SBGT: Scaling Bayesian-based Group Testing for Disease Surveillance

1st Weicong Chen
Department of Computer and Data Sciences
Case Western Reserve University
Cleveland, USA
wxc326@case.edu

2nd Hao Qi
Department of Computer Science and Engineering
University of California Merced
Merced, USA
hqi6@ucmerced.edu

3rd Xiaoyi Lu
Department of Computer Science and Engineering
University of California Merced
Merced, USA
xiaoyi.lu@ucmerced.edu

4th Curtis Tatsuoka
Department of Medicine
University of Pittsburgh
Pittsburgh, USA
cut4@pitt.edu

***Abstract*—The COVID-19 pandemic underscored the necessity for disease surveillance using group testing. Novel Bayesian methods using lattice models were proposed, which offer substantial improvements in group testing efficiency by precisely quantifying uncertainty in diagnoses, acknowledging varying individual risk and dilution effects, and guiding optimally convergent sequential pooled test selections using a Bayesian Halving Algorithm. Computationally, however, Bayesian group testing poses considerable challenges as computational complexity grows exponentially with sample size. This can lead to shortcomings in reaching a desirable scale without practical limitations. We propose a new framework for scaling Bayesian group testing based on Spark: SBGT. We show that SBGT is lightning fast and highly scalable. In particular, SBGT is up to 376x, 1733x, and 1523x faster than the state-of-the-art framework in manipulating lattice models, performing test selections, and conducting statistical analyses, respectively, while achieving up to 97.9% scaling efficiency up to 4096 CPU cores. More importantly, SBGT fulfills our mission towards reaching applicable scale for guiding pooling decisions in wide-scale disease surveillance, and other large scale group testing applications.**

***Index Terms*—Group testing, Bayesian, Lattices, Spark, COVID-19**

I. Introduction

Since the outbreak of COVID-19, there has been a renewed interest in group testing due to the dire need for widespread testing [1]–[4]. Large-scale and repeated testing for COVID-19 and future pandemics will play an essential role in disease surveillance. Efficiencies of scale in testing are needed, and group testing can provide massive gains.

The original group testing formulation was proposed by Dorfman [5], who suggested to first group test all N subjects: if the test result is negative, all subjects are classified as negative; otherwise, each subject will be tested individually. Dorfman showed that group testing could lead to considerable savings in testing relative to individual testing. However, it should be noted that this approach does not consider testing errors or prior risks in classification, which can lead to high rates of false positives and false negatives [6]. Since the outbreak of COVID-19, it has been of interest to develop one-stage group testing methods [7], [8], for instance which rely on erasure coding (EC). These group testing approaches suggested for COVID-19 are only effective for the lower-risk scenarios with meager prevalence rates ($< 1.3\%$).

Recently, we proposed a Bayesian approach [6] with an associated high-performance Bayesian group testing framework named HiBGT [9], aiming to address issues in previous group testing designs. Our approach achieves accurate, flexible, and efficient group testing by leveraging Bayesian classification on lattice models [6], [10], where Bayesian methods and lattice representations for classification seamlessly allow for the incorporation of prior risk information and the updating of classification uncertainty in light of testing error. As illustrated in Figure 1, the workflow of Bayesian group testing is described as follows: the possible diagnostic outcomes of subjects can be represented using a *lattice model* (see §III). Acknowledging varying local prevalence and individual risk levels are done in a Bayesian manner through prior probability specifications. By iteratively performing test selection with a rule that has attractive statistical properties, the *Bayesian Halving Algorithm (BHA)* (see §IV), and updating posterior probability distributions on a lattice model, short sequences of pooled test selections can be generated, and tree-based *statistical analysis* (see §V) of classification performance can be conducted.

A. Motivation

Preliminary studies conducted by HiBGT have shown that Bayesian group testing can achieve outstanding statistical performance under varied individual risk levels while considering testing errors, e.g., it constantly reaches over 99.5% correctness in identifying positive subjects with less than 0.1% false

This work was supported in part by the NSF research grant DRL #1561716 and CCF #2132049.

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00099

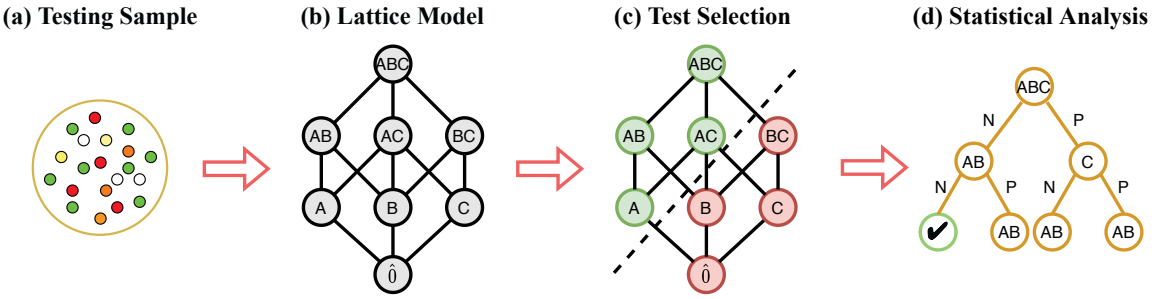


Fig. 1: Overview of Bayesian Group Testing Workflow

positive/negative across different prevalence and individual risk levels. Importantly, results show that it can help save more than 7x the number of expected tests compared to individual testing when $N = 20$. However, as illustrated in Table I, HiBGT suffers from significant computational and scaling challenges, i.e., constructing lattice models, BHA, and tree-based statistical analysis will grow exponentially in complexity as *the number of subjects* (N) increases. When $N = 30$, which is the arguable upper limit for practical COVID-19 group testing application [1], [11] such that dilution effects are still manageable relative to pooled test accuracy, a lattice model will comprise 2^{30} (over 1 billion) states, each corresponding to a unique test selection. To select a next-stage test selection, BHA iterates and computes through these states over 2^{60} (1.15×10^{18}) times. Moreover, a tree can potentially spawn millions of lattice models and test selections along branches when a statistical analysis is conducted.

TABLE I: Illustration of computational challenges in scaling up test selections using HiBGT and comparison between our proposed **SBGT**. Columns 2 and 3 are the number of states in a lattice model and the number of computation operations needed for a test selection. Columns 4 to 6 are time consumption for performing one test selection. Cells highlighted in blue are projected execution times using HiBGT.

N	# States	# Comp. Ops	HiBGT (1k Cores)	SBGT (1k Cores)	SBGT (4k Cores)
25	33M	1.1×10^{15}	4 Hours	9 sec	8 sec
26	67M	4.5×10^{15}	16 Hours	14 sec	12 sec
27	134M	1.8×10^{16}	64 Hours	42 sec	19 sec
28	268M	7.2×10^{16}	11 Days	3 min	49 sec
29	537M	2.9×10^{17}	44 Days	11 min	4 min
30	1.07B	1.1×10^{18}	6 Months	50 min	13 min

To alleviate high complexities, HiBGT proposed several optimization algorithms from the biostatistical perspective, such as shrinking the size of the lattice model when reaching partial classification and reducing the complexity of statistical analysis by sacrificing some accuracy or simplifying prior assumptions. It also leverages parallel computing at multiple levels, i.e., at BHA or statistical analysis. Despite spending hours using nearly 1,000 CPU cores, these optimization efforts only reach scales of $N = 25$ for a test selection or $N = 20$ for an edge-case/low-accuracy statistical analysis. Compared to the target upper limit ($N = 30$), the current computation scale falls short by over three magnitudes (2^{10} based on the asymptotic complexity of BHA, $N = 25$ versus $N = 30$). Additionally, the need for rapid feedback of results from group testing in dealing with infectious diseases necessitates short computation times for finding test selections, ideally within a few minutes. It thus raises tremendous challenges to push Bayesian group testing to such an unprecedented scale.

B. Contribution

In this paper, we propose a Scalable Bayesian Group Testing framework for disease surveillance, namely SBGT. Importantly, SBGT carries our mission for pushing Bayesian group testing to a broadly applicable scale, which drives us to pursue innovative designs and implementations using state-of-the-art HPC and big data technologies. We meticulously explore the design spectrum of the three major components in Bayesian group testing, including 1) modeling, i.e., how to construct and manipulate the lattice model using high-performance abstractions and operations; 2) test selection, i.e., how to optimize and parallelize BHA by acknowledging its mathematical reasoning; and 3) tree-based statistical analysis, i.e., how to design efficient parallel tree construction schemes and exploit Bayesian probabilities for acceleration.

In SBGT, we thoroughly redesign each major component of Bayesian group testing compared to HiBGT: 1) we propose an innovative bit-based abstraction for the lattice model to improve its performance dramatically; 2) we propose an optimized Bayesian Halving Algorithm (Op-BHA) by leveraging elegant order-theoretic properties [10] to significantly reduces its computational complexity; 3) we propose two unique parallel tree-construction schemes on top of the existing scheme through detailed performance profiling, which significantly streamline the statistical analysis process; and 4) we implement various data parallelization techniques and load balance optimizations in Spark runtime to improve SBGT's performance and scalability.

We systematically evaluate our artifacts on two HPC clusters (i.e., SDSC Expanse and TACC Stampede2) with up to 4096 CPU cores. Based on the evaluation, we show that SBGT is up to 376x, 1733x, and 1523x faster than HiBGT in lattice models, test selections, and statistical analyses, respectively

(SBGT can be even faster at a larger scale, but the existing work has failed to scale due to slowness). As shown in Table I, SBGT is also fast enough to guide test selections at the full applicable scale for Bayesian group testing ($N = 30$), averaging only 13 minutes across various individual risk levels. Last but not least, SBGT is near-linearly scalable; we achieve up to 97.9% scaling efficiency up to 4096 CPU cores. To the best of our knowledge, this paper is the first work to push Bayesian group testing to this unprecedented scale.

II. Overview of SBGT

As illustrated in Figure 2, the architecture of SBGT is arranged as a top-down layout with three layers. Each layer contains some modules (large boxes), and each module is labeled by its designs and implementations as feature boxes (small colored boxes). Note that features colored in red are proposed in this paper, and features colored in blue are adopted from HiBGT

Top layer, the top layer of SBGT consists of two modules: the lattice model (§III) and BHA (§IV). The lattice model is essential in providing micro-functionalities for Bayesian group testing tasks. The current implementation of the lattice model uses hashmap as the core abstraction. In SBGT, we propose *bit-based lattice model*. It leverages an innovative binary-encoded abstraction (§III-A) to formulate the lattice model into a self-indexed array and allows for efficient construction/manipulation using bitwise operations (§III-B). The test selection functionality is established by performing BHA over the lattice model. In SBGT, we propose optimizing BHA, named *Op-BHA* (IV-A), which significantly reduces BHA's computational complexity.

Middle Layer, the middle layer of SBGT is tree-based statistical analysis (§V). It relies on the intricate interplay between the top-layer modules, i.e., lattice model and BHA, to construct trees. HiBGT proposed the multi-tree scheme (§V-A) with several complexity reduction techniques based on the statistical reasoning of Bayesian group testing, i.e., lattice shrinking, the symmetry approach, and the accuracy trade-off approach. We gain insights from these designs and propose two new schemes named *the single-tree scheme* (§V-B) and *the fusion-tree scheme* (§V-C), which are proven significantly more performant in large-scale statistical analyses.

Bottom layer, the bottom layer of SBGT includes our specifically designed data parallelisms and load-balancing techniques that are deeply integrated with Spark's runtime environment. These features allow top- and middle-layer modules to be parallelized efficiently in Spark. In addition to the existing data parallelization technique offered by HiBGT, i.e., inter-tree parallelism (§V-A), we propose two new parallelisms: intra-lattice parallelism (§IV-B) and inter-lattice parallelism (§V-B), to effectively accommodate other new features in SBGT. We further propose *workload alignment* and *multi-threading on Spark* to improve each parallelism's load balance.

SBGT is recognized as a high-performance and scalable framework that realizes the workflow of Bayesian group testing using Spark while offering coherent end-to-end designs, implementations, and optimizations throughout its offered feature.

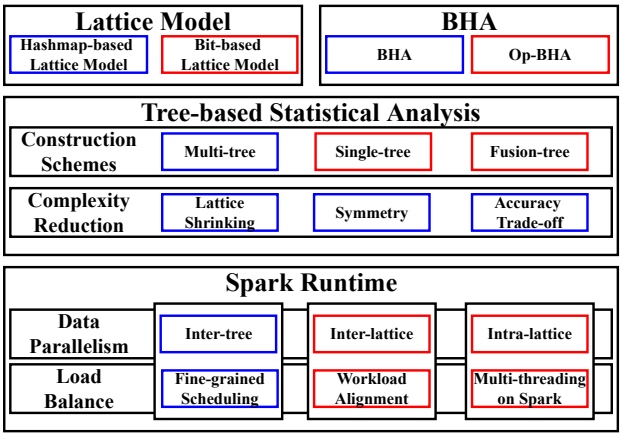


Fig. 2: Architecture overview of SBGT. Features colored in red are proposed in this paper, and features colored in blue are adopted from HiBGT.

III. Bit-based Lattice Model

Figure 1b presents the Hasse diagram of a lattice model with 3 subjects: A, B, and C. In Bayesian group testing, the classification objective is to identify the "true" profile that characterizes positive and negative disease status among the individuals being considered for pooling. Given N subjects, there are 2^N possible profiles of individual-level negative and positive diagnoses. A natural partial ordering arises among the states through inclusion, with states representing the subset of negative subjects. For example, the top element (state ABC) reflects that all subjects are negative, and the bottom element (state $\hat{0}$) is the state that all subjects are positive. Other states represent a mix of negatives and positives. Lattices are partially ordered sets that further assume for any two elements, and there is a unique greatest lower bound and unique least upper bound. This structure is key to understanding how statistical discrimination between states can occur in Bayesian group testing. More details on the lattice model are provided in [6], [10], [12]–[14].

In the Bayesian framework, for a given profile of the N subjects in terms of their disease status, to obtain its prior probability, we compute the product of the respective individual prior probabilities of their statuses in the profile. These values form a prior probability distribution across the possible states in the lattice model. After a test response is observed, e.g., we pool subjects A and C, and the returned test response is positive, state-level test response probability values for that observation are used in the Bayes rule to update prior probabilities to posterior ones. Response distributions depend on pool size and how many subjects in the pool are positive, given a state profile (the dilution effect). States with relatively high probabilities for the observed response will increase their posterior probability values. These posterior probability values

thus embody the combined empirical and prior evidence as to which state in the classification model is true.

A. Binary-encoded Abstraction

In HiBGT, we proposed using `HashMap` and key-value pairs `<state:String, prob:Double>`, e.g., `<"AC", 0.05>` as the core abstraction of lattice models. This design is natural as each state in a lattice model is associated with a prior/posterior probability and requires frequent access and modification. However, when a lattice model could comprise over 1 billion states ($N = 30$), this design becomes mediocre due to the innate deficiency of hashmap, such as memory overhead, slowness in computing hash functions, and performance instability due to hash collisions. Considering our mission is to improve the performance of Bayesian group testing by over three magnitudes, and the lattice model plays an essential role that every other component relies on, we conclude that optimizing HiBGT's hashmap-based design is unlikely to achieve our goal.

While exploring the lattice model's design space, we made tryouts in multiple data structures, such as Trie, segment tree, and directed acyclic graph (DAG). However, each has had its known limitation, and none could capture a natural abstraction for the lattice model. Therefore, our tryouts only led to either slight improvement or even worse performance than the hashmap-based design in the current study.

Motivated by the similar partially ordered structure between a lattice model and a power set, we propose a *binary-encoded abstraction* for representing lattice models. This abstraction inherently formulates a bijection between the set of states and a power set, allowing efficient *self-indexing* of each state for accessing its associated prior/posterior probability using binary numbers. The following example describes this process:

Ex III.1. Recall the lattice model from Figure 1b. The encoding function uses 1 bit to acknowledge the presence of each subject such that each state is encoded into a binary number with 3 bits, e.g., $AB \rightarrow \{110\}$, $BC \rightarrow \{011\}$. All states hence are subsequently mapped into the power set of $2^{\{111\}} = \{\{000\}, \{001\}, \{010\}, \{011\}, \{100\}, \{101\}, \{110\}, \{111\}\}$, formulating a *bijection* with the lattice model. As visualized in Figure 3, by constructing an array of size 2^N, $N = 3$, we can store and access the corresponding posterior probability of each state by translating its binary encoding to decimal, e.g., AC is encoded as $\{101\}$ and translated to decimal number 5 (incidentally, this translation is only necessary by human, but not for a computer, which makes the binary-encoded abstraction even more appealing), its prior/posterior probability is hence stored and can be accessed at index 5 of the array.

B. Bitwise Operations

The binary-indexed abstraction of lattice models promotes high-performance and concise bitwise operations to manipulate lattice models. For better illustration, we first define a key concept from order theory named *up-set* as follows:

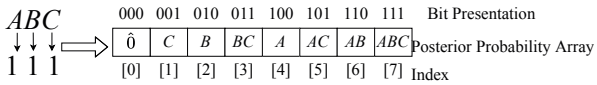


Fig. 3: Illustration of the bijection between a lattice model and a power set through binary-encoding abstraction, allowing states to be self-indexed in an array.

Definition III.1 (Up-set of a state in a lattice). For a state s in the lattice, the up-set $\uparrow s$ is the subset of states within the lattice that is at least as great as (i.e., contain) s.

The computation of up-set is critical in conducting test selections using BHA, which will be discussed in more detail in §IV. In the hashmap-based abstraction, determine if state s_1 is an up-set of state s_2 requires calling s_1`.containsAll(`s_2`)`, which takes $O(\deg(s_2) \cdot \deg(s_2))$. Whereas in the binary-indexed abstraction, since each state is self-indexed, a simple bitwise AND: s_1 `&` s_2 `==` s_2, will suffice the checking, which is theoretically faster than arithmetic addition.

Another benefit of using bitwise operations on binary-indexed abstraction is that it allows for more logically elegant algorithms for many performance-critical tasks. We illustrate one such task in Example III.2. Its usage will be further described in §IV-A.

Ex III.2. Suppose we want to find $\uparrow C$, C is 001 (decimal: 1) in binary-indexed abstraction. The absent subjects are A (binary: 100) and B (binary: 010). This can be achieved by actively generating the power sets of both absent subjects in binary, which are $2^{\{\{100\},\{010\}\}} = \{\{000\}, \{100\}\}, \{\{010\}, \{110\}\}$, respectively. Summed by the binary encoding of C ($\{\{001\}$), we get $\{\{001\}, \{101\}, \{011\}, \{111\}\}$, which is equivalent to $\{C, AC, BC, ABC\}$, or $\uparrow C$.

Subroutine `up_set()` in Algorithm 1 depicts the algorithmic procedure of this task, where the binary encoding of each element in $\uparrow s$ is generated in line 35. The complexity of this subroutine is $n \cdot 2^n$ for s, where $n = N -$ cardinality(s). This subroutine generally has a much lower complexity than the exhaustive traversal approach used in HiBGT, which takes $O(2^N)$.

We summarize the binary-encoded abstraction and bitwise operations as the bit-based lattice model. Note that due to its higher abstraction in biostatistical reasoning and lower perceptibility from complex bitwise operations, the bit-based lattice model is much less intuitive, which burdens programming efforts. We also face significant challenges in seamlessly integrating the bit-based lattice model into other new and existing designs. For example, HiBGT proposes the lattice shrinking technique that can sequentially eliminate any classified subjects from the lattice model after updating its posterior probability distribution, drastically reducing model complexity, e.g., each classified subject help reduce the complexity of

the lattice model by half. However, using the binary-indexed abstraction becomes troublesome, as it does not "localize" a state naturally. To illustrate, ABC is represented as $\{111\}$. If B is eliminated from the lattice model, then AC becomes binary $\{11\}$, and it is now impossible to identify if $\{11\}$ represents AB or BC or AC in different contexts. To address this issue, an integer is used as a state locality look-up flag: e.g., when B is classified, we change the flag from 0 to 2, indicating binary $\{010\}$ (or B) is classified, which helps identify $\{11\}$ as AC. Despite higher design complexity, the bit-based lattice model significantly improves the all-around performance for Bayesian group testing. At $N = 25$, we see up to 376x faster manipulation speed and 96.2% less memory overhead by using the bit-based lattice model than the HiBGT's hashmap-based lattice model. A detailed performance evaluation comparing the two is depicted in §VI-A.

IV. Optimizing Bayesian Halving Algorithm

BHA is the optimal strategy for guiding test selections using lattice models. To understand BHA, recall Definition III.1, a practical interpretation of an up-set in a lattice model under group testing is as follows: $\uparrow ABC$ is all the states for which a pool of A, B, and C would contain only negative subjects, $\uparrow A$ is all the states such that A is negative, etc. Conversely, the complement of the up-set of $\uparrow ABC^c = \{AB, AC, BC, A, B, C, \hat{0}\}$ represents states for which a pooled test ABC would contain at least one positive sample. Similarly, $\uparrow A^c = \{BC, B, C, \hat{0}\}$ is comprised of the states for which an individual test of a sample from subject A is likely to be positive, etc. Therefore, the up-set of a state and its complement generate a partition of the classification states based on whether the corresponding pooled test would contain all negatives or some positives.

The key objective of BHA is to systematically partition the lattice model based on the current posterior probability distribution on the lattice. By partitioning as close to half as possible, the sum of posterior probabilities, known as *posterior probability mass* (or $m(s)$ for a state s), will increase in one of the two partitions and decrease in the other. This property systematically implies that the posterior probability mass will quickly accumulate to a single state. The following lemma gives the correctness and optimality of BHA.

Lemma 1 (Correctness and optimality of BHA)**.** *BHA attains the optimal rates of convergence of the Bayesian posterior probability for correctly classifying the true state to 1 almost surely, regardless of the true state, and even under strong dilution effects.*

Proof. BHA's correctness and optimality are given in [6]. ■

Example IV.1 below depicts the process of BHA on a lattice model.

Ex IV.1. Figure 1c visualizes a partition through state A on a 3-subject lattice model. Green states represent $\uparrow A$ and red states represent $\uparrow A^c$. If $m(A)$ (the sum of posterior probabilities of green states) is the closest to 0.5 among any other states, then A is the desired test selection.

BHA finds the desired test selection by traversing and partitioning across all states. Depending on how $m(s)$ is calculated, i.e., exhaustive traversal used in HiBGT or our proposed active generation discussed in Example III.2, the overall complexity of BHA can reach $O(2^{2N})$ or $\sum_{n=0}^{N} \binom{N}{n} \cdot n2^n$. Quantitatively, for $N = 30$, BHA needs to iterate and compute through over 1 billion states (2^{30}) for over 1.15×10^{18} (2^{60}) times using exhaustive traversal, or 3.7×10^{15} times using active generation. The high complexity and the tremendous computations for large N motivate us to further explore the design spectrum of BHA beyond mathematical reasoning.

A. Op-BHA Algorithm

We give the definition of *down-set* to assist the illustration of our proposed Op-BHA algorithm.

Definition IV.1 (Down-set of a state in a lattice)**.** For a state s in the lattice, the down-set $\downarrow s$ is the subset of states within the lattice that is at most as great as (i.e., are contained by) s.

A critical observation between up-set/down-set and posterior probability mass is summarized as the following theorem.

Theorem 1 (Partial ordering of posterior probability masses in a lattice)**.** *For states a and b in a lattice model, $m(a) < m(b)$ if $a \in (\uparrow b \setminus b)$, and $m(a) > m(b)$ if $a \in (\downarrow b \setminus b)$. $m(a)$ and $m(b)$ are the posterior probability masses of a and b.*

Proof. Suppose $a \in (\uparrow b \setminus b)$ and let $\{a_1, a_2, \ldots, a_n\} = \uparrow a$. By Definition III.1, $\forall a_i$, $i = 1 \ldots n$, a_i is at least as great as a. Since a is also at least as great as b and $a \neq b$, a is greater than b. It then follows $\uparrow a \subset \uparrow b$. Since Bayesian probabilities are always positive, we have $m(a) < m(b)$. The other direction, i.e., $m(a) < m(b)$ does not necessarily mean $a \in (\uparrow b \setminus b)$ is proved by a counterexample using the 3-subject lattice model shown in Figure 1b: let subject A's prior probability be 0.1 and B's prior probability be 0.05, then initially, $m(A) = .9$ and $m(B) = 0.95$. $m(A) < m(B)$ but $A \notin \uparrow B$. The proof for the down-set follows similarly and is omitted. ■

Theorem 1 reveals the opportunity of reducing the number of states necessitated by BHA. Specifically, when traversing each state s in a lattice model, if $m(s) < 0.5$, it is safe to skip calculating posterior masses for all elements in $\uparrow s$; and if $m(s) > 0.5$, we can otherwise skip calculating posterior masses for all elements in $\downarrow s$. This concludes the key idea of Op-BHA, and we prove its correctness below.

Theorem 2 (Correctness of Op-BHA)**.** *Op-BHA is equivalent to BHA and attains the optimal correctness rates.*

Proof. The proof is obtained by contradiction. Suppose BHA finds the correct test selection s_c for a lattice model of size N, it follows that $min\{|m(s_1) - 0.5|, \ldots, |m(s_c) - 0.5|, \ldots, |m(s_{2^N}) - 0.5|\} = |m(s_c) - 0.5|$. Assuming Op-BHA finds a different test selection s_e, such that $s_e \neq s_c$. It implies that s_c is skipped by Op-BHA. If $m(s_c) < 0.5$,

then by Op-BHA, $m(s_e) < 0.5$ and $s_c \in \uparrow s_e$. Combined with Theorem 1, we have $m(s_c) < m(s_d) < 0.5$ and hence $|m(s_e) - 0.5| < |m(s_c) - 0.5|$, which suggests s_e is the correct test selection, contradicting the fact. The proof for $m(s_c) > 0.5$ is similar and is omitted. It then follows by Lemma 1 and proves the theorem. ■

An example of potential computation savings of Op-BHA is illustrated as follows.

Ex IV.2. Let $N = 30$ and every subject has a prior risk level of 0.02. Suppose Op-BHA first partitions through the topmost state t (the state that identifies all subjects are negative). Since $m(t) \approx 0.545 > 0.5$, it will skip evaluating $\downarrow t \setminus t$, which comprises all remaining states in the lattice model. We thus find the correct test selection by evaluating only one state.

Algorithm 1 describes the three-phase process for performing a test selection using Op-BHA over a bit-based lattice model. Phase 1 is the one-time prerequisite procedure of constructing a lattice model, and Phase 3 is the recurring postrequisite procedure of updating the lattice model's posterior probability distribution before performing the next Op-BHA. Lines 10 and 13 - 16 comprise BHA's primary computational tasks and are shared by Op-BHA. The rest of Phase 2 represents exclusive algorithmic designs for Op-BHA, and here we focus on them in detail. 1) In line 7, we construct a checklist for tracking each state's traversal status, i.e., skippable or unevaluated. The list is implemented using `BitSet`, which internally uses an `Int64` array, where each integer can store statuses for 64 states. We choose over a `Boolean` array as it only consumes 1/8 the memory. 2) In line 10, we propose a fixed traversal strategy aiming to maximize the number of skippable states. Specifically, the traversal starts from the topmost and bottommost states in the lattice model as they can potentially skip all remaining states, and gradually traverse to the middle states, i.e., states with cardinalities $N/2$ assuming N is even, as each middle state can only reduce up to $2^{N/2}$ states. 3) In line 20, the implementation of `down_set()` subroutine follows similarly with `up_set()` subroutine with minor changes and is hence omitted. Its algorithmic complexity shares the same formula as `up_set()` discussed in Example III.2 but differs in that n = cardinality(s). 4) To better illustrate `Op-BHA()`, each subroutine depicted in Phase 2 uses $O(2^n)$ extra space for storing the up-set/down-set, which is eliminated in the real implementation as each generated state can be consumed immediately. The implementation also adopts various optimizations to facilitate performance, such as reordering nested loops to alleviate I/O bottlenecks and reusing hot data to improve cache hits.

Complexity analysis, `Op-BHA()` in Algorithm 1 clearly illustrates the dynamic computational complexity at each loop, i.e., $O(0)$, $O(n{\cdot}2^n)$ or $O(n{\cdot}2^n{+}(N-n){\cdot}2^{N-n})$. Besides N, the aggregated complexity also depends on the posterior probability distribution, which is determined by multiple factors, such as the prior probability information, the ground

Algorithm 1: Test Selection Using Op-BHA

```
   Input: prior_prob[N]
   Output: selection
1  prob_dist[2^N]                                   ▷ probability distribution
   Phase 1 - Construct Lattice Model
2  Procedure BUILD_MODEL(prior_prob, prob_dist)
3      for i=0; i < 2^N; ++i do
4          for j=0; j < N; ++j do
5              i & 1 << j == 0 ?
                 prob_dist[i]* = prior_prob[j] :
                 prob_dist[i]* = 1 − prior_prob[j]
   Phase 2 - Op-BHA
6  Procedure Op-BHA(prob_dist)
7      chklist[2^N]                                 ▷ track skippable states
8      min ← ∞                                      ▷ track prob mass closest to 0.5
9      n ← N − cardinality(s)
10     for state s from topmost & bottommost to middle do
11         if chklist.get(s) then
12             continue                             ▷ state evaluated / skipped
13         mass = calc_mass(s, prob_dist)           ▷ O(n·2^n)
14         if |mass − 0.5| < min then
15             min ← mass
16             selection ← i
17         if mass < 0.5 then
18             chklist.set(up_set(s))               ▷ O(n·2^n)
19         else if mass > 0.5 then
20             chklist.set(down_set(i))             ▷ O((N-n)·2^{N−n})
21     return selection
22 Subroutine calc_state_mass(s, prob_dist)
23     mass ← 0, n ← N − cardinality(s)
24     for i = 0; i < 2^n; ++i do
25         mass += prob_dist[up_set[i]]
26     return mass
27 Subroutine up_set(s)
28     n ← N − cardinality(s), up_set[2^n]
29     absent_subj[n]                               ▷ correct values filled
30     for i = 0; i < 2^n; ++i do
31         ind ← s
32         for j = 0; j < n; ++j do
33             if i & (1 << j) then
34                 ind += absent_subj[j]
35         up_set[i] = prob_dist[ind]               ▷ ind ∈ ↑s
36     return up_set
   Phase 3 - Update & Shrink Lattice Model
37 Procedure UPDATE_SHRINK_MODEL(selection, response, prob_dist)
38     for i=0; i < 2^N; ++i do
39         if response == negative then
40             prob_dist[i] *= dilution(i, selection, response)
41         else if response == positive then
42             prob_dist[i] *= 1 − dilution(i, selection, response)
43     normalize(prob_dist[i])                      ▷ normalize distribution
44     shrink(prob_dist[i])                         ▷ if subject(s) is classified
```

truth (true state of subjects), test responses, and dilution effects. The enormous degrees of freedom in estimating the posterior probability distribution have prohibited a general-case complexity of Op-BHA from being deduced. However, Op-BHA's complexity is at most as complex as BHA, i.e., $\sum_{n=0}^{N} \binom{N}{n} \cdot n \cdot 2^n \approx O(2^{2N})$, which only occurs in extremely rare cases where almost every state has to be evaluated. On the other hand, from Example IV.2, which demonstrates the maximum possible state reduction, we can see Op-BHA's complexity is bounded below by $\Omega(2^N)$.

B. Parallelzing Op-BHA

Besides reducing algorithmic complexity, parallelizing Op-BHA provides another promising passage toward speeding up test selections. We propose parallelizing Op-BHA using a multi-staged MapReduce programming model based on Spark, namely *intra-lattice parallelism*. Note that the design of intra-lattice parallelism is not limited to a data parallelism model as we use. It can be easily ported to other parallel programming models, such as message passing (MPI) or partitioned global address space (PGAS). Figure 4 describes the multi-staged MapReduce model: the master first broadcasts the posterior probability distribution of the lattice model to workers. Next, based on the existing checklist, it generates a list of yet-to-be-evaluated states, known as *candidate states*. The master then dispatches a chunk of candidate states to workers using `map()` so that each worker can perform lines 11-20 of Algorithm 1 on given candidate states. Note that when each worker finishes computation, not only does it return to the master a temporarily desired *test selection*, but also an updated checklist indicating which states can be further omitted. These returned results are then processed using `reduce()`, i.e., the returned test selections are reduced by examining lines 14-16 of Algorithm 1, and the checklist is reduced by bitwise OR. This concludes the MapReduce stage. If the master detects more candidate states, i.e., the checklist's cardinality does not equal 2^N, it starts another stage of the procedure until no candidate states are available.

Differed from the serial algorithm described in Procedure `Op-BHA()` of Algorithm 1, where the checklist is updated on the fly in a shared memory view, the checklist in intra-lattice parallelism is only updated after evaluating multiple states' posterior probability masses after each stage, meaning duplicated computation is inevitable. Therefore, the chunk size C will play an essential role in the overall performance of intra-lattice parallelism, where a smaller C can result in faster evaluation and more effective state reduction per stage. However, it will potentially increase the number of total stages, hence suffering from more scheduling and data overheads. On the other hand, a larger C may reduce savings from fewer skipped states and a more severe load imbalance because executors can skip different amounts of candidate states. Therefore, there is no one-size-fits-all choice for C as there is no fixed optimal design for varied posterior probability distributions. Our implementation uses $C = 2^{\frac{N}{2}+8}$ based on heuristics to reach a "sweet point" between balancing loads and alleviating various overheads while achieving significant workload reduction for Op-BHA.

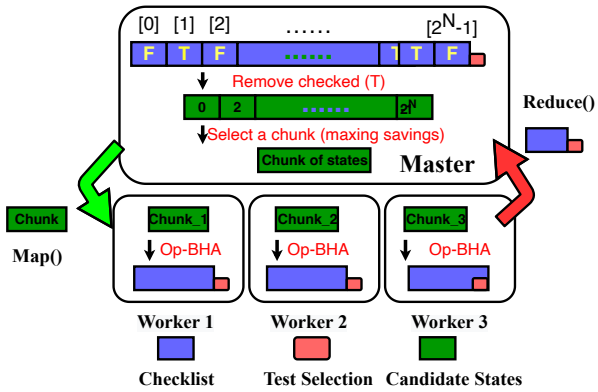


Fig. 4: Intra-lattice parallelism using multi-staged MapReduce. The master dispatches tasks at each execution stage as a chunk of candidate states to executors after collecting and processing the checklist returned from the last stage.

Multithreading on Spark, we propose shared memory inside each worker to improve the load balance of intra-lattice parallelism. We use a customized fork-join pool to leverage the famous work-stealing algorithm [15]. This algorithm allows a thread to "steal" unfinished tasks from other threads, which is ideal for executing algorithms with dynamic workloads like Op-BHA. Additionally, multithreading on Spark also reduces the number of concurrent tasks while maintaining high degrees of parallelism. It also effectively decreases data communication overheads to improve the scaling efficiency for performing test selections that guide optimal pooling strategies.

Complexity analysis, suppose p processes are used for intra-lattice parallelism, the complexity of intra-lattice parallelism is bounded above by $O(2^{2N}/p)$, as it assumes almost no states are reduced. the lower bound of intra-lattice parallelism remains at $\Omega(2^N)$, which reflects the scenario illustrated in Example IV.2, there is only one process which holds the topmost state is performing effective state reduction, while the rest $p-1$ processes are in fact performing duplicated computation.

V. Parallel Tree-based Statistical Analysis

The theoretical efficiency and effectiveness of BHA over lattice models are evidenced by statistical analyses under varied prevalence and individual risk levels. Statistical analysis is conducted by exhaustively constructing all possible test sequences in a tree style up to a certain depth (referred to as the number of *stages*) based on generating test selections using BHA and updating the posterior probability distributions along branches. Classification performance statistics for Bayesian-based group testing methods can be evaluated on this tree, including the rate of decisive classification, as reflected by stringent posterior error threshold(s) being satisfied, false positives/negative rates, and the expected number of tests.

The procedure of constructing a tree is described as follows: BHA selects one test at the first stage, which can return two possible results: negative or positive. For the negative branch, after the root lattice model updates its posterior probability distribution based on the negative response, a second-stage BHA can be performed. This holds similarly for the positive branch at the first stage. Therefore, this tree can be spawned like a fully-grown binary tree, and each branch formulates a test sequence. A typical statistical analysis can reach 24 stages comprising up to 2^{24} (over 16 million) branches.

A. Existing Approach: The Multi-tree Scheme

HiBGT proposed a tree construction approach named the multi-tree scheme, which divides the big tree into multiple small trees that each correspond to one true state. The statistical reasoning behind this scheme is that for a given true state, most of the branches in the tree are statistically non-significant and can hence be pruned; we name this technique as *branch pruning*. For example, the branch {<ABC, Positive>, <A, Positive>, <C, Positive>, <B, Positive>} is practically impossible given the true state ABC (subject A, B, and C are all negative), whereas it is statistically sound if the true state is $\hat{0}$ (subjects A, B, and C are all positive). Therefore, each tree will be drastically smaller.

The benefit of the multi-tree scheme is the easiness of parallel construction. Figure 5a depicts the parallel architecture for the multi-tree scheme with four nodes (one master and three workers). It uses *inter-tree parallelism*, where each worker can independently perform tree constructions. Since each tree can expand differently, this technique also adopts a fine-grained task scheduling protocol to achieve load balance, leading to a claimed scaling efficiency of up to 99.5% using up to 896 cores.

Despite being easy to parallelize, we find three drawbacks in the multi-tree schemes compared to constructing only one big tree. *First*, the multi-tree scheme has a much higher asymptotic complexity because the number of required trees will grow exponentially as N increases, whereas the size of the single tree is bounded and independent of N, e.g., a 24-stage tree contains up to 2^{24} branches no matter the N. *Second*, the multi-tree scheme is inefficient, as trees can share the same partial branches that resort to duplicated computation. *Third*, the multi-tree scheme does not unleash the full optimization potential of the lattice shrinking technique (see §III-B). The multi-tree scheme has to start each tree from the initial, full-size lattice model. As a comparison, the initial lattice model is only constructed once in the big tree and will be quickly shrunk along branches. Therefore, the big tree typically comprises fewer full-size and large-size lattice models.

To alleviate the first drawback, we proposed two mathematical optimizations in HiBGT to reduce the number of constructed trees: 1) the symmetry property of the lattice model can reduce the number of required trees from 2^N to $N + 1$ if individual risk levels are homogeneous, namely *the symmetry approach*; and 2) by discarding true states with low prior probabilities up to certain percentage, e.g., 0.1% total, the multi-tree scheme can retain 99.9% statistical accuracy while significantly reduce the number of true states that requires evaluation, namely *the accuracy trade-off approach*. Regardless, these two approaches trade off generality or accuracy vs. performance, which are not ideal for accuracy-critical scenarios. We identified the second and third drawbacks as the innate deficiency of the multi-tree scheme, hence not addressing them in HiBGT.

B. The Single-tree Scheme

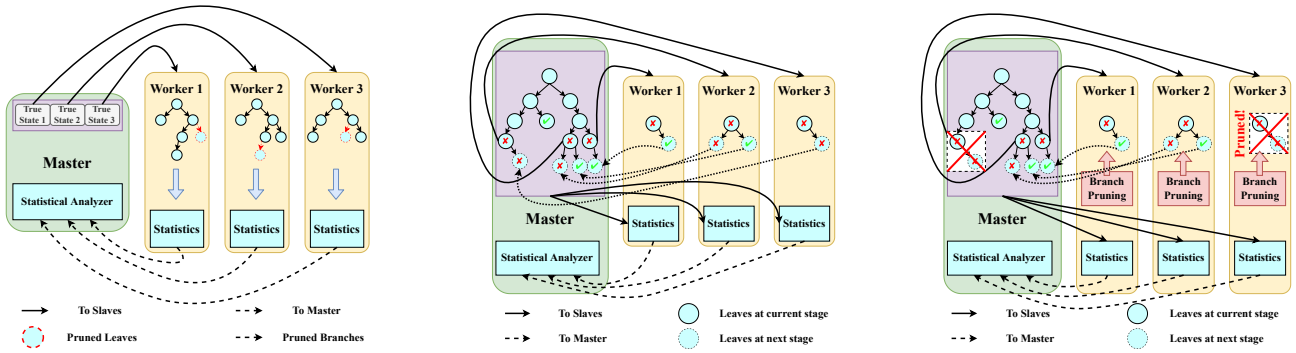
Established on the multi-tree scheme's drawbacks, we propose constructing the big tree in parallel, namely *the single-tree scheme*. Figure 5b depicts the single-tree scheme's parallel architecture using four nodes. Unlike the multi-tree scheme, which uses depth-first construction for each tree, the single-tree scheme adopts a breadth-first approach to construct the big tree stage by stage. This allows each worker to perform the test selections (Line 6 to 29 of Algorithm 1) for a portion of lattice models at the current stage by `map()` all lattice models as a resilient distributed dataset (RDD), namely *inter-lattice parallelism*. After test selections are complete, the master use `collect()` call to gather every updated lattice model and append it to the corresponding location of the tree. The master then schedules tasks for the next-stage tree construction until reaching the maximum stage.

The issue with exclusively using inter-lattice parallelism for the single-tree scheme is that it leads to insufficient degrees of parallelism during the beginning stages, e.g., even the seventh stage only contains up to 128 lattice models. We propose two solutions to solve this issue: 1) we use intra-lattice parallelism (see §IV-B) to achieve full resource utilization; and 2) we adopt multithreading on Spark into inter-lattice parallelism, such that each worker node computes Algorithm 1 instead of Spark's default one CPU core per lattice model. Another issue with inter-lattice parallelism is that when cooperated with lattice shrinking, it can lead to a skewed computation workload because lattice models can vary in size. We hence propose *task alignment*, which organizes the group of lattice models by size and divides it into subgroups; then, each subgroup is sequentially dispatched by the master as a standalone task to ensure load balance. When the number of lattice models in a subgroup does not suffice the degrees of parallelism (usually full-sized or large-sized lattice models), SBGT uses intra-lattice parallelism for this sub-task.

Our evaluation (see §VI-C) shows that the single-tree scheme is up to 515x faster than HiBGT's multi-tree scheme for a medium-scale Bayesian group testing ($N = 17$). Even applying the symmetry approach or the 99.9% approach, the single-tree scheme still outperforms the multi-tree scheme by up to 1.7x and 54.4x, respectively.

C. The Fusion-tree Scheme

Summarized from the above two tree construction schemes, we raise the question: *Can we leverage each scheme's advantages and alleviate its drawbacks by creating a deep fusion of the two?* The answer is yes. We propose a scheme named



(a) HiBGT's multi-tree scheme (b) The single-tree scheme (c) The fusion-tree scheme

Fig. 5: Parallel Tree Construction Schemes

the fusion-tree scheme. The key idea is that after the single-tree scheme finishes constructing every big tree stage, we perform the identical branch pruning process in the multi-tree scheme. As demonstrated in Figure 5c, the branch in Worker 3 is pruned, which reduces future computation workload and data communication overhead to the fusion-tree scheme. The statistical reasoning behind branch pruning in constructing a single tree is that some low-probability branches, such as {<A, Positive>, <A, Negative>, <A, Positive>, <A, Negative>, $\cdots$} are statistically meaningless to any true states, hence should be pruned immediately. From the computational perspective, pruning low-statistics lattice models at an early stage prevent lower-statistics lattice models from spawning in later stages, reducing the computation workload in the long run. Also, the lattice shrinking technique usually works poorly along such branches, which leads to computation on unnecessary large-sized or even full-sized lattice models. Last but not least, extending branches is computationally costly, which necessitates the construction of new lattice models and computations of test selections, while branch pruning is computationally efficient and as easy to parallelize as the multi-tree scheme.

Our evaluation (see §VI-C) shows that the fusion-tree scheme successfully improves statistical analysis speed on large-scale Bayesian group testing ($N = 25$) over the single-tree scheme by up to 7.7x, as it leverages advantages and avoids disadvantages from both the multi-tree scheme and the single-tree scheme. It also offers the best architectural optimization to computational efficiencies and memory footprint in our implementations, promoting SBGT to push the efficiency and performance of statistical analyses to the edge.

VI. Evaluation

We systematically evaluate the performance and scalability of SBGT and compare it with the state-of-the-art Bayesian group testing framework, i.e., HiBGT. The goal is to measure the performance benefits of the different techniques proposed in SBGT. In addition, we investigate whether we can achieve our mission of guiding real-world test selections for large-scale group testing ($N = 30$) in a rapid-testing scenario (within a few minutes) by leveraging state-of-the-art HPC technologies. **Experimental setup**, Table II details the clusters used for evaluating SBGT.

TABLE II: Specification of Two HPC Clusters

Cluster	SDSC Expanse	TACC Stampede2
Processor	AMD EPYC 7742	Intel Xeon Phi 7250
Clock Speed	2.25 GHz	1.4 GHz
No. of Cores	128	64
RAM (DDR)	256 GB	64 GB
Interconnect	InfiniBand-EDR (100 Gbps)	100 Gbps Omni-Path
Storage	NVMe PCIe SSD (1TB)	SATA-SSD (500GB)
Spark	Spark-3.2.1	Spark-3.2.1
Scale	32 nodes (**4096 cores**)	8 nodes (**512 cores**)

Prior risk scenarios, to systematically evaluate the performance of our proposed designs and implementations for SBGT, e.g., Op-BHA, different parallelisms, and parallel statistical analysis schemes, we propose a set of group testing scenarios to comprehensively imitate real-world group testing scenarios. As shown in Table III, we first define three individual risk levels: lower risk, medium risk, and higher risk, each with two different risk patterns, which reflect typical COVID-19 prevalence rates.

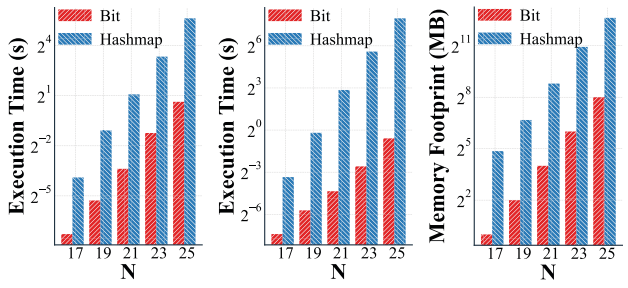
TABLE III: Specification for Prior Risk Scenarios

Name	**Risk Pattern**	**Risk Level**
0.02	All 0.02	Lower
0.1	All 0.1	Medium
0.15	All 0.15	Higher
0.2	All 0.2	Higher
1-mix	1 $\times$ 0.2, rest are 0.02	Lower
2-mix	2 $\times$ 0.2, rest are 0.02	Medium

A. Lattice Model

This experiment evaluates the performance of SBGT's bit-base lattice model and compares it with HiBGT's hashmap-based lattice model in three categories: construction speed, posterior probability distribution update speed, and lattice model memory footprint. Note that these performance statistics are independent of the prior probability distribution of the

pooled subjects. We hence fix the risk pattern to 0.02. Each category is evaluated with an increasing lattice size ranging from $N = 17$ to $N = 25$, with step size 2.



(a) Construct (b) Update (c) Memory

Fig. 6: Evaluation of SBGT's bit-based lattice models vs. HiBGT's hashmap-based lattice models. The execution time for constructing and updating a bit-based lattice model at $N = 25$ is 32x and 376x faster than a hashmap-based lattice model while using 96.2% less memory.

Construction, Figure 6a presents the comparison of the construction performance between the bit-based lattice model and the hashmap-based lattice model. The execution time for the bit-based lattice model are 0.006s, 0.025s, 0.096s, 0.425s, and 1.553s, for $N = 17$, 19, 21, 23, and 25, respectively. The execution time for the hashmap-based lattice models are 0.067s, 0.473s, 2.117s, 10.159s, and 49.804s, respectively. The bit-based lattice model is 11x to 32x faster than the hashmap-based lattice model in construction.

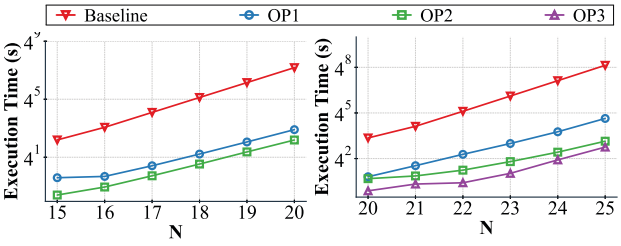
Update, Figure 6b presents the comparison of the update performance between the bit-based lattice model and the hashmap-based lattice model. The execution time for the bit-based lattice model are 0.006s, 0.019s, 0.049s, 0.166s, and 0.663s, for $N = 17$, 19, 21, 23, and 25, respectively. The execution time for the hashmap-based lattice model are 0.098s, 0.873s, 7.243s, 48.238s, and 248.955s, respectively. The bit-based lattice model is 5x to 376x faster than the hashmap-based lattice model in updating the probability distribution.

Memory, Figure 6c presents the comparison of memory footprints between the bit-based lattice model and the hashmap-based lattice model. We measure the memory footprint using `SizeEstimator` API provided by Spark. Note that the output reflects object sizes before serialization and is corrected to megabytes. A bit-based lattice model consumes 1MB, 4MB, 16MB, 64MB, and 256MB memory, for $N = 17$, 19, 21, 23, and 25, respectively, whereas a hashmap-based lattice model consumes 29MB, 102MB, 444MB, 1955MB, and 6318MB memory, respectively. The bit-based lattice model reduces memory overheads by 95.9% to 96.7%.

Overall, SBGT's bit-based lattice model dominates HiBGT's hashmap-based lattice model in both performance and memory efficiency. Notably, we see more pronounced disparities in both categories as N increases. As the fundamental component of Bayesian group testing, our innovative design and implementation of the lattice model establish a solid foundation in fulfilling our mission of large-scale test selections and statistical analyses.

B. Test Selection

This experiment evaluates the performance of SBGT's test selection designs and implementations, i.e., Op-BHA and intra-lattice parallelism, in both the single-thread and parallel environment. For the single-thread sub-experiment, we evaluate three incremental optimization designs labeled Baseline, OP1, and OP2, where Baseline is HiBGT, i.e., BHA on hashmap-based lattice models, OP1 is BHA on bit-based lattice models, and OP2 is Op-BHA on bit-based lattice models. For the parallel sub-experiment, we evaluate four incremental optimization designs labeled Baseline, OP1, OP2, and OP3, where Baseline, OP1, and OP2 share identical optimization designs with the single-thread sub-experiment, except that BHA/Op-BHA now runs in parallel. OP3 is parallel Op-BHA on bit-based lattice models plus using multi-threading on Spark. We use N ranges from 15 to 20 in the single-thread sub-experiment and 20 to 25 in the parallel sub-experiment. We use TACC Stampede2 Cluster with 512 CPU cores for evaluations. Note that for the performance statistics are assessed by the average execution time across six prior risk patterns at each N.



(a) Single Thread (b) Parallel

Fig. 7: Evaluation of SBGT's designs and implementations for test selections. (a) is single-thread, and (b) is using Spark. Baseline is HiBGT, i.e., BHA on hashmap-based lattice models, OP1 is BHA on bit-based lattice models, OP2 is Op-BHA on bit-based lattice models, and OP3 is implementing multi-threading on Spark on top of OP2. In (a), at $N = 20$, OP2 is 1023x faster than Baseline (SBGT vs. HiBGT both in serial computing). In (b), at $N = 25$, OP3 is 1733x faster than Baseline (SBGT vs. HiBGT both in parallel computing).

Single-thread, Figure 7a illustrates the test selection evaluation in a single-thread environment. Similar to the experiment with lattice models, we observe increasingly better performance in our proposed designs than in existing ones. We hence only report evaluation results at the largest N. At $N = 20$, the execution time for Baseline through OP2 are 20983s, 55.9s, and 20.5s, which can be translated to gradual speedups of 375x (bit-based lattice model vs. hashmap-based lattice model) and 2.7x (Op-BHA vs. BHA). Overall, OP2 is 1023x faster than Baseline (SBGT vs. HiBGT)

Parallel, Figure 7b illustrates the test selection evaluation in a parallel computing environment, i.e., Spark. At $N = 25$, the execution time from Baseline to OP3 are 78843s, 619s, 79.7s, and 45.5s, respectively, indicating gradual speedups of 127x (bit-based lattice model vs. hashmap-based lattice model), 7.8x (Op-BHA v.s BHA), and 1.7x (using multi-threading on Spark). Overall, OP3 is 1733x faster than Baseline (SBGT vs. HiBGT).

C. Tree Construction Schemes for Statistical Analysis

This experiment evaluates the performance of SBGT's parallel tree construction schemes, i.e., the single-tree and the fusion-tree schemes, with HiBGT's multi-tree schemes and its two optimization techniques, i.e., the symmetry approach and the accuracy trade-off approach. To ensure fairness, all schemes use full designs and implementations for lattice models and test selections, i.e., Op-BHA + intra-lattice parallelism + multi-threading on Spark over bit-based lattice models. We use SDSC Expanse Cluster with 512 CPU cores for this experiment. Note that due to slowness in existing schemes, we only perform the experiment using the 0.2 risk pattern, as it will result in constructing more complex trees than lower risk levels. We also use smaller N for existing schemes, ranging from 10 to 17. For our proposed schemes, we start with $N = 10$ and end with $N = 25$.

Existing multi-tree-based schemes, first, we evaluate performance statistics for the multi-tree scheme and its two optimization techniques. Note that for the accuracy trade-off approach, we choose to lose up to 0.1% accuracy; hence we denote it as the 99.9% accuracy approach. As shown in Figure 8, the multi-tree scheme is inefficient as the execution time increases 7x as N increases (85s, 513s, 3470s, and 22846s for $N = 14 - 17$). On the other hand, its optimization techniques effectively reduce computation workload. Using the symmetry approach, the execution time at $N = 17$ is 75s, as it reduces the number of true states from 2^{17} to 18. Using the 99.9% approach, the execution time at $N = 17$ is 2350s, reducing the number of true states to 2045.

The single-tree scheme and the fusion-tree scheme, next, we evaluate performance statistics for the single-tree and the fusion-tree schemes. As shown in Figure 8, at $N = 17$, the execution time of the single-tree scheme is 44s, whereas the fusion-tree scheme is 15s. Compared to the multi-tree scheme at the same scale, the single-tree scheme is 519x, and the fusion-tree scheme is 1523x faster. At $N = 25$, the execution time of the single-tree scheme is 4672s, whereas the fusion-tree scheme is 609s, which is 7.67x faster than the single-tree scheme. The single-tree scheme generates an average of 68462 classified and 52743 unclassified branches, while the fusion-tree scheme keeps 545 classified and only one unclassified branch by leveraging branch pruning (>99.5% pruning rate). The tree also stops expanding early at stage 23.

D. Scaling Test Selections

Compared to statistical analyses, which are usually conducted offline to provide insights into how efficient and

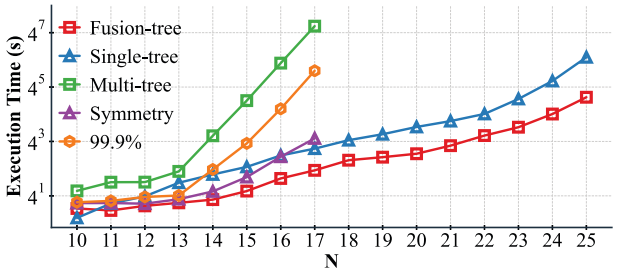


Fig. 8: Evaluation of SBGT's parallel tree construction schemes for statistical analysis. In (a), at $N = 17$, HiBGT's multi-tree scheme takes 22846s, while its symmetry and 99.9% accuracy techniques can reduce the execution time to 75s and 2350s, respectively. In (b), at $N = 17$, SBGT's fusion-tree scheme (15s) is 1523x faster than HiBGT's multi-tree scheme. At $N = 25$, the fusion-tree scheme (609s) is 7.7x faster than the single-tree scheme (4672s).

effective the Bayesian group testing can address real-world group testing scenarios, a test selection offers more practical usage in guiding a pooling decision and should be considered an online operation. Therefore, in this experiment, we evaluate the scalability of SBGT's test selections with our ultimate goal: $N = 30$, which represents a lattice model containing over 1 billion states. We use full designs and implementations for lattice models and test selections, i.e., Op-BHA + intra-lattice parallelism + multi-threading on Spark operating on bit-based lattice models. Test selections are performed across all six risk patterns to show systematic performance statistics. We conduct these experiments using SDSC Expanse with up to 4096 CPU cores, where we collect the average execution time at 512, 1024, 2048, and 4096 CPU cores. As illustrated in Figure 9, the scaling efficiencies at 4096 cores are 76.4%, 97.9%, 93%, 85.1%, 86.3%, and 69.3%, for risk pattern 0.02, 0.1, 0.15, 0.2, 1-mix, and 2-mix, respectively. The average scaling efficiency is 84.7%, and the average execution time for Op-BHA at this scale is only 13 minutes, which clearly demonstrates the capability of SBGT to guide pooling decisions for real-world, large-scale disease surveillance in a rapid testing fashion.

VII. Conclusion and Future Work

Bayesian group testing is appealing as it can recognize variable levels of individual risk and is accurate and efficient even when accounting for dilution effects. However, the required level of precision in modeling is computationally challenging, which impedes implementing Bayesian group testing at larger scales. We propose scaling Bayesian group testing for disease surveillance, namely SBGT, which allows for high-performance and scalable test selections and statistical analyses using the lattice model and with the statistically optimal Bayesian Halving Algorithm. Through systematic evaluations of various designs, implementations, and optimizations, we show that SBGT can speed up test selections and statistical analyses by up to 1733x and 1523x compared to HiBGT while

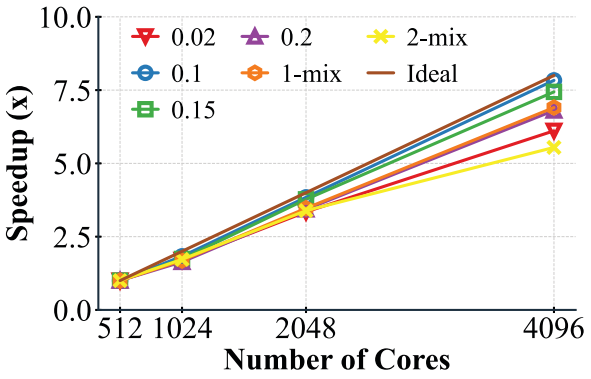


Fig. 9: Evaluation of SBGT's test selections scalability and capability for large-scale group testing, i.e., $N = 30$. The lowest and highest scaling efficiency at 4096 CPU cores are 69.3% and 97.9%. The average scaling efficiency and execution time is 84.7% and 13 minutes across six prior risk settings.

achieving strong scaling efficiency of up to 97.9% at 4096 CPU cores. In the future, we plan to further optimize SBGT with different HPC technologies (such as CUDA and MPI) for next-generation disease surveillance.

References

[1] S. Lohse, T. Pfuhl, B. Berkó-Göttel, J. Rissland, T. Geißler, B. Gärtner, S. L. Becker, S. Schneitler, and S. Smola, "Pooling of Samples for Testing for SARS-CoV-2 in Asymptomatic People," *The Lancet Infectious Diseases*, vol. 20, no. 11, pp. 1231–1232, 2020.

[2] F. Majid, S. B. Omer, and A. I. Khwaja, "Optimising SARS-CoV-2 Pooled Testing for Low-resource Settings," *The Lancet Microbe*, vol. 1, no. 3, 2020.

[3] C. A. Hogan, M. K. Sahoo, and B. A. Pinsky, "Sample Pooling as a Strategy to Detect Community Transmission of SARS-CoV-2," *Jama*, vol. 323, no. 19, p. 1967, 2020.

[4] D. Donoho, M. Lofti, and B. Ozturkler. (2020) The Mathematics of Mass Testing for COVID-19. [Online]. Available: https://sinews.siam.org/Details-Page/the-mathematics-of-mass-testing-for-covid-19

[5] R. Dorfman, "The Detection of Defective Members of Large Populations," *The Annals of Mathematical Statistics*, vol. 14, no. 4, pp. 436–440, 1943. [Online]. Available: http://www.jstor.org/stable/2235930

[6] C. Tatsuoka, W. Chen, and X. Lu, "Bayesian Group Testing with Dilution Effects," *Biostatistics*, p. kxac004, Apr. 2022. [Online]. Available: https://doi.org/10.1093/biostatistics/kxac004

[7] N. Shental, S. Levy, V. Wuvshet, S. Skorniakov, B. Shalem, A. Ottolenghi, Y. Greenshpan, R. Steinberg, A. Edri, R. Gillis, M. Goldhirsh, K. Moscovici, S. Sachren, L. M. Friedman, L. Nesher, Y. Shemer-Avni, A. Porgador, and T. Hertz, "Efficient High-throughput SARS-CoV-2 Testing to Detect Asymptomatic Carriers," *Science Advances*, vol. 6, no. 37, 2020.

[8] S. Ghosh, A. Rajwade, S. Krishna, N. Gopalkrishnan, T. E. Schaus, A. Chakravarthy, S. Varahan, V. Appu, R. Ramakrishnan, S. Ch, M. Jindal, V. Bhupathi, A. Gupta, A. Jain, R. Agarwal, S. Pathak, M. A. Rehan, S. Consul, Y. Gupta, N. Gupta, P. Agarwal, R. Goyal, V. Sagar, U. Ramakrishnan, S. Krishna, P. Yin, D. Palakodeti, and M. Gopalkrishnan, "Tapestry: A Single-Round Smart Pooling Technique for COVID-19 Testing," *medRxiv*, 2020.

[9] W. Chen, X. Lu, and C. Tatsuoka, "HiBGT: High-Performance Bayesian Group Testing for COVID-19," in *29th IEEE International Conference on High Performance Computing, Data, and Analytics*, vol. 1, no. 1. IEEE, 2022, pp. 176–185.

[10] C. Tatsuoka and T. Ferguson, "Sequential Classification on Partially Ordered Sets," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 65, no. 1, p. 143–157, 2003.

[11] J. Yu, Y. Huang, and Z.-J. Shen, "Optimizing and Evaluating PCR-based Pooled Screening During COVID-19 Pandemics," *Scientific reports*, vol. 11, no. 1, pp. 1–14, 2021.

[12] T. S. Ferguson and C. Tatsuoka, "An Optimal Strategy for Sequential Classification on Partially Ordered Sets," *Statistics Probability Letters*, vol. 68, no. 2, p. 161–168, 2004.

[13] C. Tatsuoka, "Optimal Sequencing of Experiments in Bayesian Group Testing," *Journal of Statistical Planning and Inference*, vol. 133, no. 2, pp. 479 – 488, 2005.

[14] ——, "Sequential Classification on Lattices with Experiment-Specific Response Distributions," *Sequential Analysis*, vol. 33, no. 3, p. 400–420, 2014.

[15] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, p. 720–748, sep 1999. [Online]. Available: https://doi.org/10.1145/324133.324234