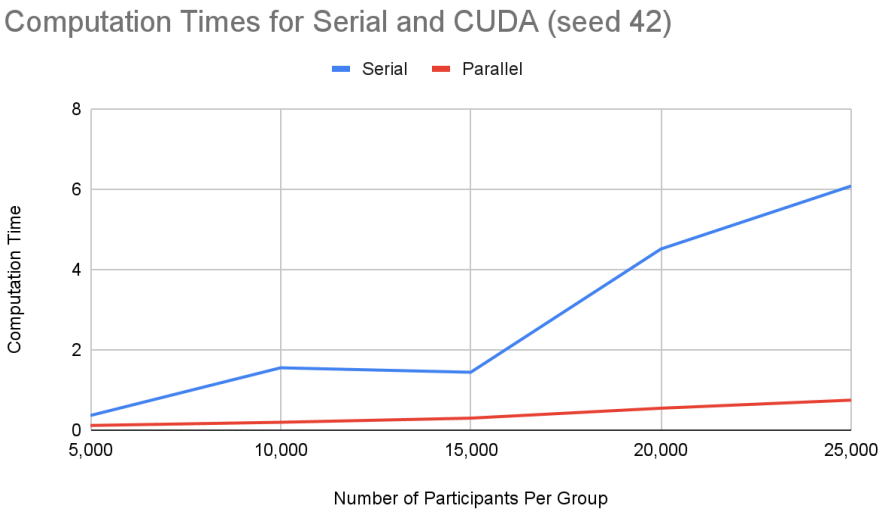
Comparison of Computation Time Across Implementations (seed = 10)					
Computation Time (s)	n = 5,000	n = 10,000	n = 15,000	n = 20,000	n = 25,000
Serial	0.447	0.658	2.561	9.39	6.617
Kernel Per Iteration	0.207	0.375	0.707	1.039	1.408
Single-Block Kernel	0.123	0.175	0.307	0.674	0.731

We evaluated the performance of our CUDA-based Gale-Shapley implementation by measuring wall-clock computation time across a range of problem sizes (n = 5,000 to 25,000). The baseline for speedup calculation was our serial CPU implementation. Each experimental run involved randomly generated participant preference lists with randomized seeds to ensure robustness across input instances.

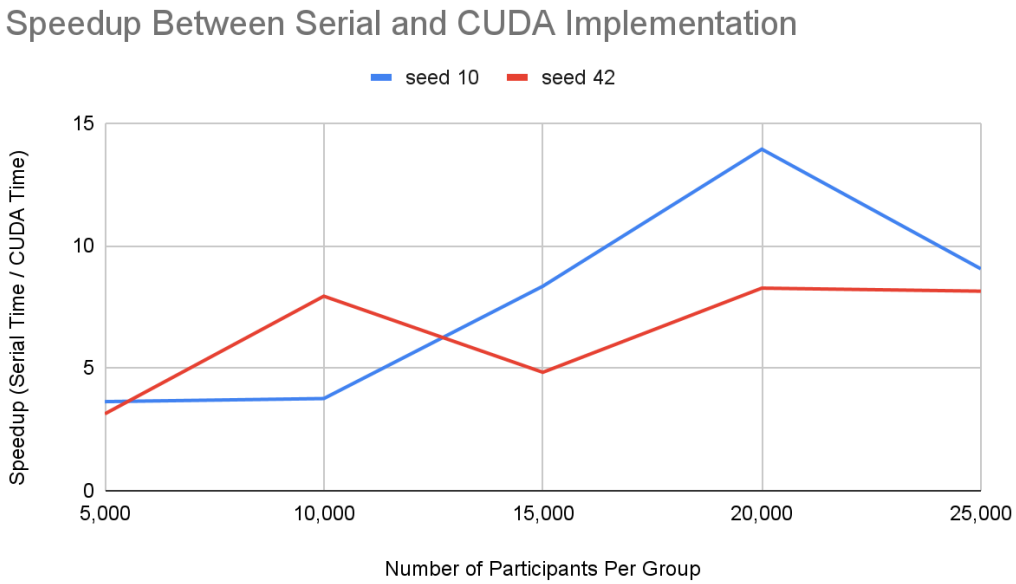
As shown in the table above, the single-block kernel approach consistently outperformed both the serial baseline and the kernel-per-iteration strategy across all problem sizes. As n increased, the number of iterations required to reach stability also grew, amplifying the overhead associated with frequent kernel launches in the kernel-per-iteration design. In contrast, the single-block kernel incurred no per-iteration launch overhead and maintained better thread synchronization, scaling more effectively. The interleaved thread assignment reduced memory scatter-gather costs, improved cache locality, and minimized warp divergence. These factors combined to make the single-block kernel the most scalable and efficient design. While hardware limits on block size prevent scaling indefinitely, the single-block approach achieved the best performance for the problem sizes tested. Further scaling would require multi-block synchronization strategies to fully utilize GPU multiprocessors.

Despite these improvements, several bottlenecks persisted. While SIMD efficiency was strong during early balanced rounds, it degraded in later stages as more men became matched, resulting in growing load imbalance and idle warps. Profiling at n = 5,000 revealed that although branch efficiency was very high (99% branch efficiency), synchronization stalls at barriers accounted for approximately 59% of issued instruction cycles, highlighting how imbalance increased as the matching stabilized. Furthermore, although flattened arrays and inverse rankings improved memory locality, hardware prefetching remained largely ineffective due to the random access patterns inherent to proposing to arbitrary women.

We also profiled performance at a larger problem size (n = 15,000) to evaluate scalability. Branch efficiency remained high (99.34%), and the achieved occupancy stayed at 100%. Scheduler utilization improved slightly (~30% SM busy rate compared to ~24% at n = 5,000), and synchronization stall cycles dropped to 38%, confirming that early iterations with more unmatched participants led to better-balanced workloads. However, the L1 cache hit rate decreased from 96% to 72% at larger n, reflecting increased randomness and access scattering as matching progressed. Despite these challenges, the single-block kernel design scaled effectively to larger problem sizes, with bottlenecks stemming primarily from the irregular and data-dependent nature of the matching problem rather than GPU resource underutilization.



The graph above shows the absolute computation time for both the serial CPU implementation and the CUDA single-block kernel across different problem sizes. The CUDA implementation consistently achieved much lower computation times, with the gap widening as n increased. Notably, while the serial time grows steeply with problem size, the parallel time grows much more gradually, highlighting the scalability advantages of the CUDA approach. Even at $n = 25{,}000$, the GPU computation time remained under one second, compared to over 6 seconds for the CPU.



As shown in the speedup graph, our CUDA implementation achieved increasing speedup relative to the serial baseline as problem size increased. For $n = 20{,}000$, we observed a peak speedup of approximately 14x for seed 10, and 8x for seed 42. Although variability exists between seeds, the overall trend demonstrates decent scaling. Larger problem sizes provide more work to amortize fixed overheads like kernel launch and synchronization, improving parallel efficiency. Slight differences in speedup between seeds can be attributed to randomness in participant preferences, which affects the number of proposal iterations needed to reach stability.

Overall, key takeaways from our CUDA optimization efforts include the critical importance of minimizing warp divergence, maximizing memory coalescing through interleaved access patterns, and carefully balancing synchronization overhead with scalability. Future improvements for larger-scale problems would likely require exploring hierarchical kernel designs and better multi-block coordination. Nonetheless, the CUDA implementation achieved a dramatic reduction in runtime, meeting our goals of significantly accelerating the stable matching computation compared to serial CPU execution.

DISCUSSION

Across the three parallelization strategies we explored, shared address space, CUDA, and MPI, our experiments revealed clear differences in scalability, efficiency, and suitability for the Gale-Shapley stable matching problem.

The shared address space implementation uses multithreading on a single machine and provides very small speedups over the serial baseline. While early scaling from 1 to 2 or 4 threads fared well, further increasing the number of threads began to degrade performance. This is most likely due to synchronization overhead, as threads need to synchronize frequently at iteration barriers, limiting parallel efficiency. Furthermore, memory contention among threads trying to access and update shared data structures led to poor cache behavior and cache line bouncing. Overall, shared memory parallelism was not able to fully utilize available hardware and is fundamentally constrained when scaling to very large problem sizes because it relies on centralized memory. As a result, while simple and conceptually straightforward, shared address space parallelization was the least effective of the three methods.

CUDA, in contrast, achieved significantly better performance by leveraging the massive thread-level parallelism available on GPUs. Mapping each man to a thread and using fine-grained locking allowed thousands of proposals to proceed concurrently, taking advantage of fast on-device memory access. Interleaved data assignment and minimizing warp divergence helped maximize SIMD efficiency. However, CUDA's performance still faced limitations. As matching progressed and more men became matched, load imbalance arose, with many threads idling while waiting for a few unmatched men to complete proposals. Furthermore, the random memory access patterns caused by arbitrary proposal orders limited hardware prefetching and occasionally led to cache contention. While CUDA showed strong speedups for moderate-sized datasets (n = 5,000 to 25,000), its scaling potential is ultimately limited by the GPU's maximum thread and memory capacity, making it harder to scale to extremely large datasets without complex multi-block coordination.

Among the three strategies, MPI delivered the best scalability and overall performance. By distributing both computation and data generation across processors, MPI parallelism avoided centralized bottlenecks and memory contention. Each processor worked independently on its subset of participants, only communicating necessary proposals and responses during iterations. This decentralized design enabled solving much larger problem instances (n = 40,000+) than shared address space or CUDA could handle. However, MPI parallelism also came with a lot of communication overhead, particularly in late stages with fewer free men. Nevertheless, these communication costs were outweighed by the benefits of distributed memory and parallel computation for large n.

In summary, the MPI-based distributed algorithm was the most scalable and best suited for large-scale instances of the stable matching problem. CUDA offered strong speedup for medium-sized problems where a GPU's resources could be effectively utilized, while shared address space parallelism was limited by synchronization and memory hierarchy effects and was only useful for relatively small-scale problems. The best choice for a parallelization strategy depends heavily on the intended problem size. For small instances, shared memory may suffice. For mid-sized problems, CUDA offers decent speedups. But for truly large instances, MPI is best.

Through our exploration of parallelization strategies, we learned that the nature of the stable matching problem, with its irregular, dynamic matching progress and memory access patterns, poses significant challenges for parallel computing. Fine-grained synchronization, memory contention, and load imbalance are key obstacles that limit ideal speedup across all approaches. Despite these challenges, parallelism does enable some meaningful acceleration.

REFERENCES

White, C., & Lu, E. (2013, Summer). *An improved parallel iterative algorithm for stable matching [Poster]*. Salisbury University, NSF REU Research Experience. https://sc13.supercomputing.org/sites/default/files/PostersArchive/tech_posters/post302s2-file2.pdf.

Jones, Bryan. (2025). *Gale-Shapley*. GitHub. https://github.com/bryancjones/gale-shapley.

DISTRIBUTION OF WORK

We divided the project work equally between both group members. Both of us collaborated on designing and implementing the different versions of the Gale-Shapley stable matching algorithm. The initial implementations for the message passing (MPI) and CUDA versions were written by Rhea, while the data generation, shared address space version, and the validation script were implemented by Lucy. We then worked together to refine data structures, synchronization strategies, and optimizations such as inverse ranking and decentralized data generation.

Both of us contributed to running scalability tests and collecting performance data. Rhea primarily collected data from the GHC machines and conducted CUDA profiling and cache miss analysis. Lucy collected additional performance data from both the PSC and GHC machines. We jointly analyzed our results and collaboratively wrote and edited the full final report, dividing sections based on our areas of expertise but reviewing and refining each other's work throughout the process.

Given equal contribution throughout all phases of the project, we believe that the credit should be distributed 50% - 50% between the two of us.