PARALLEL RANDOM SAMPLING WITHOUT REPLACEMENT

NAWAT NGERNCHAM

**A SENIOR PROJECT SUBMITTED IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF BACHELOR OF SCIENCE
(COMPUTER SCIENCE)
MAHIDOL UNIVERSITY INTERNATIONAL COLLEGE
MAHIDOL UNIVERSITY
2024**

Thesis
entitled

PARALLEL RANDOM SAMPLING WITHOUT REPLACEMENT

was submitted to the Mahidol University International College, Mahidol University
for the degree of Bachelor of Science (Computer Science)
on
April 21, 2024

.....
Nawat Ngerncham
Candidate

.....
Asst. Prof. Kanat Tangwongsan
Advisor

.....
Dr. Piti Ongmongkolkul, Ph.D.
Chair of Science Division
Mahidol University International College
Mahidol University

.....
Dr. Sunsern Cheamanunkul, Ph.D.
Program Director
Bachelor of Science in Computer Science
Mahidol University International College
Mahidol University

ACKNOWLEDGEMENTS

Firstly, I would like to mention that this thesis does not include as much work as I wish it did. My original topic for senior project was on Vertex Removal in Approximate Nearest Neighbor Graphs. However, due to the limited time and unusable results, I had to pivot my topic to this instead in the last 6 weeks of my last trimester.

With that aside, I would like to first thank my advisor, Ajarn Kanat, who has guided me through this senior project despite all of my stupid ideas. I definitely wouldn't have finished it in time for graduation if it weren't for him. Of course, it goes without saying that I am also very grateful of all of my Ajarns who have taught and helped me through my 4 years of undergraduate studies. I will make sure that all the lessons that they have taught me (drilled into my head) are put to good use.

I also want to thank my (dear) friends and the 1502 family. You definitely are the funniest, weirdest, and occasionally the most unhinged bunch I have met. I will always remember the fun times we had together and I hope that we still know each other after we graduate. I also hope that the memes that we put on the Meme Board will be eternalized for future generations (batches) to see.

Finally, I would like to thank mom and dad for always being the most supportive people I've ever met, even when I may have not treated you the best during my most stressful times. Please know that I will be forever thankful for every thing that you have done for me.

Nawat Ngerncham

PARALLEL RANDOM SAMPLING WITHOUT REPLACEMENT

NAWAT NGERNCHAM 6390496 ICCS/B

B.Sc. (COMPUTER SCIENCE)

SENIOR PROJECT ADVISOR: ASST. PROF. KANAT TANGWONGSAN, Ph.D.

ABSTRACT

Random sampling without replacement (RSWR) is the process of picking k elements uniformly at random from a data set of size N such that none of them are the same. In the real world, it is commonly used in test-train data set splitting, sub-sampling for improving performance, and much more. Among existing algorithms for RSWR, there are sequential algorithms that perform very well when k is small relative to N. On the other end of the spectrum, there are parallel algorithms that perform well when k is large relative to N. As a result, there is a performance gap between when $k \ll N$ and $k \approx N$ where we must pick between the sequential algorithm that only gets slower as k grows or the parallel algorithm that wastes too much computation.

This senior project describes a parallel RSWR algorithm based on Parallel Permutation and analytically show that it does not waste too much extra computation. We also show empirically that it fills in the performance gap that existing RSWR algorithms have.

KEY WORDS : PARALLEL ALGORITHMS / RANDOM SAMPLING / THEORY

23 pages

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I
BACKGROUND

1.1 Introduction

Random sampling without replacement (RSWR) is a widely used statistical technique used in data science and machine learning to sub-sample the entire data set for better efficiency or to do train-test data set splitting. The problem of RSWR is formally defined below.

Definition 1.1.1 (Random Sampling without Replacement)**.** Given a data set X, we construct a *random sample without replacement* S such that $S \subseteq X$ with size k and every element of X has equal probability of being included in S.

Before we describe the existing algorithms, let us first define the *nested fork-join parallelism model* and the quantities *work* and *span*. They are used to model how multi-core, shared-memory machines work and measure the theoretical performance of parallel algorithms, respectively.

Definition 1.1.2 (Fork-Join Parallelism)**.** The *nested fork-join parallelism* model is a computation model where the program starts from one thread, splits into two threads to execute two computations in parallel—fork—then combines the results of both computations together in the original thread once both computations are done—join. The nested nature comes from how each forked computation can also fork again within its own computation.

Definition 1.1.3 (Work)**.** The *work*, often denoted by W, of a parallel algorithm is the total number of operations done in the whole computation.

Definition 1.1.4 (Span)**.** The *span*, often denoted by S, of a parallel algorithm is the longest chain of dependencies in a computation directed acyclic graph (DAG).

To further illustrate this, consider Algorithm 1. Suppose that computations A, B, and C take a, b, and c operations, respectively. The *work* of Algorithm 1 is $a + b + c$ and the *span* is $\max(a, b) + c$.

Algorithm 2 PICKANDREMOVE

Input: Data set X, Sample size k
Output: Sample S with $|S| = k$
  for $i = 1, 2, \ldots, k$ **do**
    Pick data point x from X uniformly at random
    Add x to S
    Remove x from X
  end for
  return S

Figure 1.2: Naive Pick-and-Remove Algorithm

Algorithm 1 ParComp

Compute A and B in parallel
Compute C

Figure 1.1: Example Parallel Algorithm

1.2 Existing Algorithms

1.2.1 Naive Pick-and-Remove Sampling

In this section, we will describe existing algorithms used for RSWR. The first algorithm—Naive Pick-and-Remove—is described in Algorithm 2. While it has a running time of $O(k)$ assuming that we can perform removal in $O(1)$, it is extremely sequential. That is, the current iteration *always* relies on the results of the previous iteration. Consequently, it cannot be parallelized and its performance on a modern, multi-core machine will suffer as k grows compared to other algorithms that can be parallelized.

1.2.2 Priority Sampling

The second algorithm—Priority Sampling—is described in Algorithm 3. This method of sampling works because, by assigning uniform random priorities to each element, the probability that an element is among the lowest k is equal.

Priority Sampling has a running time of $O(n)$ w.h.p.[1] This is because iterating through X to assign

[1] An event happens with high probability (w.h.p.) if it happens with probability $\geq 1 - n^{-c}$ for some constant $c > 0$

Algorithm 3 PRIORITYSAMPLE

Input: Data set X, Sample size k
Output: Sample with size k
Assign priority from $\mathrm{Uniform}(0, 1)$ to each element of X
$x_k \leftarrow$ QUICKSELECT(X, k) using priority for comparison
return $\{x \in X$: Priority of x is at most the priority of $x_k\}$

Figure 1.3: Priority Sampling Algorithm

Algorithm 4 PARPRIORITYSAMPLE

Input: Data set X, Sample size k
Output: Sample with size k
Assign priority from $\mathrm{Uniform}(0, 1)$ to each element of X in parallel
$x_k \leftarrow$ PARQUICKSELECT(X, k) using priority for comparison
return $\{x \in X$: Priority of x is at most the priority of $x_k\}$ constructed in parallel

Figure 1.4: Parallel Priority Sampling Algorithm

a priority and filtering X down to the returning sample can be done in linear time and QUICKSELECT has $O(n)$ running time w.h.p. [1]

From this, it is quite straightforward to derive a parallel version of Priority Sampling simply by assigning the priorities and filtering in parallel and replacing QUICKSELECT with its parallel variant. As a result, we now have the parallel variant of Priority Sampling described in Algorithm 4. This algorithm has $O(n)$ work and $O(\log^2 n)$ running time w.h.p. as PARQUICKSELECT has $O(n)$ work and $O(\log^2 n)$ w.h.p. [2] and priority assignment and filtering in parallel has $O(n)$ work and $O(\log n)$ span.

In practice, priority sampling performs quite well overall as we'll see later in Chapter 3. However, the amount of wasted work increases as the sample size becomes smaller relative to the data set size since we need to work on the entire data set instead of a small subset of it.

1.2.3 Permutation Sampling

The last algorithm—Permutation Sampling—is described in Algorithm 5. Since we are shuffling the data set around, we end up with a new permutation where each element of the original data set has equal probability of ending up within the first k elements of the new permutation. In fact, the process of swapping is similar to picking and removing an element from the data set in the Pick-and-

Algorithm 5 PERMUTATIONSAMPLE

Input: Data set X, Sample size k
Output: Sample with size k
  Generate swap target H of length n such that $i \leq H[i] \leq n$
  PERMUTE(X, H)
  return $X[:k]$

Figure 1.5: Permutation Sampling Algorithm

Algorithm 6 PERMUTE

Input: Data set X, Swap target H
  for $i = 1, 2, \ldots, n$ **do**
    SWAP($A[i], A[H[i]]$)
  end for

Figure 1.6: Knuth Shuffle Algorithm

Remove Algorithm but no removal explicitly happens. Instead, the already picked element are simply swapped out of the pool of elements that can be picked from. This follows from how swap targets H is generated. For the construction of a new permutation, we use Knuth Shuffle (a.k.a. Fisher-Yates) [3] as described in Algorithm 6.

Permutation Sampling has a running time of $O(n)$ as generating the swap targets H, doing Knuth Shuffle based on H, and truncating the permuted array down to k elements can be done in linear time.

In fact, we can also only generate k swap targets instead of n and terminate Knuth Shuffle early after the first k iterations. Since we are truncating the returning sample down to the first k elements, swapping any elements past the k-th does not affect the output of the algorithm in any way. With this in mind, we can improve the running time of this algorithm down to $O(k)$ which is the same as the Pick-and-Remove Algorithm. However, it also suffers the same problem as the Pick-and-Remove Algorithm as Knuth Shuffle also seems like a highly sequential algorithm.

1.3 Parallel Permutation Sampling

Unlike the Pick-and-Remove Algorithm, Knuth Shuffle does not fully rely on the results of previous iterations. Specifically, if we are currently at iteration i, the Pick-and-Remove Algorithm relies on

knowing which elements are removed in iterations $1, 2, \ldots, i-1$. On the other hand, Knuth Shuffle would be trying to swap index i with some other index and only needs to know which indices i has been swapped with before iteration i. That is, index i is ready to be swapped once every element that has to be swapped with it has been swapped. We will analyze this further in the next section.

1.3.1 Analyzing the Dependency of Knuth Shuffle

The main content of this section will be a summary of useful results from Shun et al [4]. Firstly, Shun et al. showed that the swap targets H can be transformed into a dependency DAG similar to how forks and joins of parallel algorithms can be described as a computation DAG. In fact, this dependency DAG also resembles a randomized binary tree.

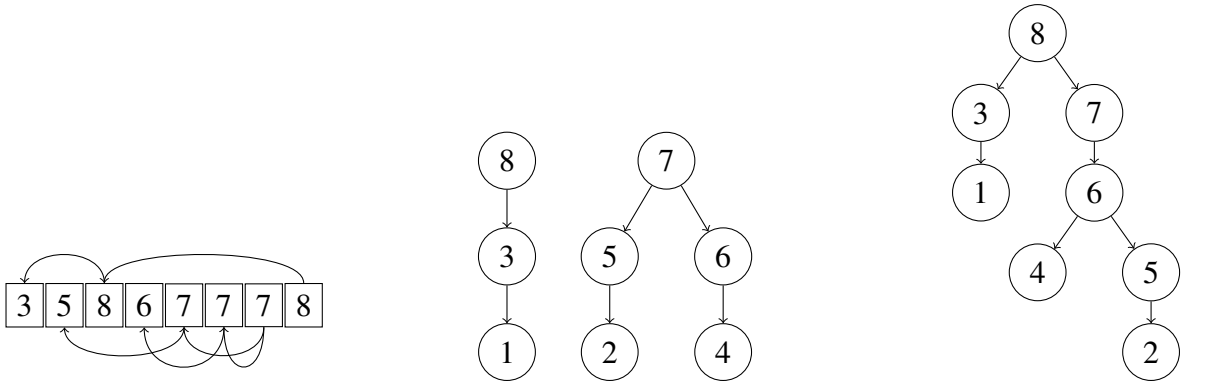
Let H be the generated swap targets. Consider valid indices i, j of H such that $1 \leq i \leq j \leq |H|$. We say that index j depends on index i if $H[i] = j$. Intuitively, this means that i is to be swapped with j and this swap must happen before j is swapped with another index later on. We also define a special case when $H[i] = i$ to be when i depends on nothing. Based on this, we can construct a dependency forest where each node is labeled with each index of H and node j is the parent of node i if index j depends on index i. An example of this transformation can be seen in Figures 1.7a and 1.7b. In the visualization, node j points to node i if index j depends on index i.

Once we have a dependency forest, we can then modify it into a dependency DAG. This is done by first modifying each disconnected tree. For every node that has more than one child, we chain each child with each other in descending order. For example, node 7 has two children in Figure 1.7b: node 5 and node 6. We disconnect node 5 from node 7 and chain it to node 6 instead as seen in Figure 1.7c.

After every disconnected tree is modified, we connect the root of each tree T to be the right child of the root of another tree T' whose root's label has higher index than T's root's label. This results in the roots of every disconnected tree making up the right spine of the dependency DAG in descending order. For example, the root labels of the disconnected trees shown in Figure 1.7b are 7 and 8, so we connect node 7 to be the right child of node 8 as seen in Figure 1.7c. Furthermore, Shun et al. also showed that this dependency DAG shares the same distribution as a randomized binary tree. Hence, we have that this dependency DAG has height of $O(\log n)$ w.h.p. [4]

The next result Shun et al. showed is that if we are able to detect *readiness* in constant time, then Parallel Knuth Shuffle would have $O(n)$ work and $O(\log n)$ span w.h.p. [4] First, let us define *readiness* in the dependency DAG.

Definition 1.3.1 (Readiness)**.** In a dependency DAG, we say that node i (representing step i) is *ready*



(a) Dependency of H (b) Extracted into a forest (c) Modified into DDAG

Figure 1.7: DDAG constructed from $H = [\underset{1}{3}, \underset{2}{5}, \underset{3}{8}, \underset{4}{6}, \underset{5}{7}, \underset{6}{7}, \underset{7}{7}, \underset{8}{8}]$

to be processed if all of its children have already been processed.

For example, let us consider the dependency DAG show in Figure 1.7c. We have that node 8 is ready once nodes 3 and 7 have been processed, node 6 is ready once nodes 4 and 5 have been processed, node 5 is ready once node 2 has been processed, and so on.

For each node in the dependency DAG, we need to check if it is ready until it is actually ready. Once it is ready, we process it and mark it as processed so that we do not check if the node is ready ever again. This means that the number of times we need to check for the readiness of a node is its height in the dependency DAG.

Since nodes that are at the same level are not dependent on each other, they can be processed in parallel. Specifically for Knuth Shuffle, this means that we can transform its swap targets into a dependency DAG and swap the indices that are represented by nodes of the same level in parallel.

If we could check for readiness in constant time, then the span of Parallel Knuth Shuffle is only dependent on the number of times the check is performed which is simply the height of dependency DAG or $O(\log n)$ w.h.p. Similarly, the expected work can be derived from the sum of every node's expected height. This is because for each node, it has to check for readiness until it is processed. If readiness checks and processing (swapping) can be done in constant time, then the work per node is simply its height. Hence, the expected work of the entire algorithm can be computed by the following recurrence:

$$W(n) = O(\log n) + \frac{1}{n}\sum_{i=0}^{n-1}(W(i) + W(n-i-1))$$

which solves to $O(n)$ [4].

Lastly, Shun et al. showed that Parallel Knuth Shuffle produces the same output as a sequential one given the same swap targets [4]. This follows from the fact that the relative order that swaps happen

Algorithm 7 WRITEMIN

Input: Memory address l, New value x
$DRAM[l] \leftarrow \min(DRAM[l], x)$

Figure 1.8: WRITEMIN primitive

is the same as the sequential variant. That is, Sequntial Knuth Shuffle processes index $1, 2, \ldots, n$ and so does the parallel one. The only difference is that some indices are processed in parallel if they are not trying to swap to the same location.

1.3.2 Deterministic Reservation

In order to check for readiness in constant time, we will use a framework for designing parallel algorithms by Blelloch et al. called Deterministic Reservation [5]. The framework is mainly designed for parallel algorithms whose computation DAG is deterministic. That is, how threads are forked will always be the same for the same input.

For instance, PARMERGESORT will always have the same dependency DAG for any unique input I since I is always split in the middle to be sorted in recursive calls. Hence, there is only one computation DAG for I. On the other hand, PARQUICKSORT could have two different computation DAGs on the same input I. In one, the L (lesser than) partition could be larger than the G (greater than) partition so recursive calls on L could have more nested forks than on G while it could be the other way around in another computation DAG dependending on the pivot picked.

Through deterministic reservation, we are able to enforce a degree of determinism despite the possibility of different CPU scheduling. The main idea of deterministic reservation is to process the input in rounds where each round is split into two phases: Reservation and Commit. Intuitively, the reservation phase decides which steps is going to be processed in the current round and the commit phase carries out the actual processing. In practice, reservations and commits are only done on a prefix of the remaining steps each round for better performance.

Additionally, deterministic reservation comes with a primitive WRITEMIN described in Algorithm 7 for reconciling conflicts during the reservation phase. Specifically, it ensures that out of all the steps that are trying to reserve a slot, the smallest step will always get the reservation. Its utility will become clearer later on when we describe the algorithm for Parallel Knuth Shuffle. (The WRITEMAX variant of this primitive also exists.)

While WRITEMIN on its own could cause race conditions where it reads $DRAM[l]$ before an even

Algorithm 8 RESERVE

Input: Reservation R, Swap targets H, Index i
  WRITEMIN($R[i], i$)
  WRITEMIN($R[H[i]], i$)

Figure 1.9: RESERVE function

Algorithm 9 COMMIT

Input: Reservation R, Swap targets H, Index i
Output: 0 if swap happened and 1 otherwise
  if $R[i] = R[H[i]] = i$ **then**
    SWAP($A[i], A[H[i]]$)
    return 0
  end if
  return 1

Figure 1.10: COMMIT function

smaller value than x is written to address l, this can easily be solved via the use of atomic variables as reservation slots usually store a primitive type so its atomic variant is generally supported in modern programming languages and machines.

1.3.3 Parallel Knuth Shuffle

Finally, we describe Parallel Knuth Shuffle in Algorithm 10. Since we are using deterministic reservation for designing the algorithm, we will first describe how the reservation and commit phases will be carried out. The reserve and commit functions are described in Algorithms 8 and 9.

Intuitively, when reserving for index i, RESERVE will try to write i to $R[i]$ and $R[H[i]]$ to reserve itself and its swap target for the swap in the commit phase. Then, WRITEMIN ensures that if index $j < i$ tries to swap with i or $H[i]$, then index i will not be processed in this round and index j will processed before i since it will always win the reservation. Then, during the commit phase, COMMIT only swaps the elements at index i and $H[i]$ only if i successfully reserved the spot at index i and $H[i]$.

For example, consider when the swap targets $H = [3, 5, 8, 6, 7, 7, 7, 8]$ and deterministic reservation is processing 3 elements in each round. Suppose that we have already processed the first 4 elements. Then, we would be trying to swap elements at index 5, 6, and 7. So, we would be calling RESERVE($R, H, 5$), RESERVE($R, H, 6$), and RESERVE($R, H, 7$) in parallel. By using WRITEMIN,

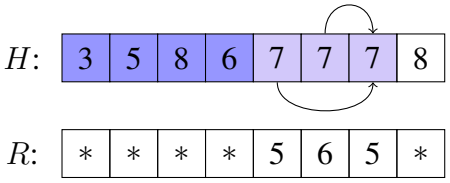


Figure 1.11: Swap target H and reservation slots R after reserving indices 5, 6, and 7

Algorithm 10 PARPERMUTE

Input: Array A, Swap targets H

$I \leftarrow [1, 2, \ldots, n]$ ▷ unprocessed indices
$R \leftarrow [n, n, \ldots, n]$ of length n ▷ reservation slot
while I is not empty **do**
  parfor $i \in I$ **do** ▷ only prefix of I in practice
    RESERVE(R, i)
  end parfor
  parfor $i \in I$ **do**
    COMMIT(R, i)
  end parfor
  Remove committed indices from I
end while

Figure 1.12: Parallel Knuth Shuffle

we will always end up with the R seen in Figure 1.11 no matter which order the RESERVEs are executed. Finally, COMMIT will only execute SWAP(A[5], A[7]) before ending the round.

Finally, we fully describe Parallel Knuth Shuffle in Algorithm 10. It is easy to see that RESERVE and COMMIT are constant-time functions. With results from Shun et al. described in Section 1.3.1, we know that the number of times that RESERVE and COMMIT on index i will be called without swapping—something analogous to the analysis' readiness checks—is the height of node i in the dependency DAG constructed from the input swap targets H. Hence, we have that Parallel Knuth Shuffle has $O(n)$ work and $O(\log n)$ span w.h.p.

With Parallel Knuth Shuffle, we can easily derive the parallel variant of Permutation Sampling as described in Algorithm 11. Since generating swap targets H and constructing $X[:k]$ can also be done in parallel within $O(n)$ work and $O(\log n)$ span, we have that Parallel Permutation Sampling has $O(n)$ expected work and $O(\log n)$ span w.h.p. However, we still run into the same problem of doing too much work since Parallel Knuth Shuffle shuffles the entire array instead of the first k elements.

Algorithm 11 PARPERMUTATIONSAMPLE

Input: Data set X, Sample size k
Output: Sample with size k
Generate swap target H of length n such that $i \leq H[i] \leq n$ in parallel
PARPERMUTE(X, H)
return $X[:k]$

Figure 1.13: Permutation Sampling Algorithm

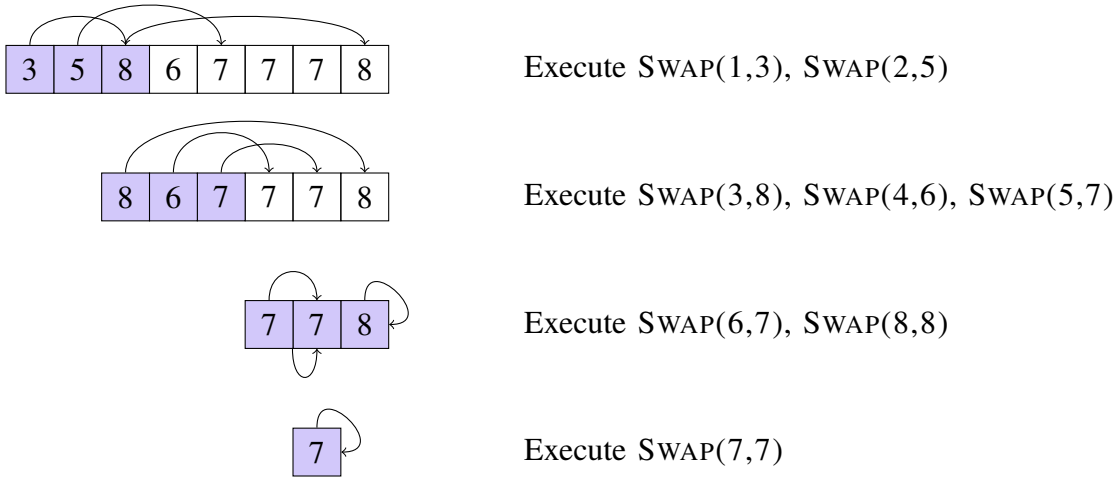


Figure 1.14: Example of how Parallel Knuth Shuffle would work

CHAPTER II
PARALLEL PERMUTATION SAMPLING WITH EARLY TERMINATION

As described in the previous chapter, the analysis that Shun et al. did was for shuffling entire input array in parallel. However, Permutation Sampling only requires the first k iterations of Knuth Shuffle to be executed as any element swapped past the first k will be ignored. Hence, we will re-analyze the work and span of Parallel Knuth Shuffle in the case that we only process the first k elements of the input array.

First, let us modify Algorithm 11 such that it will only process the first k elements. Namely, the only difference is that we only put 1 to k into the array of unprocessed indices instead of the full 1 to n and use an associative array instead of an array of length n for reservation slots. Note that in our benchmarks, R is still only implemented via a concurrent array of length n instead of a phase-concurrent associative array since we did not have enough time to implement it.

2.1 Parallel Knuth Shuffle with Early Termination

2.1.1 Span Analysis

It is easy to see that the span of Parallel Knuth Shuffle with early termination must be bounded above by the span of Parallel Knuth Shuffle without early termination. Hence, it follows directly from Shun et al. that the span of Parallel Knuth Shuffle with Early Termination is $O(\log n)$ [4].

2.1.2 Work Analysis

Similar to Shun et al., we will derive the work of our algorithm by summing up the length of the dependency chains. However, unlike Shun et al., the dependency chains produced by our algorithm may not merge at all since we do not process the full array that we are shuffling. Hence, we cannot rely on the fact that the dependency chain will end up as a binary tree and use a recurrence as in their paper. Instead, we will simply sum up the expected length of dependency chains starting at indices that we are processing.

First, we will identify how dependency chains are formed. We define our own dependency chain similar to Shun et al.'s [4] That is, given a swap targets H and i, j such that $1 \leq i < j \leq |H|$, we say

Algorithm 12 PARPERMUTEEARLY

Input: Array A, Swap targets H, Last index that needs to be swapped k
  $I \leftarrow [1, 2, \ldots, k]$ ▷ k instead of n
  $R \leftarrow$ empty phase-concurrent associative array
  while I is not empty **do**
    parfor $i \in I$ **do**
      RESERVE(R, i)
    end parfor
    parfor $i \in I$ **do**
      COMMIT(R, i)
    end parfor
    Remove committed indices from I
  end while

Figure 2.1: Parallel Knuth Shuffle with Early Termination

that index j *depends on* index i if $H[i] = j$ and i depends on nothing if $H[i] = i$. We then define the length of a dependency chain as follows.

Definition 2.1.1 (Length of Dependency Chain)**.** Let $\mathcal{L}(t)$ be the length of the dependency chain starting at index t. $\mathcal{L}(t)$ is defined below.

$$\mathcal{L}(t) = \begin{cases} 0 & \text{if } t \geq k \text{ or } H[t] = t \\ 1 + \mathcal{L}(H[t]) & \text{otherwise} \end{cases}$$

Note that the first case of this definition comes from the fact that we do not care about dependency chains past the k-th index and that there is no dependency chain if index t doesn't depend on anything, i.e. when $H[t] = t$.

Lemma 2.1.2 (Expected Length of Dependency Chain)**.** *Let $\bar{\mathcal{L}}(t)$ be the expected length of the dependency chain starting at index t.*

$$\bar{\mathcal{L}}(t-1) = \bar{\mathcal{L}}(t) + \frac{1}{n+2-t}$$

Proof. By the definition of expectation, we have that

$$\bar{\mathcal{L}}(t) = \frac{1}{n-(t-1)} \left(\sum_{i=t}^{n} \bar{\mathcal{L}}(i) + 1 \right)$$

since we cannot swap with any of the $t-1$ indices that have already been swapped. Then, we can expand this into

$$\bar{\mathcal{L}}(t) = \frac{1}{n+1-t} \left(\bar{\mathcal{L}}(t+1) + \bar{\mathcal{L}}(t+2) + \cdots + \bar{\mathcal{L}}(k-1) \right) + 1$$

Note that $\bar{\mathcal{L}}(t)$ and $\bar{\mathcal{L}}(j)$ for $j \geq k$ do not show up in the expansion because the length of the dependency chain when index t has to swap with itself and $\mathcal{L}(j)$ for $j \geq k$ are all 0 and so are their expectations. Multiplying both sides with $n+1-t$ and we obtain

$$(n+1-t)\bar{\mathcal{L}}(t) = \bar{\mathcal{L}}(t+1) + \bar{\mathcal{L}}(t+2) + \cdots + \bar{\mathcal{L}}(k-1) + n + 1 - t \tag{2.1}$$

Similarly, we can obtain the following equation via the same derivation.

$$(n+2-t)\bar{\mathcal{L}}(t-1) = \bar{\mathcal{L}}(t) + \bar{\mathcal{L}}(t+1) + \cdots + \bar{\mathcal{L}}(k-1) + n + 2 - t \tag{2.2}$$

Finally, we can subtract Equation (2.1) from Equation (2.2) to arrive at the equation in the claim. Note that $\bar{\mathcal{L}}(j)$ cancels each other out for $j = t+1, t+2, \ldots, k-1$ so they do not show up below.

$$\begin{aligned}
(n+2-t)\bar{\mathcal{L}}(t-1) - (n+1-t)\bar{\mathcal{L}}(t) &= \bar{\mathcal{L}}(t) + (n+2-t) - (n+1-t) \\
(n+2-t)\bar{\mathcal{L}}(t-1) &= (n+1-t)\bar{\mathcal{L}}(t) + \bar{\mathcal{L}}(t) + (n+2-t) - (n+1-t) \\
(n+2-t)\bar{\mathcal{L}}(t-1) &= (n+2-t)\bar{\mathcal{L}}(t) + 1 \\
\bar{\mathcal{L}}(t-1) &= \bar{\mathcal{L}}(t) + \frac{1}{n+2-t}
\end{aligned}$$

□

Theorem 2.1.3 (Expected Sum of Dependency Chains)**.** *Let $W_n(k)$ denote the expected sum of dependency chains starting at indices $1, 2, \ldots, k$.*

$$W_n(k) = O\left(k \log\left(\frac{n}{n-k}\right)\right)$$

Proof. By the definition of $W_n(k)$, we have the following.

$$W_n(k) = \bar{\mathcal{L}}(1) + \bar{\mathcal{L}}(2) + \cdots + \bar{\mathcal{L}}(k)$$

By Lemma 2.1.2, we can expand this to

$$\begin{aligned}
&= \frac{1}{n} + \bar{\mathcal{L}}(2) + \frac{1}{n-1} + \bar{\mathcal{L}}(3) + \cdots + \bar{\mathcal{L}}(k) \\
&= \frac{1}{n} + \frac{1}{n-1} + \bar{\mathcal{L}}(3) + \frac{1}{n-1} + \frac{1}{n-2} + \bar{\mathcal{L}}(4) + \cdots + 0 \\
&\;\;\vdots \\
&= \frac{1}{n} + \frac{2}{n-1} + \cdots + \frac{k-1}{n+1-k}
\end{aligned}$$

Namely, we arrive at the last line as $\frac{1}{n}$ shows up once, $\frac{1}{n-1}$ shows up twice, and so on. This can then be rewritten as follows.

$$\begin{aligned}
W_n(k) &= \frac{1}{n} + \frac{2}{n-1} + \cdots + \frac{k-1}{n+1-k} \\
&= \sum_{i=1}^{k-1} H_{n+1-i} - (k-1)H_{n+1-k} \\
&\leq (k-1)H_n - (k-1)H_{n+1-k} \\
&= (k-1)(H_n - H_{n+1-k}) \\
&< k(H_n - H_{n+1-k}) \\
&= O\left(k \log\left(\frac{n}{n-k}\right)\right)
\end{aligned}$$

□

From Theorem 2.1.3, we now know that the work of Parallel Knuth Shuffle with Early Termination or Algorithm 12 is $O\left(k \log\left(\frac{n}{n-k}\right)\right)$. Consequently, this means that when $k \ll n$, the work approaches $O(k)$. This implies that Parallel Knuth Shuffle does not do any extra work which is our goal. Unfortunately, using our method of analysis, the work approaches $O(n \log n)$ when $k \approx n$ since our method could not capture how the dependency chains merge into each other as we process more indices. However, we know that Parallel Knuth Shuffle has $O(n)$ work and our algorithm is simply that but terminates early, so it is impossible for the work to just increase when k approaches n. As a result, this still means that our algorithm still has a quantifiable theoretical improvement over Parallel Full Knuth Shuffle or Algorithm 10.

2.2 Parallel Permutation Sampling with Early Termination

Using Parallel Knuth Shuffle with Early Termination, we will now describe Parallel Permutation Sampling with Early Termination in Algorithm 13. Again, constructing swap targets H of length k and $X[:k]$ can be done in $O(k)$ work and $O(\log k)$ span via a simple parallel tabulate and truncation operation. Since Parellel Knuth Shuffle with Early Termination has $O\left(k \log\left(\frac{n}{n-k}\right)\right)$ work and $O(\log n)$ span by Theorem 2.1.3 and Shun et al. [4], respectively, we have that Parallel Permutation Sampling with Early Termination also has $O\left(k \log\left(\frac{n}{n-k}\right)\right)$ work and $O(\log n)$ span since the work and span is dominated by that of Parallel Knuth Shuffle with Early Termination's.

Algorithm 13 PARPERMUTATIONEARLYSAMPLE

Input: Data set X, Sample size k
Output: Sample with size k
Generate swap target H of length k such that $i \leq H[i] \leq n$ in parallel
PARPERMUTEEARLY(X, H, k)
return $X[:k]$

Figure 2.2: Permutation Sampling Algorithm

CHAPTER III
EXPERIMENTS

To better quantify the performance of Parallel Permutation Sampling with Early Termination, we will benchmark it and its sequential variant along with the Naive Pick-and-Remove Sampling, Sequential and Parallel Priority Sampling, and Sequential and Parallel Full Permutation Sampling. Every algorithm was implemented in C++ and compiled with `g++` version 13.2.1 with the `-O3` flag. We achieved parallelism via the use of ParlayLib [6] and many of its built-in concurrent data structures and algorithms.

All experiments are run on a machine with an AMD Ryzen 9 5900X 3.7 GHz 12-core CPU and 3600 MHz 32GB of main memory running EndeavorOS with the Arch Linux 6.8.4 kernel. The data set these algorithms are sampling consists of $N = 5 \times 10^8$ randomly generated 32-bit integers. Each algorithm is tested on 1, 12, and 24 threads and various different values of k between 2.5×10^4 to 5×10^8 (size of data set). Relating k to N, the values of k's tested range from $k \approx 100 \log_2 N$ until $k = N$. The full list of k's can be seen in Tables A.1, A.2, and A.3. For each combination of value of k and the number of threads, each algorithm is run 4 times but the timing of the first run is discarded.

Note that in Figures 3.1 and 3.2, Naive refers to Pick-and-Remove Sampling, the Seq and Par prefixes refer to the sequential and parallel variants of an algorithm, respectively. Note also that from this point on, we will refer to Permutation with Early Termination as just Permutation and Permutation without Early Termination as Full Permutation.

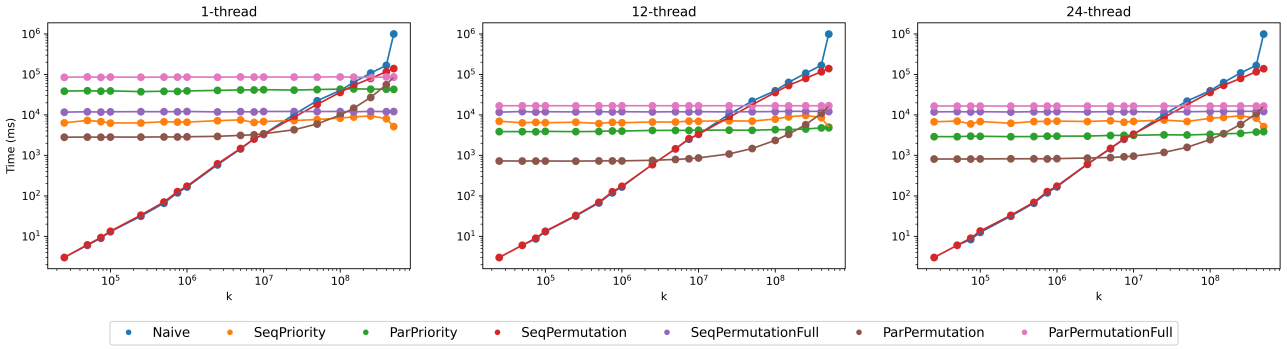


Figure 3.1: Benchmark results in log-log scale

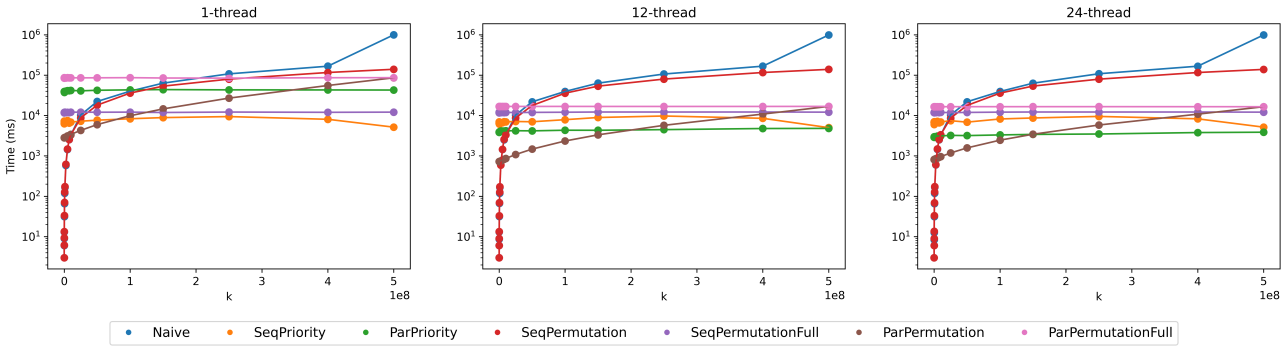


Figure 3.2: Benchmark results in log-linear scale

In Figure 3.1, we can see that the implementations whose running time changes depending on the value of k are Pick-and-Remove Sampling, Sequential Permutation Sampling, and Parallel Permutation Sampling. However, the running time of Parallel Permutation Sampling starts off quite high and stays that way up until when $k \approx 10^7$ since it has the overhead of initializing the reservation slots as an array of size N instead of a more efficient associative array. We can also see that the running time of Pick-and-Remove and Sequential Permutation Sampling are about the same except for $k = N$ where Pick-and-Remove's running time spikes significantly higher than Sequential Permutation's.

On the other hand, both variants of Priority Sampling and Full Permutation Sampling whose sequential or parallel work and span are purely functions of N takes the same amount of time to run regardless of k as expected. Among these, the order from best- to worst-performing on multiple threads is: Parallel Priority, Sequential Priority, Parallel Full Permutation, and Sequential Full Permutation Sampling. However, the parallel variants become the worst-performing algorithms when run on a single thread. Interestingly, Parallel Permutation Sampling still outperforms every other algorithm except for the two $O(k)$ algorithms on a single thread when k is relatively small.

One more trend we can see is that, while Parallel Priority Sampling runs slightly faster with more threads, Parallel Permutation, both early and full, take about the same amount of time to run with 12 and 24 threads.

Zooming out to the log-linear scale in Figure 3.2, we can see that Parallel Permutation Sampling does not perform as well when k gets relatively large. This is to be expected as Parallel Full Permutation is among the worst-performing algorithms and the performance of Parallel Permutation will only converge to its full variant's as k approaches N. Specifically, on 12 and 24 threads and ignoring Pick-and-Remove and Sequential Permutation Sampling, Parallel Permutation Sampling performs the best up until $k \approx N/4$. For any $k > N/4$, Parallel Priority Sampling consistently performs the best.

CHAPTER IV
CONCLUSION

In this thesis, we have proven analytically that Parallel Permutation Sampling with Early Termination is a parallel algorithm for random sampling without replacement (RSWR) that has work proportional to the target sample size k and span that does not degrade beyond existing algorithms. This is better than existing parallel algorithms for RSWR whose work is only proportional to the size of the input data set N. Additionally, we have also shown empirically through benchmarks that, while Parallel Permutation Sampling with Early Termination does not perform very well as k approaches N, it helps bridge the performance gap between when $k \ll N$ and $k \approx N$ such that when k is in the approximate range of $50\sqrt{n} \leq k \leq N/4$, it is the best performing algorithm and it does not waste computation in sampling the data set. Furthermore, the implementation of our algorithm can be further optimized with a more efficient data structure that improve the performance even more and widen the range of k that it is the best performing algorithm.

APPENDIX

APPENDIX A
AVERAGE RUNNING TIME OF SAMPLING ALGORITHMS

In this appendix, we show the average running time in milliseconds of each sampling algorithm rounded to the nearest whole number. Notation-wise, the S and P prefix refers to the sequential and parallel variants of an algorithm, respectively. Note that Naive refers to the Pick-and-Select algorithm, Priority algorithms are shortened down to Prior, Permutation with and without Early Termination are shortened down to Perm and PermFull, respectively.

Table A.1: 1-thread Timings (ms)

Algorithm k	Naive	SPrior	PPrior	SPerm	SPermFull	PPerm	PPermFull
2.5×10^4	3	6353	38570	3	11687	2808	84723
5.0×10^4	6	7284	39418	6	11894	2827	86212
7.5×10^4	9	6830	38471	9	11713	2808	84727
1.0×10^5	13	6283	39191	13	11924	2832	85688
2.5×10^5	31	6331	37418	33	11961	2818	84937
5.0×10^5	66	6760	38328	70	11963	2853	85608
7.5×10^5	118	6639	37986	127	11931	2843	85612
1.0×10^6	165	6615	39015	173	12096	2878	86059
2.5×10^6	583	7189	40162	617	11828	2945	85322
5.0×10^6	1437	7503	41322	1475	11960	3111	86282
7.5×10^6	2520	6500	41205	2552	11831	3229	85612
1.0×10^7	3281	6858	41880	3312	12114	3409	86129
2.5×10^7	10265	7228	40861	8756	12094	4283	85628
5.0×10^7	22261	7659	42204	18066	12056	5959	85960
1.0×10^8	40047	8314	43202	35968	12136	9869	86605
1.5×10^8	63329	8880	43884	53368	11876	14577	84925
2.5×10^8	107599	9406	43209	79702	12183	26967	84333
4.0×10^8	166492	8059	42998	116247	11996	55419	86283
5.0×10^8	996889	5149	42755	139707	12130	85907	86587

Table A.2: 12-thread Timings (ms)

Algorithm k	Naive	SPrior	PPrior	SPerm	SPermFull	PPerm	PPermFull
2.5×10^4	3	6952	3852	3	11659	727	16792
5.0×10^4	6	6353	3855	6	12068	722	16788
7.5×10^4	9	6561	3826	9	11750	727	16773
1.0×10^5	13	6389	3908	13	11954	723	16805
2.5×10^5	32	6577	3844	33	11806	721	16770
5.0×10^5	67	6116	3896	69	11977	728	16778
7.5×10^5	119	6559	4013	126	11794	731	16783
1.0×10^6	166	6404	3984	172	11970	731	16795
2.5×10^6	594	6645	4126	593	11800	753	16801
5.0×10^6	1444	6633	4128	1453	11949	792	16789
7.5×10^6	2502	6994	4129	2577	11833	826	16801
1.0×10^7	3309	6973	4185	3313	11950	863	16794
2.5×10^7	10270	7113	4202	8695	11827	1081	16785
5.0×10^7	21830	7001	4168	17711	11948	1472	16786
1.0×10^8	39429	7885	4319	35736	12059	2338	16802
1.5×10^8	63315	8931	4309	53407	12210	3321	16778
2.5×10^8	106585	9728	4471	79870	12198	5714	16763
4.0×10^8	166766	8522	4763	116237	12211	10852	16780
5.0×10^8	986407	5010	4800	139019	12114	16798	16823

Table A.3: 24-thread Timings (ms)

Algorithm k	Naive	SPrior	PPrior	SPerm	SPermFull	PPerm	PPermFull
2.5×10^4	3	6731	2898	3	11831	810	16527
5.0×10^4	6	7026	2894	6	11984	812	16541
7.5×10^4	8	5974	2971	9	11805	811	16487
1.0×10^5	12	6859	2952	14	11965	816	16520
2.5×10^5	31	6195	2895	33	11754	821	16490
5.0×10^5	66	6857	2910	69	11946	823	16536
7.5×10^5	119	6868	2978	127	11829	820	16532
1.0×10^6	167	6996	2976	174	11982	830	16533
2.5×10^6	598	6862	2984	610	11816	858	16516
5.0×10^6	1457	7167	3103	1490	11967	885	16520
7.5×10^6	2501	6593	3111	2550	11809	921	16518
1.0×10^7	3274	6924	3128	3336	11957	956	16528
2.5×10^7	10278	7433	3221	8633	11784	1181	16547
5.0×10^7	21903	6860	3181	17766	11926	1578	16550
1.0×10^8	39327	8211	3307	35858	12121	2452	16551
1.5×10^8	63109	8694	3405	53596	12212	3434	16564
2.5×10^8	108291	9491	3483	79509	12205	5815	16575
4.0×10^8	166475	8275	3784	116077	12203	10804	16545
5.0×10^8	989396	5150	3857	138769	12116	16500	16511

REFERENCES

[1] Luc Devroye. Exponential bounds for the running time of a selection algorithm. *Journal of Computer and System Sciences*, 29(1):1–7, 1984. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(84)90009-6. URL https://www.sciencedirect.com/science/article/pii/0022000084900096.

[2] Christian Siebert. Scalable and efficient parallel selection. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wa śniewski, editors, *Parallel Processing and Applied Mathematics*, pages 202–213, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-55224-3.

[3] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, jul 1964. ISSN 0001-0782. doi: 10.1145/364520.364540. URL https://doi.org/10.1145/364520.364540.

[4] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms*, 2015. URL https://api.semanticscholar.org/CorpusID:1687316.

[5] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. *SIGPLAN Not.*, 47(8):181–192, feb 2012. ISSN 0362-1340. doi: 10.1145/2370036.2145840. URL https://doi.org/10.1145/2370036.2145840.

[6] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib - a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, page 507–509, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450369350. doi: 10.1145/3350755.3400254. URL https://doi.org/10.1145/3350755.3400254.