2025 34th International Conference on Parallel Architectures and Compilation Techniques (PACT)

A Stable Marriage Requires a Shared Residence with Low Contention and Mutual Complementarity

Jiaxin Liu
The Ohio State University
Columbus, OH, USA
liu.11080@osu.edu

Rubao Lee
Freelance
Columbus, OH, USA
lee.rubao@ieee.org

Cathy H. Xia
The Ohio State University
Columbus, OH, USA
xia.52@osu.edu

Xiaodong Zhang
The Ohio State University
Columbus, OH, USA
zhang@cse.ohio-state.edu

***Abstract*—The Stable Marriage Problem (SMP) is a combinatorial optimization problem aimed at creating stable pairings between two groups, traditionally referred to as men and women. SMP has been widely applied across various domains, including healthcare, education, and cloud computing, to optimize resource allocation, matching, and utilization. The classical approach to solving SMP is based on the Gale-Shapley algorithm, which constructs stable pairings sequentially. However, this algorithm is both time-consuming and data-intensive, resulting in significant slowdowns even for a moderate number of participants. Despite various attempts to parallelize the Gale-Shapley algorithm over the years, progress has been consistently impeded by three major bottlenecks: (1) frequent and expensive data movement, (2) high synchronization overhead, and (3) workload-dependent and irregular data accessing and parallel processing patterns.**

To resolve the above-mentioned bottlenecks, in this paper, we introduce Bamboo-SMP, a highly efficient parallel SMP algorithm, and its implementation in a hybrid environment of GPUs and CPUs. We have made three key development efforts to achieve high performance for Bamboo-SMP. First, Bamboo-SMP effectively exploits the data accessing locality with a lightweight data structure to maximize the ”shared residence space”. Second, Bamboo-SMP employs an advanced hardware atomic operation to decrease execution latency with ”low contention”. Third, Bamboo-SMP is implemented in a hybrid environment of GPU and CPU, leveraging the high bandwidth of the GPU for massive parallel operations and the low latency of CPU for fast sequential tasks. By fostering ”mutual complementarity” between CPU and GPU, Bamboo-SMP attains superior performance, consistently exceeding the best existing methods by 6.69x to 21.44x across a wide range of workloads. Moreover, Bamboo-SMP demonstrates excellent scalability, efficiently solving large-scale SMP instances while achieving sustained speedups of 5.6× to 13.8× on 4 GPUs. To the best of our knowledge, Bamboo-SMP is the fastest and most scalable solution for SMP data processing.

***Index Terms*—Stable Marriage Problem, Massively Parallel Algorithm, Multi-GPUs, Scalability, Hybrid Computing Environment**

I. Introduction

The Stable Marriage Problem (SMP), introduced by David Gale and Lloyd Shapley in 1962, seeks to find a stable matching between two equally sized sets of participants, each with ranked preferences. These ranked preferences involve each participant creating a list that orders all members of the opposite set from most to least preferred. A matching is a bijection that pairs each participant in one set with a unique partner in the opposite set. A matching is stable when no pair of individuals would prefer each other over their assigned partners. The Gale-Shapley (GS) algorithm, also known as the Deferred Acceptance (DA) algorithm, guarantees to find a stable matching for any instance of the SMP. The GS algorithm operates as follows: each man proposes to his most preferred woman, each woman then considers all her proposals and tentatively accepts the one she prefers most, rejecting the others. A rejected man then proposes to his next preferred woman, and this process repeats until all men and women are matched [1].

The SMP has long been a cornerstone of both theoretical research and practical combinatorial optimization. Its versatility has not only spurred the development of diverse variants and new insights in recent years [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13] but also enabled widespread applications in real-world scenarios [14], [15], [16], [17], [18], [19], [20], [21], [17], [22], [23]. In healthcare, SMP optimizes resource distribution by matching organ donors to patients, patients to cancer treatment centers, and the elderly to healthcare facilities [24], [25], [26]. The educational sector benefits from SMP by assigning students to schools and dormitory rooms, thereby improving student satisfaction and meeting institutional needs [27], [28], [29], [30]. SMP also plays a crucial role in the labor market, facilitating mutually beneficial employment relationships [31], [32], [33], [34], [35], [36], [37]. In modern technology, SMP is applied for cloud resource allocation, task offloading for computer networks and IoT devices, and switch scheduling, leading to enhanced network performance and resource utilization [38], [39], [40], [41], [42], [43], [44], [14], [45]. The profound impact of SMP across diverse fields was recognized to Dr. Alvin Roth and Dr. Lloyd Shapley by the Nobel Prize in Economics in 2012.

A fundamental bottleneck in computing the SMP arises from the quadratic time and space complexity of the GS algorithm, rendering it inherently data-intensive. The scale of real-world SMP applications is immense: medical residency matching involves over 47,000 students [46], content delivery networks handle hundreds of thousands of requests [39], and some data-intensive matching systems must handle instances on the order of millions [47]. As the number of participants grows, the quadratic scaling quickly overwhelms centralized computational resources with the demands of processing preference lists and other data structures. However, the increased input size also exposes substantial data parallelism, as in each

979-8-3315-8295-1/25/$31.00 ©2025 IEEE
DOI 10.1109/PACT65351.2025.00044

round of the parallel GS algorithm, all unmatched men can simultaneously advance along their preference lists and issue proposals independently of one another. Graphics Processing Units (GPUs) are throughput-oriented manycore processors that offer extremely high peak computational throughput through high memory bandwidth and massive hardware concurrency [48]. This architectural design aligns naturally with the massive parallelism inherent in SMP, enabling thousands of proposals to be processed concurrently. Consequently, GPUs serve as a powerful platform for accelerating large-scale SMP.

However, accelerating the GS algorithm is difficult due to several constraints inherent in its execution pattern. First, the GS algorithm involves frequent access to its data structures. As the number of participants grows, the volume of data movement grows rapidly. When the working set exceeds the cache capacity, cache misses dominate execution time, creating a bottleneck for both sequential and parallel implementations. Second, contention arises in a parallel setting when multiple men propose to the same woman simultaneously. Resolving these conflicts requires synchronization to update the matching state consistently, which introduces substantial overhead and diminishes the benefits of parallelism. Finally, the algorithm exhibits uneven workload distribution. Once a woman becomes matched, she will not become unmatched again [49]. Consequently, the pool of unmatched men decreases over time, which monotonically reduces the available degree of parallelism over time. This gradual reduction in available parallel work creates load imbalance across threads and leads to underutilized parallel resources.

Research on parallel algorithms for SMP has not been fully effective, largely because earlier approaches failed to target the fundamental bottlenecks outlined above. To the best of our knowledge, only two parallel algorithms moderately outperform the sequential GS algorithm: the parallel Gale-Shapley algorithm and the parallel McVitie-Wilson algorithm [50]. Although these approaches reduce synchronization overhead by removing global barriers, they are primarily implemented on CPUs. Prior work has largely overlooked the memory access behavior of the GS algorithm, leaving the locality as underutilized opportunities for improving performance in large-scale SMP computation. Moreover, their performance on GPUs is limited by severe contention for shared resources and imbalanced workload distribution, in some cases making them less efficient than their CPU counterparts.

In this paper, we address three critical research questions for designing a high-performance parallel GS algorithm.

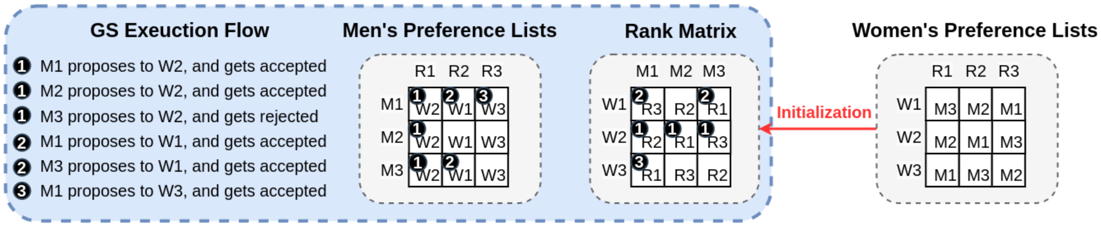
1) Can GS parallel processing be substantially accelerated by exploiting locality?
2) Can advanced GPU hardware primitives be leveraged to improve synchronization efficiency under high contention?
3) Can GPU resources be harnessed for large-scale parallelism while adaptively relying on the CPU to efficiently handle inherently sequential tasks?

To answer these questions, we introduce Bamboo-SMP [1], a high-performance parallel processing framework. Bamboo-SMP incorporates three key design elements: First, we recognize a critical access pattern that exposes locality to exploit. This insight enables the construction of an effective data structure that colocates related data for a locality-aware sequential algorithm, thereby fully leveraging spatial locality and reducing data movements. Second, we parallelize this locality-aware algorithm on the GPU by employing advanced hardware synchronization primitives. This considerably reduces redundant atomic operations under high contention and improves synchronization efficiency in parallel GS processing. Finally, we integrate these techniques into a unified CPU–GPU framework. Bamboo-SMP uses the GPU's high bandwidth and massive thread-level parallelism when a large number of unmatched men remain. As the pool of unmatched men shrinks and only limited parallelism persists, Bamboo-SMP transitions to the CPU, which provides lower latency and effectively executes the remaining sequential work. This adaptive design ensures robust efficiency across diverse workload distributions while fully exploiting the complementary strengths of CPU and GPU architectures.

Specifically, we make the following contributions:

- We design an effective data structure for the GS algorithm that explicitly exploits data locality. By colocating all necessary rank references during the proposing procedure, this locality-aware design minimizes global memory accesses and substantially reduces execution latency.
- We parallelize the locality-aware GS algorithm by leveraging GPU hardware primitives, minimizing atomic operations under high memory contention and improving synchronization efficiency, particularly for workloads with high conflicts in proposals between the two participant groups.
- We integrate these techniques into a unified framework, Bamboo-SMP, for the computation of SMP on both CPUs and GPUs. Bamboo-SMP adaptively exploits the GPU's high bandwidth and massive thread parallelism when ample parallelism is available, and transitions to the CPU to efficiently handle the remaining inherently sequential work. This adaptive execution model ensures robust performance across diverse workload types.
- Through comprehensive experimental evaluations, we demonstrate that Bamboo-SMP outperforms all existing algorithms by **6.69×** to **21.44×** across a wide range of workloads. In addition to its superior performance, Bamboo-SMP also exhibits excellent scalability, efficiently processing SMP instances well beyond the single-GPU memory limit and sustaining speedups of **5.6× to 13.8×** on four GPUs. These results underscore both the effectiveness of Bamboo-SMP and its robust ability to adapt to diverse and large-scale computational demands of SMP.

[1] Bamboo is a unique plant known for its fast growth and resilience in challenging environments. This well aligns with our approach to developing a high-performance SMP solution.


GS Execution Flow

❶ M1 proposes to W2, and gets accepted
❶ M2 proposes to W2, and gets accepted
❶ M3 proposes to W2, and gets rejected
❷ M1 proposes to W1, and gets accepted
❷ M3 proposes to W1, and gets accepted
❸ M1 proposes to W3, and gets accepted

Men's Preference Lists

	R1	R2	R3
M1	❶ W2	❷ W1	❸ W3
M2	❶ W2	W1	W3
M3	❶ W2	❷ W1	W3

Rank Matrix

	M1	M2	M3
W1	❷ R3	R2	❷ R1
W2	❶ R2	❶ R1	❶ R3
W3	❸ R1	R3	R2

Women's Preference Lists

	R1	R2	R3
W1	M3	M2	M1
W2	M2	M1	M3
W3	M1	M3	M2

Fig. 1: Execution flow of the GS algorithm for an SMP instance of 3 men and 3 women.

II. Background

A. The Stable Marriage Problem

The Stable Marriage Problem (SMP) involves two equal-sized groups of participants, say n men and n women. Each participant has a ranked preference list of all members from the opposite group. Figure 1 gives a simple SMP example to match three men and three women, where the two groups are $\{M_1, M_2, M_3\}$ for men and $\{W_1, W_2, W_3\}$ for women. Each member in $\{M_1, M_2, M_3\}$ ranks all members in $\{W_1, W_2, W_3\}$ in a strict order, and vice versa.

Given these two groups, a matching is a one-to-one correspondence from participants in one group to those in the other. A blocking pair in a given matching is a pair of participants from opposite groups who would both prefer each other over their current partners. If such a pair exists, the matching is unstable because these two participants would be motivated to leave their assigned partners and pair up with each other instead. In other words, a matching is stable if no two participants prefer each other over their current partners. The goal of the SMP is to find a stable matching, where no blocking pairs exist.

B. The Gale-Shapley Algorithm

The Gale-Shapley (GS) algorithm, also known as the Deferred Acceptance algorithm, is a foundational method for solving the SMP. We next illustrate the execution flow of the GS algorithm using an example where the preference lists are given in Figure 1.

Initially, all participants are free and the GS algorithm maintains a `Queue` for all unmatched men. In the first round, each man from the `Queue` proposes to his top choice in the first column of the men's preference matrix in Figure 1. Thus, M_1 proposes to W_2, who tentatively accepts as it is her only proposal, forming the match (M_1, W_2). Next, M_2 proposes to W_2. Since W_2 prefers M_2 over M_1, she accepts M_2's proposal and rejects M_1, thereby freeing M_1, who re-enters the `Queue`. This results in (M_2, W_2) as the tentative match. Similarly, M_3 proposes to W_2, gets rejected, and re-enters the `Queue`. The tentative match remains (M_2, W_2).

In the second round, as a free (rejected) man, M_1 proposes to W_1, the next highest-ranked woman in the second column in Men's preference Lists in Figure 1. W_1 tentatively accepts, resulting in a match (M_1, W_1) alongside (M_2, W_2). Subsequently, free man M_3 proposes to W_1. Since W_1 prefers M_3 over M_1, she accepts M_3's proposal and rejects M_1, putting M_1 back into the `Queue` for unmatched men. The tentative matches are now (M_3, W_1) and (M_2, W_2).

Becoming a free man again, M_1 proposes to W_3, the next woman on his preference list in the third round. W_3 accepts M_1's proposal as her best choice, resulting in the match (M_1, W_3) alongside (M_3, W_1) and (M_2, W_2). All participants are now matched, and the algorithm terminates with the stable matching: $(M_1, W_3), (M_2, W_2)$, and (M_3, W_1), with no blocking pairs.

Efficient implementation of the GS algorithm requires determining, in constant time, whether woman w prefers m over her current partner when m proposes to w. This is achieved using a data structure called *rank matrix*, where the entry `RankMatrix`[w, m] represents the rank r of man m in woman w's preference list. As shown in Figure 1, the rank matrix is constructed from the women's preference lists prior to the execution of GS algorithm. Without this precomputation, each of the $O(n^2)$ proposals would require scanning an entire preference list of length $O(n)$, leading to $O(n^3)$ complexity. By enabling constant-time lookups, the rank matrix reduces the overall execution complexity to $O(n^2)$.

The *perfect case* occurs when all men are matched after the first round, resulting in a stable matching with a complexity of $O(n)$. Such a scenario is rare, however, it represents an embarrassingly parallel workload, making it the best case for parallel processing.

C. The Mcvitie-Wilson Algorithm

The McVitie and Wilson (MW) algorithm is another implementation of the GS algorithm, based on the principle that the proposal order does not affect the resulting stable matching [49], [51]. In the GS algorithm, all unmatched men are placed into a `Queue`. During each iteration of the main loop, an unmatched man is taken from the `Queue` to make proposals, and any rejected man is added back to the `Queue`. In the MW algorithm, however, if a proposer is rejected by the woman he proposes to, he immediately moves on to propose to the next woman on his list, instead of being placed back in a `Queue`. Similarly, if a woman accepts a new proposal, her previous partner will continue proposing to his next preference instead of being added back to the `Queue`. This streamlined proposal process eliminates the need for a `Queue` to manage unmatched men and lays the foundation of parallel implementation of MW on GPUs [52].

III. Bottlenecks in GS computation

We examine three major bottlenecks in GS computation: frequent data movement (Subsection III-B), excessive synchronization overhead (Subsection III-C), and irregular parallelism caused by imbalanced workload distribution (Subsection III-D). To ground our analysis, we first design three synthesized SMP workloads based on real-world application statistics in Subsection III-A.

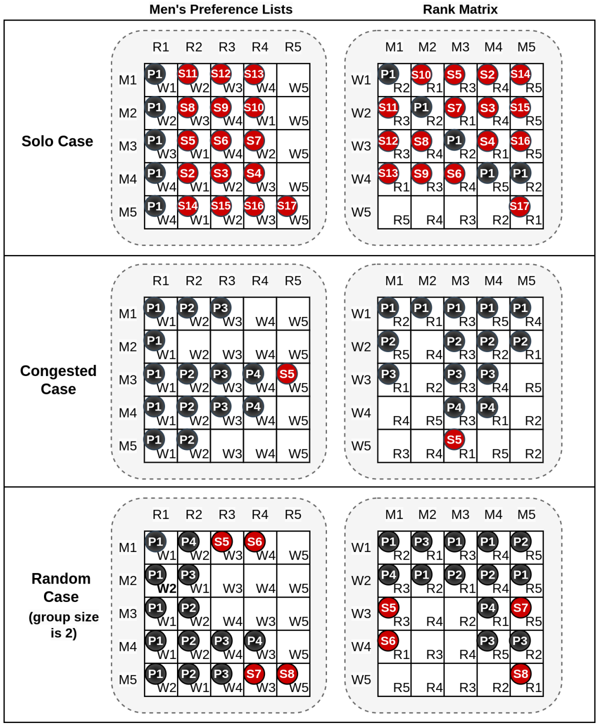
A. Our Approach in the Selection of Workloads

Ideally, bottlenecks in GS computation would be evaluated using real-world data. However, this is infeasible because most SMP applications involve highly confidential information subject to strict privacy constraints. Examples include college admissions, hospital–doctor assignments, organ donation, and kidney exchange, where institutions cannot disclose complete preference lists to the public. Although some researchers have attempted to incorporate real-world data, the missing preference lists are typically synthesized artificially, often failing to capture how participants actually rank one another and thereby reducing their validity as representative workloads for evaluating GS bottlenecks.

To faithfully capture real ranking behavior and construct convincing workloads for performance evaluation, we generate synthetic workloads explicitly informed by real-world matching statistics. The National Resident Matching Program (NRMP) is a United States–based private, non-profit organization established in 1952 to place medical school graduates into residency training programs. Although the complete rank-order lists of applicants and institutions from NRMP are inaccessible due to strict privacy constraints, the NRMP publishes annual reports containing aggregate statistics that capture key features of real-world matching outcomes. Guided by these statistics and representative matching patterns, we design our workload framework and derive three representative SMP workloads: the *solo*, *congested*, and *random cases*.

Figure 2 presents an SMP instance and the corresponding execution flow of the parallel GS algorithm for each synthesized workload type. Independent proposals are executed round by round until a stable matching is reached. Proposal rounds are annotated as **P** for parallel execution and **S** for sequential execution, with subscripts indicating the order (e.g., **P1** for the first parallel round, **S2** for the second sequential step). These annotations are embedded in the men's preference lists and the rank matrix to mark the specific entries accessed in each round.

In the *solo case*, for example, the final step **S17** corresponds to entry (M5, R5) in the preference list and its associated entry (W5, M5) in the rank matrix. The *solo case* exemplifies a predominantly sequential matching process, in which the majority of men are paired during the initial round, while only one or a few continue proposing until the process concludes. This workload reflects patterns observed in NRMP outcomes, where a large fraction of applicants secure top matches—65% in addiction medicine and 53.3% in laryngology obtain their first choice, more than 80% match within their top three, and 16.5% remain unmatched [53]. In Figure 2, after the first parallel round (**P1**), only M4 is rejected, and subsequent proposals (**S2–S17**) proceed sequentially. With n participants on each side, the total number of proposals is $n^2 - (n-1)$, of which only n can be issued concurrently. As n grows, the ratio of parallel to total proposals diminishes toward zero, leaving minimal parallelism to exploit.



Men's Preference Lists

Rank Matrix

Solo Case

	R1	R2	R3	R4	R5
M1	P1 W1	S11 W2	S12 W3	S13 W4	W5
M2	P1 W2	S8 W3	S9 W4	S10 W1	W5
M3	P1 W3	S5 W1	S6 W4	S7 W2	W5
M4	P1 W4	S2 W1	S3 W2	S4 W3	W5
M5	P1 W4	S14 W1	S15 W2	S16 W3	S17 W5

	M1	M2	M3	M4	M5
W1	P1 R2	S10 R1	S5 R3	S2 R4	S14 R5
W2	S11 R3	P1 R2	S7 R1	S3 R4	S15 R5
W3	S12 R3	S8 R4	P1 R2	S4 R1	S16 R5
W4	S13 R1	S9 R3	S6 R4	P1 R5	P1 R2
W5	R5	R4	R3	R2	S17 R1

Congested Case

	R1	R2	R3	R4	R5
M1	P1 W1	P2 W2	P3 W3	W4	W5
M2	P1 W1	W2	W3	W4	W5
M3	P1 W1	P2 W2	P3 W3	P4 W4	S5 W5
M4	P1 W1	P2 W2	P3 W3	P4 W4	W5
M5	P1 W1	P2 W2	W3	W4	W5

	M1	M2	M3	M4	M5
W1	P1 R2	P1 R1	P1 R3	P1 R5	P1 R4
W2	P2 R5	R4	P2 R3	P2 R2	P2 R1
W3	P3 R1	R2	P3 R3	P3 R4	R5
W4	R4	R5	P4 R3	P4 R1	R2
W5	R3	R4	S5 R1	R5	R2

Random Case (group size is 2)

	R1	R2	R3	R4	R5
M1	P1 W1	P4 W2	S5 W3	S6 W4	W5
M2	P1 **W2**	P3 W1	W3	W4	W5
M3	P1 W1	P2 W2	W4	W3	W5
M4	P1 W1	P2 W2	P3 W4	P4 W3	W5
M5	P1 W2	P2 W1	P3 W4	S7 W3	S8 W5

	M1	M2	M3	M4	M5
W1	P1 R2	P3 R1	P1 R3	P1 R4	P2 R5
W2	P4 R3	P1 R2	P2 R1	P2 R4	P1 R5
W3	S5 R3	R4	R2	P4 R1	S7 R5
W4	S6 R1	R3	R4	P3 R5	P3 R2
W5	R5	R4	R3	R2	S8 R1

Fig. 2: Three types of SMP workloads

The *congested case* exemplifies an extreme level of contention, where all men share identical preference lists and compete for the same choices. This dynamic produces widespread rejections, resembling residency specialties such as neurosurgery, dermatology, and general surgery, which exhibit disproportionately high first-choice ratios relative to available positions [33]. In this setting, each round of proposals targets the same woman, so only one can be accepted per step, and the process repeats until the final unmatched man makes a sequential proposal. In Figure 2, steps **P1–P4** involve five, four, three, and two men making proposals in parallel, with men M2, M5, M1, and M4 accepted in sequence. The last sequential step (**S5**) by M3 completes the matching process.

The *random case* captures the clustering behavior observed in real-world applications, where applicants often rank similar options consecutively before considering alternatives. NRMP data, for example, report a median of 12.2 contiguous ranks [53]. To reflect this pattern, we construct workloads by grouping preferences into clusters and shuffling within each group, with the group size adjustable to analyze how different levels of clustering influence contention. Notably, when the group size is set to one, the random case degenerates into the congested case. In Figure 2, the group size is two, producing clusters W1, W2, W3, W4, and W5. During the first two parallel rounds (**P1**, **P2**), all men propose to W1, W2, with M1 and M3 accepted. In the next two rounds (**P3**, **P4**), the remaining unmatched men (M2, M4, M5) propose to W3, W4, leaving one man still unmatched. The final proposals (**S5**–**S8**) then proceed sequentially until a stable matching is achieved.

B. Random Memory Accesses are Harmful

Both GS and MW are memory-bound algorithms, structured as a stream of proposals in which each step performs minimal computation but requires two memory lookups: one to traverse the preference lists and another to query the rank matrix. In particular, rank matrix accesses are costly because they are random and unpredictable at runtime. Such irregularity causes frequent cache misses and high DRAM latency, making these lookups a persistent bottleneck across all workloads.

To demonstrate the impact, we measured the share of execution time spent on rank matrix accesses in GS and MW across multiple workloads. As shown in Figure 3, these lookups account for **up to 75% and at least 55%** of the total execution time, dominating overall performance. Reducing this overhead is therefore a central optimization goal in Bamboo-SMP. However, the random nature of these accesses disrupts traditional techniques such as caching and prefetching [54], [55], which rely on regular patterns. Likewise, GPU-specific mechanisms such as shared memory and specialized caches provide little benefit, since each matrix entry is typically accessed only once and exhibits negligible reuse.

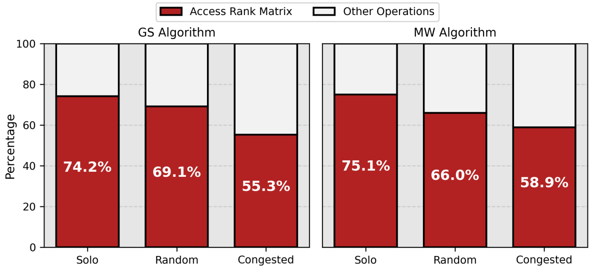


Fig. 3: Time breakdown of the GS and MW algorithms

C. Inefficient Synchronization

Parallelizing the GS and MW algorithms on multi-core CPUs or GPUs leverages their inherent parallelism, with each thread representing a man and making proposals independently. When multiple men propose to the same woman simultaneously, only the most preferred proposer is accepted [49]. Although this process is conceptually simple, efficient parallelization is difficult. Conventional synchronization methods introduce substantial overhead in this context. Locks enforce exclusive access but incur high overhead from frequent acquisitions and releases caused by fine-grained updates of the GS algorithm. Barrier synchronization, by contrast, forces all threads to pause at fixed points, meaning all men must wait after proposing until every woman has processed her choices. Yet not all threads require this synchronization: rejected men could immediately issue new proposals. This mismatch creates idle time and poor resource utilization, making barrier synchronization inefficient. Atomic operations such as compare-and-swap (`atomicCAS`), provide a finer-grained alternative. `atomicCAS` updates a memory location only if it matches an expected value, ensuring correctness without global barriers [56]. In the GS and MW, `atomicCAS` is used to update a woman's partner rank atomically, preventing race conditions. However, under high contention it performs poorly: frequent `atomicCAS` failures lead to repeated retries and wasted work [57]. Our analysis in Section IX shows that in the *congested case*, where all men share identical preference lists, the number of atomic operations can grow to $O(n^3)$, overshadowing the GS algorithm's $O(n^2)$ complexity.

D. Irregular Parallelism Inherent in SMP

CPUs (Central Processing Units) and GPUs (Graphics Processing Units) are two standard, widely available computing units, each tailored to distinct computational roles. Modern CPUs are designed to minimize response time for single-threaded and latency-sensitive tasks. To achieve this, they employ deep and sophisticated pipelines with architectural advances such as out-of-order execution, branch prediction, and speculative execution. In addition, CPUs integrate large last-level caches to exploit spatial and temporal locality, thereby mitigating the high cost of main memory access [58], [59]. Modern GPUs, by contrast, are designed to maximize throughput for massively parallel workloads such as dynamic programming algorithms [60], graph processing [61], [62], and large-scale data analytics [63], [64], [65]. They rely on the SIMT (Single Instruction, Multiple Thread) execution model, enabling tens of thousands of threads to execute concurrently [66], [67].

Although GPUs achieve high computing throughput when data parallelism is plentiful, their design philosophy makes them ill-suited for sequential tasks. GPU pipelines are deliberately simplified to conserve chip area and power within the thermal envelope, with the saved resources repurposed into a larger number of cores and hardware threads to maximize throughput. GPUs also employ a relatively shallow two-level cache hierarchy optimized for bandwidth rather than latency. These trade-offs are highly effective for massively parallel execution but leave GPUs dependent on large degrees of parallelism to hide memory stalls. When parallelism is limited or imbalanced, cores become idle and resources underutilized, ultimately delaying task completion and degrading overall performance. This challenge of irregular parallelism is particularly pronounced in SMP workloads. To illustrate its impact, we conducted comparative experiments on five implementations of the GS and MW algorithms using the *solo case* with 10,000 participants, where most proposals must be executed sequentially.

TABLE I: Performance Comparison of five implementations of the GS and MW algorithms running on CPUs and GPUs

Implementation	Description	Time (ms)
GS-Seq-CPU	The Sequential GS on CPU	412.9
MW-Seq-CPU	The Sequential MW on CPU	451.1
GS-Par-CPU	The Parallel GS on CPU	546.9
MW-Par-CPU	The Parallel MW on CPU	551.5
MW-Par-GPU	The Parallel MW on **GPU**	**26543.7**

The results in Table I show that GPU performance is markedly inferior to CPU performance in sequentially dominated workloads, largely due to GPUs' poor handling of long sequential computations. Specifically, the `MW-Par-GPU` implementation is **58.8× slower** than `MW-Seq-CPU`, a slowdown attributable to both the sequential nature of the *solo case* and the higher memory access latencies of GPUs. Moreover, even `GS-Par-CPU` and `MW-Par-CPU` run slower than their sequential counterparts, primarily due to the synchronization overhead introduced by `atomicCAS` operations.

IV. Bamboo-SMP

Having identified three critical issues limiting the performance of parallel GS and MW, we develop **Bamboo-SMP**, a parallel processing framework designed to deliver high performance across diverse workloads. Bamboo-SMP leverages optimized data structures, advanced atomic operations, and a hybrid CPU–GPU execution model to overcome these bottlenecks and ensure efficient performance.

A. Locality-Aware Computing

1) PRMatrix: A major performance bottleneck in the GS algorithm stems from frequent rank-matrix lookups, whose non-sequential nature incurs substantial overhead. By examining the GS and MW algorithms, we identify a critical property: **there exists a bijection between a man's decision regarding which woman to propose to and his rank in her preference list**. Let `PrefListsM` denote the men's preference lists and `RankMatrix` represent the women's rank matrix. When a man `m` proposes to a woman at rank `r` in his preference list, the woman `PrefListM[m,r]` is identified. The rank of the proposer `m` in this woman's preference list is then determined by accessing `RankMatrix[PrefListM[m,r], m]`. This establishes a direct one-to-one correspondence between entries in the men's preference list and the rank matrix.

To harness this inherent relationship, we introduce a novel locality-aware data structure called **PRMatrix**. In PRMatrix, interrelated entries from the men's (**P**)reference lists and (**R**)ank matrix are co-located within a unified entity, termed a **PRNode**. Each PRNode can be retrieved with a single memory access, simultaneously providing both the target woman and the man's rank within her preference list, thereby eliminating the need for separate lookups in the rank matrix.

Constructing a PRMatrix requires a preprocessing step to consolidate these entries. For example, as illustrated in Figure 4, when man `M1` is proposing to the woman at `R1`, `W2` is retrieved from `PrefListsM[M1,R1]`. Subsequently, `RankMatrix[W2,M1]` is accessed, yielding `R2`. These two data elements are combined into `PRNode(W2,R2)`, which is then stored in `PRMatrix[M1,R1]`.

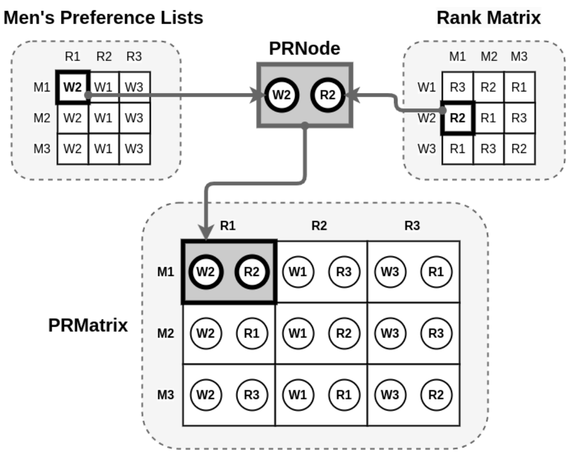


Fig. 4: Construction of PRMatrix

In existing GS implementations, the rank matrix initialization is either omitted [68], which increases the execution-phase complexity from $O(n^2)$ to $O(n^3)$ as stated in Subsection II-B, or executed sequentially on the CPU [69], [70]. In contrast, our approach exploits GPUs to initialize both the rank matrix and PRMatrix in parallel. Although this preprocessing step still requires quadratic time, each matrix entry can be computed independently, making the task embarrassingly parallel and well suited for GPU execution. The massive parallelism and high throughput of GPUs render this additional overhead negligible in practice.

2) LA-Seq-CPU Implementation: Building on PRMatrix, we develop the **(L)**ocality-**(A)**ware **(Seq)**uential implementation of the GS algorithm on the **(CPU)**, referred to as **LA-Seq-CPU**. The algorithm, shown in Algorithm 1, is organized into three phases: *initialization*, *execution*, and *postprocessing*. ❶ *Initialization.* The `LAInit` constructs the PRMatrix on the GPU and initializes two key data structures: `Next`, which records the next woman each man has yet to propose to, and `PartnerRank`, which tracks the rank of each woman's current partner. ❷ *Execution.* In this phase, the subroutine `LAProp` is invoked for each unmatched man. Notably, `LAProp` retrieves a `PRNode` at each iteration (line 13 in Algorithm 1), thereby eliminating random rank-matrix accesses and fully exploiting the locality provided by PRMatrix. ❸ *Postprocessing.* Once all men are matched, `PostProc` finalizes the stable matching `S` by matching each woman to the man at her recorded partner rank.

B. Contention Resolver

As discussed in Subsection III-C, traditional synchronization methods suffer from severe inefficiency under high

Algorithm 1 The LA-Seq-CPU implementation

```
Input: PrefListsM, PrefListsW
Output: A stable matching S
procedure LA-SEQ-CPU PROCEDURE
    PRMatrix, Next, PartnerRank ← LAINIT        ▷ initialization Phase
    for $m \leftarrow 0$ to $n-1$ do                  ▷ execution Phase
        LAPROP($m$)
    end for
    S ← POSTPROC                                ▷ postprocessing Phase
    return S
end procedure
procedure LAPROP($m$)
    $wRank \leftarrow 0$
    while true do
        $pr \leftarrow$ PRMatrix[$m, wRank$]
        $pRank \leftarrow$ PartnerRank[$pr.w$]
        $wRank \leftarrow wRank + 1$
        if $pRank > pr.mRank$ then
            Next[$m$] $\leftarrow wRank$
            PartnerRank[$pr.w$] $\leftarrow pr.mRank$
            if $pRank = n$ then
                break
            else
                $m \leftarrow$ PrefListsW[$pr.w, pRank$]
                $wRank \leftarrow$ Next[$m$]
            end if
        end if
    end while
end procedure
```

contention. To address this limitation, a finer-grained hardware primitive capable of reliably handling such contention is required. NVIDIA CUDA architectures provide advanced atomic primitives for arithmetic synchronization, among which `atomicMin` is particularly well-suited to this task. One `atomicMin` instruction atomically reads the value at a memory address, updates it with the minimum of the current and new values, and returns the original. Crucially, this ensures that each thread completes its operation in a single attempt, eliminating retries and minimizing wasted work [71], [72].

```
1  int m = blockIdx.x * blockDim.x + threadIdx.x;
2  if (m < n) {
3      int wRank = 0;
4      while (true) {
5          PRNode pr = PRMatrix[m * n + wRank];
6          int pRank = atomicMin(&PartnerRank[pr.w], pr
               .mRank);
7          wRank++;
8          if (pRank > pr.mRank) {
9              Next[m] = wRank;
10             if (pRank == n) break;
11             m = PrefListsW[pr.w * n + pRank];
12             wRank = Next[m];
13         }
14     }
15 }
```

Listing 1: Implementation of LA-Par-GPU-MIN Kernel

By leveraging the advanced atomic function in modern GPU architectures, we developed a **(Par)**allelized adaptation of the **(LA)**-Seq-CPU on **(GPU)**, referred to as **LA-Par-GPU**. As detailed in Listing 1, the `LA-Par-GPU` kernel is specifically designed to handle SMP workloads characterized by high contention. It exploits data locality through `PRMatrix` and minimizes wasted work using `atomicMin`, and it is therefore also denoted as **LA-Par-GPU-MIN**. In this kernel, Line 6 directly updates `PartnerRank`[w] with $mRank$, returning the woman's current partner rank $pRank$ without requiring an expected value. If the update fails, man m is rejected and proceeds to the next woman on his preference list. If successful, two cases arise: (1) if $pRank = n$, the woman was previously unmatched and the loop terminates; (2) otherwise, the woman replaces a less-preferred partner, and the CUDA thread assumes the identity of that displaced man (Line 11). This update mechanism illustrates the efficiency of `atomicMin` under high contention. In such scenarios, the maximum number of `atomicMin` operations executed by LA-Par-GPU-MIN is $O(n^2)$, a substantial improvement over the $O(n^3)$ operations required with `atomicCAS`. [2]

C. Hybrid Execution Model

1) Adaptive Execution Mechanisms: While PRMatrix minimizes data movements and the `LA-Par-GPU-MIN` GPU kernel manages contention effectively, both approaches have limitations. In *congested cases* and *random cases*, the locality-aware sequential algorithm may underperform compared to existing parallel GS or MW implementations since it fails to exploit the inherent parallelism of these SMP workloads. In solo cases, the costly synchronization overhead and GPU-memory latency can make `LA-Par-GPU-MIN` even slower than the basic GS algorithm.

To address these limitations, Bamboo-SMP introduces an adaptive execution mechanism that dynamically switches between GPU and CPU to maintain optimal performance across diverse workloads. Execution begins on the GPU with the `LA-Par-GPU-MIN` kernel, enabling all free men to issue proposals concurrently. Throughout the computation, Bamboo-SMP continuously monitors the available degree of parallelism. When only one unmatched man remains, the execution flow shifts to the CPU, where sequential processing becomes more effective. This adaptive transition prevents the slowdown that occurs when reduced-parallelism workloads are left on the GPU and ensures balanced utilization of both devices.

```
do {
    cudaMemcpyAsync(PartnerRankHost,
        PartnerRankDevice, n_ * sizeof(int),
        cudaMemcpyDeviceToHost, memcpyStream);
    cudaStreamSynchronize(memcpyStream);
    int unmatchedID = n * (n - 1) / 2;
    int unmatchedNum = 0;
    int w = 0;
    while (unmatchedNum <= 1 && w < n) {
        int mRank = PartnerRankHost[w];
        if (mRank == n) {
            unmatchedNum++;
        } else {
            int m = PrefListsW[w * n + mRank]
            unmatchedID -= m;
        }
        w++;
    }
} while (unmatchedNum > 1);
```

Listing 2: MonitorProcedure

[2] A detailed mathematical proof is provided in Theorem IX.2 and IX.3.

To implement the adaptive execution policy, we introduce the `MonitorProcedure`, which continuously tracks the matching state and identifies the point at which only one unmatched man remains. At this threshold, the system seamlessly transitions from GPU-based parallel execution to CPU-based sequential processing. The pseudocode for this monitoring routine is provided in Listing 2.

As shown in Listing 2, at the start of each iteration, the device array `PartnerRankDevice` is asynchronously copied to the host (`PartnerRankHost`) using `cudaMemcpyAsync`, creating a snapshot without interrupting the GPU kernel. `cudaStreamSynchronize` ensures the snapshot is complete before proceeding. Two host-side variables are then initialized: `unmatchedNum`, which counts the number of unmatched men, and `unmatchedID`, set to the sum of all men's IDs, $n(n-1)/2$, and used to identify the last unmatched man. The procedure iterates over the host snapshot of partner ranks: if `mRank` equals n, the man is unmatched and `unmatchedNum` is incremented; otherwise, the matched man's ID, `PrefListsW[w * n + mRank]`, is subtracted from `unmatchedID`. When exactly one unmatched man remains, his ID is revealed by `unmatchedID`, and the monitor terminates, triggering a seamless switch to CPU-based locality-aware sequential execution.

2) Heterogeneous Computing Model: Building on the adaptive execution mechanism, Bamboo-SMP employs a heterogeneous computing model that coordinates device–host interaction (Figure 5) to manage data structures and adjust the execution flow, delivering optimal performance across all SMP workloads. The pseudocode for **Bamboo-SMP** is shown in Algorithm 2.[3]

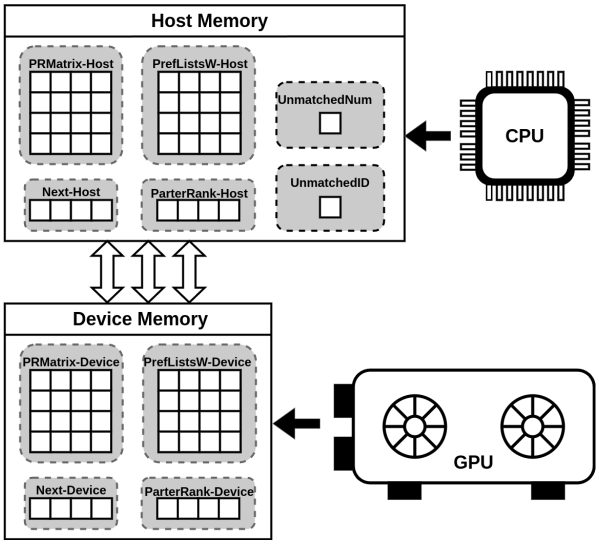


Fig. 5: The Heterogeneous System

Bamboo-SMP prefixes the three-phase design of **LA-Seq-CPU** with an additional *precheck* phase, resulting in four phases: *precheck*, *initialization*, *execution*, and *postprocessing*. ❶ *Precheck.* Bamboo-SMP first checks for trivial perfect cases by letting each man propose to his top choice. If all top choices are distinct, the matching is already stable and the algorithm terminates; otherwise, execution proceeds to initialization. ❷ *Initialization.* If the instance is not perfect, Bamboo-SMP invokes `LAInit` on the GPU to construct the PRMatrix and supporting data structures. An atomic flag, `termFlag`, is also initialized to 0 to indicate that matching is in progress. ❸ *Execution.* Two threads are launched concurrently: `tGPU`, which executes `doWorkOnGPU` to run the `LA-Par-GPU-MIN` kernel, and `tCPU`, which executes `doWorkOnCPU` to run the `MonitorProcedure`. These threads compete to update `termFlag`: if it is set to 1, the GPU completed all proposals; if set to 2, the CPU has detected a single unmatched man and continues sequential execution. ❹ *Postprocessing.* Once either the GPU or CPU completes, the main thread joins the corresponding worker and detaches the other. The stable matching is then finalized directly from `PartnerRankHost`.

Algorithm 2 Bamboo-SMP

```
Input: PrefListsM-Host and PrefListsW-Host
Output: A stable matching S
procedure BAMBOO-SMP
    isPerfect ← PRECHECK                              ▷ precheck phase
    if isPerfect = false then
        Run LAInit on the GPU                         ▷ initialization phase
        termFlag.set(0)
        std::thread tGPU(doWorkOnGPU)                 ▷ execution phase
        std::thread tCPU(doWorkOnCPU)
        while mode == 0 do
            mode ← termFlag.load()
            if mode = 1 then
                tGPU.join()
                tCPU.detach()
            else if mode = 2 then
                tCPU.join()
                tGPU.detach()
            end if
        end while
    end if
    S ← POSTPROC( )                                   ▷ postprocessing phase
    return S
end procedure
```

V. Experiment Evaluation

This section presents our experimental evaluation of Bamboo-SMP to demonstrate the effectiveness of the three key techniques introduced in Section IV: locality-exploitation, contention resolution via `atomicMin`, and the hybrid CPU-GPU computing model.

a) Experimental Setup: All experiments were conducted on a server at the Ohio Supercomputer Center, equipped with 2 AMD EPYC 7643 CPUs totaling 96 cores, 1 NVIDIA Tesla A100 GPU, and 1 TB of host memory. The software environment included g++ version 11.2.0, CMake version 3.25.2, and nvcc version 12.6.77.

[3] The proof of correctness of `LA-Par-GPU-MIN` kernel and our heterogeneous computing model is provided in Theorem IX.7 and IX.8.

b) Baselines: To provide a comprehensive comparison, we implemented all five state-of-the-art implementations discussed in Section III-D. Parallel CPU versions were implemented using C++ standard library threads. To isolate and quantify the contributions of each proposed technique, we also implemented four additional GPU kernels:

- **MW-Par-GPU-CAS**: Parallel MW on GPU with contention resolved using `atomicCAS`.
- **MW-Par-GPU-MIN**: Parallel MW on GPU with contention resolved using `atomicMin`.
- **LA-Par-GPU-CAS**: Locality-aware parallel GS on GPU with contention resolved using `atomicCAS`.
- **LA-Par-GPU-MIN**: Locality-aware parallel GS on GPU with contention resolved using `atomicMin`.

c) Datasets.: The absence of publicly accessible SMP workloads necessitated the use of synthetic datasets to capture the diversity and complexity of real-world scenarios. Table II summarizes the four representative synthetic workloads used in our experiments, highlighting their distinct characteristics and execution pattern.

Workload	**Workload Characteristics**	**Execution Pattern**
Perfect Case	All participants have distinct top choices.	Embarrassingly parallel.

Solo Case	Most participants matched early; few remain unmatched requiring further rounds.	Highly sequential;
		Minimal parallelism;
		Suitable for CPU.

Congested Case	Identical or similar preferences among participants causing frequent rejections.	High contention;
		Substantial parallelism;
		Suitable for GPU.

Random Case	Preferences grouped in clusters with random order within groups.	Moderate contention;
		Substantial parallelism;
		Suitable for GPU.

TABLE II: Summary of Synthetic SMP Workloads.

A. Where Does Time Go?

This subsection evaluates in detail the impact of the PRMatrix and `atomicMin` techniques in both sequential and parallel processing across diverse workloads.

1) Performance Benefits of PRMatrix: We empirically demonstrate that PRMatrix is a broadly effective optimization. It achieves substantial performance gains in *solo cases* and introduces only negligible overhead in other scenarios.

a) Cache behavior: We first examine cache behavior to confirm that PRMatrix improves locality by reducing cache misses. The top row of Fig. 6 reports cache miss ratios for five implementations: `GS-Seq-CPU`, `MW-Seq-CPU`, `LA-Seq-CPU`, `MW-Par-GPU-CAS`, and `LA-Par-GPU-CAS`, evaluated on *solo*, *congested*, and *random* workloads. The GPU implementations are omitted in the *solo* case due to severe bottlenecks during sequential execution, as discussed in Section III-D. **Across all three workloads, PRMatrix reduces cache miss ratios significantly.** In the *solo* case on CPU, cache miss ratios fall from 23.4% (`GS-Seq-CPU`) and 22.9% (`MW-Seq-CPU`) to just 3.1% (`LA-Seq-CPU`). In the *congested* case, CPU miss ratios decrease from 28.9% and 27.3% to 15.7%, while GPU miss ratios drop from 41.6% (`MW-Par-GPU-CAS`) to 20.0% (`LA-Par-GPU-CAS`). In the *random* case, CPU miss ratios fall to 11.4%, and GPU ratios drop from 46.6% to 23.7%.

b) Runtime performance: We next examine runtime performance. The bottom row of Fig. 6 shows initialization and execution times (in seconds) for the same implementations and workloads, aligned vertically with the cache plots above. GPU-based parallel initialization is applied uniformly across all implementations to ensure fairness. **Overall, PRMatrix achieves remarkable speedups across all workloads while introducing only negligible overhead.** In the *solo* case, `LA-Seq-CPU` achieves a 4.64× speedup, reducing runtime from 19.00s (`GS-Seq-CPU`) to 4.09s. This improvement arises from eliminating a large proportion of random memory accesses. In the *congested* case, `LA-Seq-CPU` reduces runtime from 53.81s to 28.50s, a 1.88× speedup, while on GPU, `LA-Par-GPU-CAS` slightly outperforms `MW-Par-GPU-CAS` (0.58s versus 0.64s). In the *random* case, `LA-Seq-CPU` reduces runtime from 31.50s to 11.14s, a 2.82× improvement, and on GPU, `LA-Par-GPU-CAS` performs comparably to `MW-Par-GPU-CAS` (0.46s versus 0.45s).

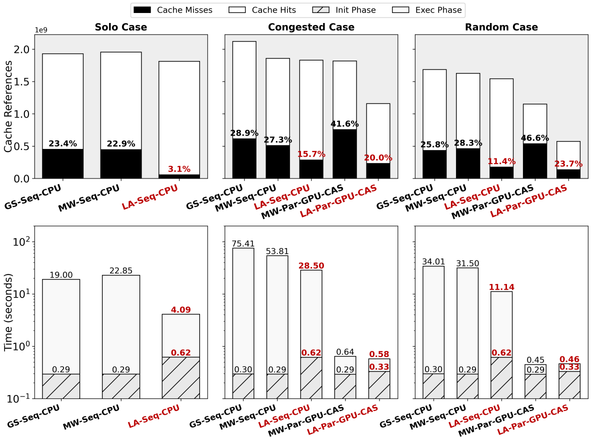


Fig. 6: Cache behavior (top) and runtime performance (bottom) of five implementations across Solo (left), Congested (center), and Random workloads (right).

2) Operational efficiency of Contention Resolver: To evaluate the effectiveness of `atomicMin`, we evaluated four GPU kernels on *congested* and *random* workloads. The top row of Fig. 7 reports their execution times, and the bottom row shows the number of atomic operations, profiled with `ncu`. The baseline kernel, `MW-Par-GPU-CAS`, employs no optimizations. `MW-Par-GPU-MIN` introduces `atomicMin` to mitigate contention and improve synchronization. `LA-Par-GPU-CAS` leverages PRMatrix to improve memory locality and reduce data movement. Finally, `LA-Par-GPU-MIN` combines both techniques to capture the benefits of each. These results confirm that `atomicMin` significantly enhances synchronization efficiency under contention, and that its benefits extend across both *congested* and *random* workloads when combined with PRMatrix.

In the *congested* case (top-left subplot), with workload sizes up to 30,000, the baseline kernel underperforms compared to its optimized variants. At the largest workload size, atomicMin alone achieves a 2.33× speedup, PRMatrix achieves 1.29×, and their combination delivers a 3.44× speedup. These improvements correspond closely to the reduction in global atomic operations enabled by atomicMin, which eliminates wasted work under heavy contention. As shown in the bottom-left subplot, both MW-Par-GPU-MIN and LA-Par-GPU-MIN nearly halve the number of atomic operations compared to their atomicCAS-based counterparts.

For *random* workloads (right subplots), we fixed the total workload size at 30,000 and varied group sizes from 1 to 50. The top-right subplot shows that both PRMatrix and atomicMin improve performance across all group sizes. Although contention naturally decreases as group size grows, resulting in a less pronounced reduction in atomic operations as shown in the bottom-right subplot, the combined kernel consistently achieves the best execution times.

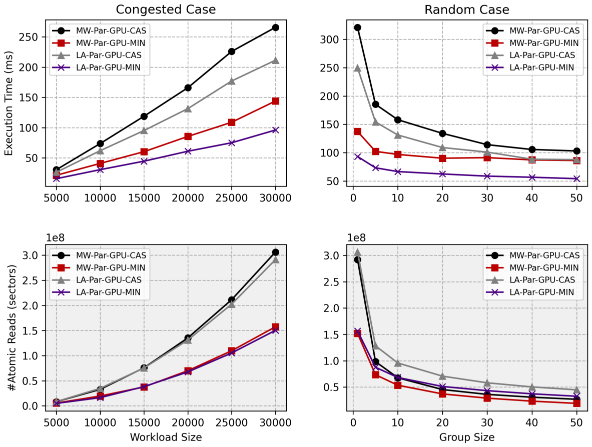


Fig. 7: Execution times (top) and atomic operations (bottom) of four GPU kernels for Congested (left) and Random (right) workloads.

B. Performance Evaluation

To demonstrate the high performance of Bamboo-SMP relative to state-of-the-art algorithms, we evaluated it on four representative scenarios: *perfect*, *congested*, *random*, and *solo*. For each scenario, the GPU workload size was scaled to 30,000. Figure 8 presents, for every implementation and scenario, the breakdown of time spent in the prechecking phase (specific to Bamboo-SMP), initialization, execution, and postprocessing. In baseline algorithms, the rank matrix is initialized on the CPU, consistent with their original design. In contrast, Bamboo-SMP initializes both the rank matrix and PRMatrix directly on the GPU, exploiting massive parallelism as described in Section IV-A.

From these results, we draw the following observations. ❶ Across all workloads, Bamboo-SMP consistently achieves the lowest latency, underscoring its robustness and effectiveness as a general solution. It outperforms MW-Par-GPU-CAS by over 20× in the *congested* and *random* cases, and MW-Par-CPU by more than 6× in the *solo* case. ❷ The prechecking step in Bamboo-SMP and the postprocessing step in all algorithms consume negligible time and have no impact on overall performance. ❸ In the *perfect* case, Bamboo-SMP bypasses unnecessary phases entirely, sharply reducing total runtime. ❹ In all workloads, GPU-based initialization of PRMatrix in Bamboo-SMP delivers substantial speedup by exploiting independence across rank matrix entries. ❺ In *solo* cases, Bamboo-SMP transitions execution to the host once only one unmatched man remains, leveraging the CPU's low-latency operations and eliminating rank matrix access via PRMatrix. ❻ In *congested* and *random* cases, CUDA kernels reduce execution time through massive parallelism, further enhanced by the combined benefits of PRMatrix and the contention resolver.

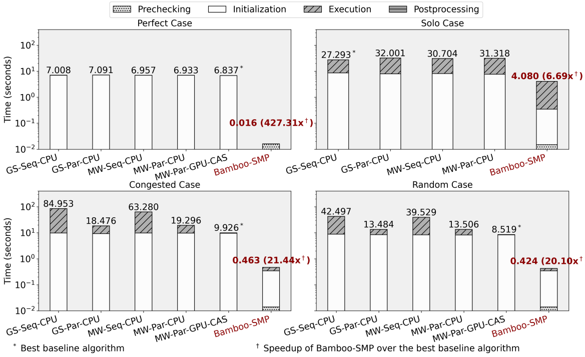


Fig. 8: Performance comparison of Bamboo-SMP and baseline algorithms

C. Scalability Evaluation on Multi-GPUs

The scalability of Bamboo-SMP is inherently bounded by the memory capacity of a single GPU, which on the A100 corresponds to problem sizes of up to about 30K participants. Scaling data size beyond this limit requires distributing larger workloads across multiple GPUs, which evaluates algorithm's ability to maintain high performance as additional processors are used to handle larger problem sizes. To show Bamboo-SMP's scalability as the number of SMP instances increases, we implemented a multi-GPU version on four-GPU A100 nodes interconnected via NVLink, without relying on unified memory. During execution, each GPU manages the preference lists and associated data of a disjoint subset of participants, partitioned evenly by participant ID, while GPU kernels directly access remote GPUs whenever non-local data is required.

Figure 9 presents the execution times of Bamboo-SMP on 4 GPUs as workload sizes increase to 90K participants. Since these problem sizes exceed the single-GPU memory limit, we compare performance against the best CPU baselines: GS-PAR-CPU (Random and Congested) and GS-SEQ-CPU (Solo). The wide gap between the baseline curves (black dashed lines) and Bamboo-SMP (red solid lines) clearly demonstrates substantial speedups, reaching up to 10.9× (Random), 13.8× (Congested), and 5.6× (Solo). These performance gains illustrate Bamboo-SMP's strong multi-GPU scalability and its critical role in solving large-scale SMP instances.

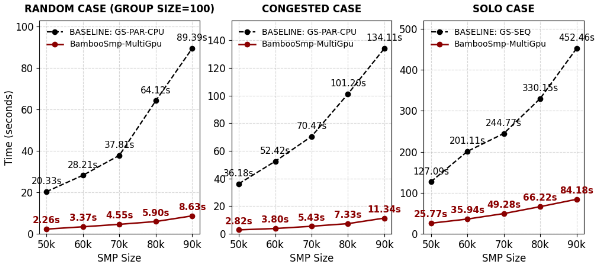


Fig. 9: Execution times of Bamboo-SMP with increasing problem sizes on a multi-GPU platform

The key to achieving high scalability is to ensure that the average parallel processing overhead or latency—including synchronization delays and inter-GPU communication latency—increases only minimally as the workload size and the number of GPU nodes grow [73]. The average latency is computed as the total overhead across all nodes divided by the number of nodes, enabling proportional and balanced scaling of workloads with respect to the GPU cluster size. Our experiments demonstrate that when scaling SMP workloads across two, three, or more GPU nodes, the average latency increases only marginally with each additional node. This low overhead enables efficient execution of large-scale workloads across dozens of GPU nodes. Furthermore, as inter-node GPU communication hardware continues to advance, the overall scalability of such systems is expected to improve substantially.

VI. Related Work

In addition to the parallelized GS and MW algorithms, several theoretical parallel approaches have been proposed to solve the SMP with complexities lower than the GS algorithm's $O(n^2)$. SMP can be framed as a linear programming and solved using the primal-dual interior path method [74], [75], [76]. This approach achieves an impressive sublinear time complexity of $O(n^{1/2} \log^3 n)$. However, it requires at least $O(n^4)$ processors, making it hard to parallelize in practice. To reduce the number of processors, a parallel iterative improvement method was proposed, which starts with a randomly generated initial matching and iteratively refines it towards stability [77]. This method [78] and its variations [79], [80] still require n^2 processors for parallel processing. A divide and conquer parallel algorithm was developed with a time complexity of $O(n \log \log n)$, utilizing n processors on a Concurrent Read Exclusive Write (CREW) Parallel Random Access Machine (PRAM) [81], [82], and it runs slower than the sequential GS algorithm [83]. A two-phase parallelization of the GS algorithm uses the master-slave strategy, alternating between rounds of proposals and rejections [83], and its inefficiency was demonstrated in [56].

VII. Conclusion

We presented Bamboo-SMP, a high-performance parallel framework that reduces data access latency through a locality-aware structure, leverages CUDA-based synchronization to eliminate redundant atomic operations under high contention, and integrates a GPU–CPU hybrid computing model to ensure efficiency across diverse workloads. Our experimental results demonstrate that Bamboo-SMP consistently outperforms existing algorithms, achieving significant speedups across a wide range of workloads. Its performance improvements and multi-GPU scalability underscore its potential to solve large-scale SMP problems and to meet real-time requirements in practical applications.

VIII. Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions. The work is supported in part by the U.S. National Science Foundation under grants MRI[1]2018627, CCF-2005884, CCF-2210753, CCF-2312507, and OAC-2310510.

IX. Addendum: The Soundness of Bamboo-SMP

Lemma IX.1. *To find the minimum among n numbers using n threads with `atomicCAS` on a shared memory location (initialized with a value larger than all n numbers), the total number of `atomicCAS` operations is $O(n^2)$.*

Proof. Let the initial value in the shared memory be v_{n+1}, and let the values proposed by the threads be $v_1 < v_2 < \cdots < v_n < v_{n+1}$. The thread with the smallest value v_1 executes `atomicCAS` only once. The thread with the second smallest value v_2 may execute at most twice: it fails once due to v_1, then succeeds or terminates on its second attempt. In general, the thread with the k-th smallest value v_k executes at most k times, failing once for each of the $k-1$ smaller values that updated the location earlier. Thus, the maximum number of executions is $T(n) = \sum_{k=1}^{n} k = O(n^2)$. □

Theorem IX.2. *For an SMP instance with n men and n women, the number of `atomicCAS` executions is $O(n^3)$.*

Proof. Consider the worst case where all n men share identical preference lists. In the first round, all n threads contend for the same memory location, requiring $O(n^2)$ operations by Lemma IX.1. In the second round, $n-1$ men remain unmatched, again incurring $O(n^2)$ operations. This pattern continues across n rounds. Therefore, the total number of `atomicCAS` executions is $\sum_{i=1}^{n} O(n^2) = O(n^3)$. □

Theorem IX.3. *For an SMP instance with n men and n women, the maximum number of `atomicMin` executions is $O(n^2)$.*

Proof. Let the initial value in the shared memory be v_{n+1}, and let the proposed values be $v_1 < v_2 < \cdots < v_n < v_{n+1}$. Each thread executes `atomicMin` exactly once, since the operation always succeeds by writing the minimum of the proposed and current value. Hence, determining the minimum among n values requires at most n executions. As this process repeats for each of the n rounds of proposals in the SMP instance, the worst-case total is $O(n^2)$. □

During the execution of the `LA-Par-GPU-MIN` kernel, each man is either **matched** or **unmatched**. A man m is matched with a woman w if `prefListsW[w, partnerRank[w]] = m`; otherwise, he is unmatched. A man is **active** if there exists a thread executing `atomicMin(&partnerRank[w], mRank)` with `prefListsW[w, mRank] = m`; otherwise he is **inactive**.

Lemma IX.4. *Throughout execution, if a man is matched, he is inactive; conversely, if a man is unmatched, he is active.*

Proof. We prove this lemma by induction.

Base Case. Initially, every man is unmatched and active, since each thread begins by executing a proposal via `atomicMin`.

Induction Hypothesis. Assume that at some execution point, this lemma holds.

Induction Step. Although a woman may receive multiple proposals concurrently, each is issued through `atomicMin`, which serializes updates. For a proposal `atomicMin(&partnerRank[w], mRank)`, one of three cases can occur:

1) If w is unmatched, she accepts. Man m becomes matched, his thread terminates, and he becomes inactive.
2) If w is matched with a more-preferred man (`partnerRank[w] < mRank`), she rejects. Man m remains unmatched, advances to the next woman on his list, and thus remains active.
3) If w is matched with a less-preferred man (`partnerRank[w] > mRank`), she accepts. Man m becomes matched and inactive, while the displaced partner becomes unmatched, resumes proposals on that thread, and is therefore active.

In all cases, the invariant is preserved. □

Lemma IX.5. *`LA-Par-GPU-MIN` kernel terminates with a matching.*

Proof. To begin, termination of the `LA-Par-GPU-MIN` kernel is equivalent to reaching a matching. If the kernel halts, no threads remain active. By Lemma IX.4, this implies that all men are matched with distinct women, and hence the result is a complete matching. Conversely, once a complete matching is formed, no further proposals occur and all threads terminate.

To finish the argument, we show this kernel always terminates in two steps. *1. The number of acceptances is finite.* Suppose that $n(n-1)+2$ acceptances occur. Then at least one woman must accept twice after all others had already accepted once, which implies all women were already matched before the latest acceptance. But this contradicts the existence of that last acceptance. Hence the number of acceptances is at most $n(n-1)+1$, and therefore finite. *2. Every active man makes progress.* If a man is active, he eventually produces an acceptance. If he were rejected by all women, then all women would already be matched, leaving no active men and contradicting the fact that he is making proposals. Thus, every active man contributes to the bounded set of acceptances. Since the number of acceptances is finite and every active man eventually produces one, the kernel must terminate. □

Lemma IX.6. *`LA-Par-GPU-MIN` kernel always terminates with a* ***stable*** *matching.*

Proof. We prove this by contradiction. Assume that the matching μ produced by `LA-Par-GPU-MIN Kernel` is not stable. Then there must exist a blocking pair (m, w) such that m prefers w over $\mu(m)$, and w prefers m over $\mu(w)$. During execution, when m proposes to $\mu(m)$ and is accepted, m has already proposed to w and been rejected. At that time, w was already matched with a man she preferred over m. Therefore, $\mu(w)$ is strictly preferred by w to m, contradicting the assumption that w prefers m over $\mu(w)$. Thus, no blocking pair exists, and μ is a stable matching. □

Theorem IX.7. *The `LA-Par-GPU-MIN Kernel` must terminate with a* ***man-optimal*** *stable matching, where no man can obtain a more preferred partner in any other stable matching.*

Proof. Suppose the matching μ produced by `LA-Par-GPU-MIN` is stable but not man-optimal. Then there exists another stable matching μ' and a man m such that m prefers $\mu'(m) = w'$ over $\mu(m) = w$. During execution, some men may be rejected by their partners in μ'. Let m be among the first rejected. Suppose m is rejected because w' is matched with m', which means w' prefers m' to m. Let $\mu'(m') = w''$. Since m is rejected first, m' could not have been rejected by w'', so m' must prefer w' to w''. Otherwise, m' would have proposed to w'' first and been rejected before turning to w', contradicting the minimality of m's rejection. Thus, in μ', both m' and w' prefer each other over their partners, forming a blocking pair. This contradicts the stability of μ'. Hence, μ must be man-optimal. □

Theorem IX.8. *The Bamboo-SMP hybrid execution model must terminate with a* ***man-optimal*** *stable matching*

Proof. The correctness of our heterogeneous computing model holds regardless of which device produces the matching. If all computation is performed on the GPU, the proof of Theorem IX.7 applies directly. If the CPU takes over, the snapshot of proposal states determines which man proceeds, exactly as it would on the GPU, thereby preserving correctness. □

References

[1] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15, 1962.

[2] Y. Geng and M. Gao, "Distributed stable marriage with incomplete list and ties using spark," *S15/projects/stable_marriage_spark_report. pdf*, 2015.

[3] S. Wu, L. H. U, and P. Karras, "k-best egalitarian stable marriages for task assignment," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 3240–3252, 2023.

[4] J. Hirvonen and S. Ranjbaran, "Fast, fair and truthful distributed stable matching for common preferences," *arXiv preprint arXiv:2402.16532*, 2024.

[5] B. Li, Y. Cheng, G. Wang, and Y. Sun, "Incremental bilateral preference stable planning over event based social networks," *Complexity*, vol. 2019, no. 1, p. 1532013, 2019.

[6] R. Anurag and A. Bhattacharya, "Sms: Stable matching algorithm using skylines," in *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, 2016, pp. 1–4.

[7] K. Fritsch and S. Scherzinger, "Solving hard variants of database schema matching on quantum computers," *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3990–3993, 2023.

[8] P. A. Bernstein, S. Melnik, and J. E. Churchill, "Incremental schema matching," in *VLDB*, vol. 6. Seoul, Korea, 2006, pp. 1167–1170.

[9] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *Proceedings 18th international conference on data engineering*. IEEE, 2002, pp. 117–128.

[10] A. Jhingran, "Moving up the food chain: Supporting e-commerce applications on databases," *ACM SIGMOD Record*, vol. 29, no. 4, pp. 50–54, 2000.

[11] S. Scott, *A study of stable marriage problems with ties*. University of Glasgow (United Kingdom), 2005.

[12] B. Gao, Z. Zhou, F. Liu, F. Xu, and B. Li, "An online framework for joint network selection and service placement in mobile edge computing," *IEEE Transactions on Mobile Computing*, vol. 21, no. 11, pp. 3836–3851, 2021.

[13] B. Genc, M. Siala, G. Simonin, and B. O'Sullivan, "On the complexity of robust stable marriage," in *International Conference on Combinatorial Optimization and Applications*. Springer, 2017, pp. 441–448.

[14] L. Du, P. Cheng, L. Chen, W. Ni, J. Zhao, and X. Lin, "Stable task assignment with range partition under differential privacy," in *International Conference on Database Systems for Advanced Applications*. Springer, 2024, pp. 243–253.

[15] R. C.-W. Wong, "Spatial matching." 2017.

[16] L. Guanjie, M. Zeng, K. Wu, G. Wang, Z. Shan, and K. Lei, "Blockchain-based cooperative game bilateral matching architecture for shared storage," *Available at SSRN 4654316*.

[17] Z. Chen, P. Cheng, L. Chen, X. Lin, and C. Shahabi, "Fair task assignment in spatial crowdsourcing," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020.

[18] Y. Tong, J. She, B. Ding, L. Wang, and L. Chen, "Online mobile micro-task allocation in spatial crowdsourcing," in *2016 IEEE 32Nd international conference on data engineering (ICDE)*. IEEE, 2016, pp. 49–60.

[19] J. She, Y. Tong, L. Chen, and C. C. Cao, "Conflict-aware event-participant arrangement and its variant for online setting," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 9, pp. 2281–2295, 2016.

[20] B. Zhao, P. Xu, Y. Shi, Y. Tong, Z. Zhou, and Y. Zeng, "Preference-aware task assignment in on-demand taxi dispatching: An online stable matching approach," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 2245–2252.

[21] J. She, Y. Tong, L. Chen, and C. C. Cao, "Conflict-aware event-participant arrangement and its variant for online setting," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 9, pp. 2281–2295, 2016.

[22] Y. Tong, L. Wang, Z. Zimu, B. Ding, L. Chen, J. Ye, and K. Xu, "Flexible online task assignment in real-time spatial data," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1334–1345, 2017.

[23] B. Li, Y. Cheng, Y. Yuan, G. Wang, and L. Chen, "Simultaneous arrival matching for new spatial crowdsourcing platforms," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, ser. IJCAI'20, 2021.

[24] A. E. Roth, T. Sönmez, and M. U. Ünver, "Kidney exchange," *The Quarterly journal of economics*, vol. 119, no. 2, pp. 457–488, 2004.

[25] N. Seidi, "A stable matching assignment for cancer treatment centers using survival analysis," *arXiv preprint arXiv:2401.10469*, 2024.

[26] L. Huang, K. Zhang, Y. Sun, G. Shen, and D. Coursey, "Application of gale-shapley algorithm in optimal matching for healthcare facilities to elderly population: the case of hangzhou, china," *Applied Economics*, pp. 1–12, 2024.

[27] A. Abdulkadiroğlu, P. A. Pathak, and A. E. Roth, "The new york city high school match," *American Economic Review*, vol. 95, no. 2, pp. 364–367, 2005.

[28] Z. Sun, N. Yamada, Y. Takenami, D. Moriwaki, and M. Yokoo, "Stable matchings in practice: A constraint programming approach," *arXiv preprint arXiv:2401.07761*, 2024.

[29] P. Biró, A. Hassidim, A. Romm, R. I. Shorrer, and S. Sovago, "The large core of college admission markets: Theory and evidence," in *Proceedings of the 23rd ACM Conference on Economics and Computation*, 2022, pp. 958–959.

[30] A. Khalili-Fard, R. Tavakkoli-Moghaddam, N. Abdali, M. Alipour-Vaezi, and A. Bozorgi-Amiri, "A roommate problem and room allocation in dormitories using mathematical modeling and multi-attribute decision-making techniques," *Journal of Modelling in Management*, 2024.

[31] C. Yang, Y. Hou, Y. Song, T. Zhang, J.-R. Wen, and W. X. Zhao, "Modeling two-way selection preference for person-job fit," in *Proceedings of the 16th ACM Conference on Recommender Systems*, 2022, pp. 102–112.

[32] R. Kaur, V. Goyal, V. M. Gunturi, and C. Long, "A matching based spatial crowdsourcing framework for egalitarian task assignment," in *2022 23rd IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 2022, pp. 185–187.

[33] National Resident Matching Program, "Charting outcomes™: Characteristics of u.s. do seniors who matched to their preferred specialty: 2024 main residency match," Online, 2025, available from: https://www.nrmp.org/match-data/2024/08/charting-outcomes-characteristics-of-u-s-do-seniors-who-matched-to-their-preferred-specialty-2024-main-residency-match/.

[34] P. Xia, B. Liu, Y. Sun, and C. Chen, "Reciprocal recommendation system for online dating," in *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, 2015, pp. 234–241.

[35] B. Zhao, P. Xu, Y. Shi, Y. Tong, Z. Zhou, and Y. Zeng, "Preference-aware task assignment in on-demand taxi dispatching: an online stable matching approach," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33012245

[36] N. M. Kou, L. H. U, N. Mamoulis, Y. Li, Y. Li, and Z. Gong, "A topic-based reviewer assignment system," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1852–1855, 2015.

[37] Y. Tong, Z. Zhou, Y. Zeng, L. Chen, and C. Shahabi, "Spatial crowdsourcing: a survey," *The VLDB Journal*, vol. 29, pp. 217–250, 2020.

[38] H. Xu and B. Li, "Egalitarian stable matching for vm migration in cloud computing," in *2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2011, pp. 631–636.

[39] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 3, pp. 52–66, 2015.

[40] T. Wang, F. Liu, J. Guo, and H. Xu, "Dynamic sdn controller assignment in data center networks: Stable matching with transfers," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[41] W. N. W. Muhamad, S. S. M. Aris, K. Dimyati, M. A. Javed, A. Idris, D. M. Ali, and E. Abdullah, "Energy-efficient task offloading in fog computing for 5g cellular network," *Engineering Science and Technology, an International Journal*, vol. 50, p. 101628, 2024.

[42] S. S. Yellampalli, M. Chalupa, J. Wang, H. J. Song, X. Zhang, H. Yue, and M. Pan, "Client selection in federated learning: A dynamic matching-based incentive mechanism."

[43] Y. Zhang, L. Cui, and Y. Zhang, "A stable matching based elephant flow scheduling algorithm in data center networks," *Computer Networks*, vol. 120, pp. 186–197, 2017.

[44] B. Li, Y. Cheng, Y. Yuan, G. Wang, and L. Chen, "Three-dimensional stable matching problem for spatial crowdsourcing platforms," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1643–1653. [Online]. Available: https://doi.org/10.1145/3292500.3330879

[45] H. AlHakami, F. Chen, and H. Janicke, "An extended stable marriage problem algorithm for clone detection," *arXiv preprint arXiv:1408.2969*, 2014.

[46] National Resident Matching Program, "Results and Data: 2025 Main Residency Match," https://www.nrmp.org/about/news/2025/05/nrmp-releases-2025-main-residency-match-results-and-data-report-providing-in-depth-insight-into-the-largest-residency-match-in-history/, 2025, accessed: 2025-09-06.

[47] D. Fan, R. Lee, and X. Zhang, "X-blossom: Massive parallelization of graph maximum matching."

[48] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, Ü. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient sparse-matrix multi-vector product on gpus," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 66–79.

[49] D. Gusfield and R. W. Irving, *The stable marriage problem: structure and algorithms*. MIT press, 1989.

[50] F. Manne, M. Naim, H. Lerring, and M. Halappanavar, "On stable marriages and greedy matchings," in *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 2016, pp. 92–101.

[51] D. G. McVitie and L. B. Wilson, "The stable marriage problem," *Communications of the ACM*, vol. 14, no. 7, pp. 486–490, 1971.

[52] ——, "Algorithm 411: Three procedures for the stable marriage problem," *Commun. ACM*, vol. 14, no. 7, p. 491–492, jul 1971. [Online]. Available: https://doi.org/10.1145/362619.362632

[53] National Resident Matching Program, "Results and data: Specialties matching service, 2025 appointment year," Online, 2025, available from: https://www.nrmp.org/match-data/2025/02/specialty-match-program-results-2021-2025/.

[54] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 32–41.

[55] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[56] H. H. Lerring, "Parallel algorithms for matching under preference," Master's thesis, The University of Bergen, 2017.

[57] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 103–112.

[58] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 451–460.

[59] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[60] D. Fan, R. Lee, and X. Zhang, "X-ted: Massive parallelization of tree edit distance," *Proc. VLDB Endow.*, vol. 17, no. 7, p. 1683–1696, Mar. 2024. [Online]. Available: https://doi.org/10.14778/3654621.3654634

[61] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on gpu," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 38–52. [Online]. Available: https://doi.org/10.1145/3293883.3295733

[62] K. Meng, L. Geng, X. Li, Q. Tao, W. Yu, and J. Zhou, "Efficient multi-gpu graph processing with remote work stealing," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 191–204.

[63] L. Geng, R. Lee, and X. Zhang, "Librts: A spatial indexing library by ray tracing," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 396–411. [Online]. Available: https://doi.org/10.1145/3710848.3710850

[64] ——, "Rayjoin: Fast and precise spatial join," in *Proceedings of the 38th ACM International Conference on Supercomputing*, ser. ICS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 124–136. [Online]. Available: https://doi.org/10.1145/3650200.3656610

[65] S. Zhang, M. Xiao, C. Guo, L. Geng, H. Wang, and X. Zhang, "Hypha: a framework based on separation of parallelisms to accelerate persistent homology matrix reduction," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 69–81. [Online]. Available: https://doi.org/10.1145/3330345.3332147

[66] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[67] Z. Jia and P. Van Sandt, "Dissecting the ampere gpu architecture via microbenchmarking," in *GPU Technology Conference*, vol. 43, 2021.

[68] H. Wilde, V. Knight, and J. Gillard, "Matching: A python library for solving matching games," *Journal of Open Source Software*, vol. 5, no. 48, p. 2169, 2020.

[69] J. P. Krishnaa and R. Meenakshi, "CS6023: Matching with Preferences," https://github.com/meenakshiravisankar/stable-matching/blob/master/GPU_Project_Report.pdf, 2020, course Project Report, Roll No: CS14B049, AE15B051. Accessed: 2025-04-20.

[70] N. Udhayasankar, "A parallel approach to the stable matching problem," https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Naveen-Udhayasankar-Spring-2022.pdf, 2022, cSE 633 Course Project Report, Spring 2022. Accessed: 2025-04-20.

[71] NVIDIA, "Cuda toolkit documentation," Online, 2024, available from: https://docs.nvidia.com/cuda/.

[72] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016, pp. 1–13.

[73] X. Zhang, Y. Yan, and K. He, "Latency metric: An experimental method for measuring and evaluating parallel program and architecture scalability," *Journal of Parallel and Distributed Computing*, vol. 22, no. 3, pp. 392–410, 1994.

[74] T. Feder, N. Megiddo, and S. A. Plotkin, "A sublinear parallel algorithm for stable matching," *Theoretical computer science*, vol. 233, no. 1-2, pp. 297–308, 2000.

[75] A. Subramanian, "A new approach to stable matching problems," *SIAM Journal on Computing*, vol. 23, no. 4, pp. 671–700, 1994.

[76] T. Fleiner, "A fixed-point approach to stable matchings and some applications," *Mathematics of Operations research*, vol. 28, no. 1, pp. 103–126, 2003.

[77] E. Lu and S. Zheng, "A parallel iterative improvement stable matching algorithm," in *International Conference on High-Performance Computing*. Springer, 2003, pp. 55–65.

[78] A. A. Barkley and J. A. Martin, "Implementing the parallel iterative improvement algorithm for the stable marriage problem on gpus."

[79] S. Wynn, A. Kyritsis, S. Alberi, and E. Lu, "Selection improvements for the parallel iterative algorithm for stable matching," *arXiv preprint arXiv:2401.07467*, 2024.

[80] C. White and E. Lu, "An improved parallel iterative algorithm for stable matching," *SuperComputing 2013, Denver, Colorado, USA*, 2013.

[81] S.-S. Tseng and R. C. T. Lee, "A parallel algorithm to solve the stable marriage problem," *BIT Numerical Mathematics*, vol. 24, pp. 308–316, 1984.

[82] S. Tseng, "The average performance of a parallel stable marriage algorithm," *BIT Numerical Mathematics*, vol. 29, pp. 448–456, 1989.

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

Appendix

Artifact Evaluation

A. Availability

The artifact is available at both GitHub: https://github.com/victorliu-sq/PACTAE and Zenodo: https://zenodo.org/records/16800775. Users can clone the repository to your local machine or HPC environment, then run:

```
./runme.sh
```

This will automatically generate all experimental results presented in the paper.

B. Hardware Requirements

- At least one NVIDIA GPU, with each Streaming Multiprocessor (SM) supporting a block size of at least 1024 threads.
- A multi-core CPU of at least 32 threads.
- At least 24 GB of both host memory and GPU memory.
- At least 120 GB of free disk space for storing generated synthetic data.

C. Software Requirements

The following software must be installed and available in your `PATH`:

- `bash`
- `wget` or `curl`
- `perf`
- `ncu`
- GCC $\geq$ 11.4.0
- CMake $\geq$ 3.22.1
- Python $\geq$ 3.10
- CUDA Toolkit $\geq$ 12.6

D. Evaluation

If the above software requirements are met, simply navigate to the root of the codebase and execute:

```
./runme.sh
```

This script will:

1) Download dependencies
2) Compile the framework
3) Generate synthetic datasets
4) Execute all experiments
5) Produce all figures and tables in `data/figures`

The entire experiment may take approximately 4–6 hours to complete, depending on the hardware configuration. The outputs include Figures 3, 5, 7, 8, 9, and Table 1. Absolute execution times, speedups, and other numerical values may vary depending on the hardware used. In Figure 9, `MW-Par-GPU-CAS` is omitted for the Solo case due to excessive execution time.