Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 4284 – 4297
© TÜBİTAK
doi:10.3906/elk-1904-135

Parallel algorithms for computing sparse matrix permanents

Kamer KAYA
Department of Computer Science and Engineering, Faculty of Engineering and Natural Sciences,
Sabancı University, İstanbul, Turkey

Received: 16.04.2019 • **Accepted/Published Online:** 29.08.2019 • **Final Version:** 26.11.2019

Abstract: The permanent is an important characteristic of a matrix and it has been used in many applications. Unfortunately, it is a hard to compute and hard to approximate the immanant. For dense/full matrices, the fastest exact algorithm, Ryser, has $\mathcal{O}(2^{n-1}n)$ complexity. In this work, a parallel algorithm, SkipPer, is proposed to exploit the sparsity within the input matrix as much as possible. SkipPer restructures the matrix to reduce the overall work, skips the unnecessary steps, and employs a coarse-grain, shared-memory parallelization with dynamic scheduling. The experiments show that SkipPer increases the performance of exact permanent computation up to 140× compared to the naive version for general matrices. Furthermore, thanks to the coarse-grain parallelization, 14–15× speedup on average is obtained with $\tau = 16$ threads over sequential SkipPer. Overall, by exploiting the sparsity and parallelism, the speedup is 2000× for some of the matrices used in the experimental setting. The proposed techniques in this paper can be used to significantly improve the performance of exact permanent computation by simply replacing Ryser with SkipPer, especially when the matrix is highly sparse.

Key words: Sparse matrices, permanent, Ryser's algorithm, multicore CPUs, shared-memory parallelism, load balancing

1. Introduction

Given an $n \times n$ matrix $\mathbf{A}$ with entries $a_{i,j}$ for $1 \leq i, j \leq n$, the permanent is computed as

$$\texttt{perm}(\mathbf{A}) = \sum_{\sigma \in \mathcal{P}} \prod_{i=1}^{n} a_{i,\sigma(i)}, \tag{1}$$

where $\mathcal{P}$ is the set of all permutations of the numbers in $\{1, 2, \cdots, n\}$.

The permanent has many applications and relations in mathematics, statistics, and computer science; for instance, it is related to Fibonacci and Lucas numbers [1] as well as order statistics [2]. Permanents also have found applications in quantum computing: boson sampling, introduced by Aaronson and Arkhipov [3], is a restricted quantum computing model that samples from the probability distribution of identical bosons and this distribution can be obtained by computing the permanents of some matrices obtained throughout the process. To analyze the efficiency of classical computers in simulating boson sampling, a benchmark of the exact permanent computation algorithms was recently performed [4]. Another application of the permanent is from bioinformatics, where exact values are required to find the genotype probability distributions required for DNA profiling [5]. The permanent computation problem is also related to identity testing in simple read-restricted

*Correspondence: kaya@sabanciuniv.edu

This work is licensed under a Creative Commons Attribution 4.0 International License.

circuits [6]. In addition to these, the permanent has interesting relations with graph theory; with an efficient algorithm to compute $\texttt{perm}(\mathbf{A})$, one can count the number of perfect matchings for a bipartite graph or find the number of vertex-disjoint cycle covers of a directed graph. These two statements are direct conclusions of (1) when a (0,1)-matrix $\mathbf{A}$ is generated as the bi-adjacency or adjacency matrix, respectively. More applications of permanents and their relations with theory and practice can be found in [7]. Unfortunately, computing the permanent of a (0,1)-matrix is #P-Complete [8].

Since permanent computation is hard, approximating the permanent is a well-studied problem. Jerrum et al. [9] discussed an approach using Markov chains, which can provide an $(1+\varepsilon)$-approximation for the permanent of a nonnegative matrix in fully polynomial time, with $\tilde{O}(n^{10})$ complexity. Their Markov chain Monte Carlo (MCMC) approach makes use of the underlying graph being bipartite, and the techniques cannot be generalized easily to the general graph case. Štefankovic et al. [10] took the MCMC approach and highlighted the difficulties that arose. They also proposed a Markov Chain efficiently estimating the number of perfect matchings in a special graph class. Gurvits and Samorodnitsky [11] and Linial et al. [12] used matrix scaling and proposed deterministic approximations with exponential guarantees (2^n and e^n, respectively). Dufosse et al. also employed matrix scaling to improve the variance of the existing approximations in practice [13]. There also exist studies focusing on special classes, e.g., positive semidefinite matrices [14] and random matrices with unit variance and vanishing mean [15].

Various studies exist in the literature focusing on the exact computation of (1) [16–19]. When the matrix is dense/full, Ryser's algorithm from 1963 [20] uses the inclusion/exclusion principle and computes the permanent in $\mathcal{O}(2^{n-1}n^2)$ time. Later, Niejenhuis and Wilf proposed to process the sets in Gray-code order to reduce the complexity to $\mathcal{O}(2^{n-1}n)$ [21]. This algorithm is still the most efficient algorithm at hand for generic matrices. From now on, it will be denoted as Ryser.

Besides all the applications of permanents, as mentioned above, it is an interesting problem in practice to understand how fast can we compute the permanent with the current parallel architectures at hand. This work solely focuses on the exact permanent computation problem for sparse matrices and proposes a parallel, novel algorithm called SkipPer. When the matrix is sparse, computing the permanent is not straightforward; indeed, when the number of possible transversals, i.e. sets of n nonzero matrix entries such that no two come from the same row or the same column, is small, one can enumerate all the transversals and sum their contributions to find the permanent. Unfortunately, this is not the case for many sparse matrices. Even when the number of transversals is large, depending on the level of sparsity, SkipPer incurs optimizations that yield a shorter permanent computation time. In addition, although the Gray-code order seems to necessitate inherently sequential iterations, SkipPer can parallelize the execution flow and assign a balanced work load to the threads via dynamic scheduling. Furthermore, since SkipPer does not perform heavy preprocessing on the input matrix, it can easily be plugged into existing permanent computation codes and replace Ryser to obtain significant speedups based on the sparsity.

The rest of this paper is organized as follows: Section 2 presents Ryser's algorithm and related notation. Section 3 describes the basic approach proposed to exploit the sparsity. A further optimization that focuses on skipping the zero-contributing Gray codes with its coarse-grain parallelization is described in Section 4. The related studies in the literature, as well as their differences from the proposed techniques, are summarized in Section 5. Section 6 presents the results of the experiments and Section 7 concludes the paper.

2. Background and notation

For an $n \times n$ matrix, Eq. 1 defines the permanent in terms of transversals. With a naive algorithm, computing the total contribution of all transversals takes $\mathcal{O}(n \times n!)$ time. To reduce the complexity, Ryser's algorithm restructures the original permanent equation and performs inclusion/exclusion as follows:

$$\texttt{perm}(\mathbf{A}) = -1^n \sum_{S \subseteq \{1,2,\cdots,n\}} (-1)^{|S|} \prod_{i=1}^{n} \sum_{j \in S} a_{i,j}. \tag{2}$$

The pseudocode of the Niejenhuis–Wilf variant of Ryser's algorithm, RYSER, is given in Algorithm 1. In the pseudocode, GRAY_g is the gth Gray code with $(n-1)$ bits; a 4-bit Gray code is shown on the right of the algorithm. For a binary-reflected Gray code,

$$\text{GRAY}_g = g \oplus (g \gg 1),$$

where $\oplus$ is the bitwise XOR operation and $g \gg 1$ shifts the bitwise representation of g to the right by one bit.

Let $\text{GRAY}_g[j]$ be the jth bit of GRAY_g, where $j = 1$ denotes the least significant bit. The set bits for each code, i.e. the ones that are equal to 1, represent the indices of the selected columns. To store and keep the impact of these columns, the algorithm keeps maintaining a vector $\mathbf{x}$, whose ith entry relates to the contribution of the ith row of the matrix. For each g, the algorithm first finds the difference between GRAY_g and GRAY_{g-1}, i.e. the changed bit in GRAY_g (line 8). Then the updates due to this bit modification are performed on $\mathbf{x}$ (line 12). Finally, the product of these vector entries is added to p (line 14). Note that the Niejenhuis–Wilf variant practically uses an $(n-1)$-bit Gray code, not an n-bit one, for an $n \times n$ matrix to halve the execution time.

Algorithm 1 : RYSER

Input: $\mathbf{A}$ ($n \times n$ matrix)
Output: $\texttt{perm}(\mathbf{A})$

1: **for** $i = 1$ to n **do**
2: $\quad sum = 0$
3: $\quad$ **for** $j = 1$ to n **do**
4: $\quad\quad sum \leftarrow sum + a_{i,j}$
5: $\quad \mathbf{x} \leftarrow a_{i,n} - \frac{sum}{2}$
6: $p \leftarrow \prod_{i=1}^{n} \mathbf{x}[i]$
7: **for** $g = 1$ to $2^{n-1} - 1$ **do**
8: $\quad j \leftarrow \log_2(\text{GRAY}_g \oplus \text{GRAY}_{g-1})$
9: $\quad s \leftarrow 2 \times \text{GRAY}_g[j] - 1$
10: $\quad prod \leftarrow 1$
11: $\quad$ **for** $i = 1$ to n **do**
12: $\quad\quad \mathbf{x}[i] \leftarrow \mathbf{x}[i] + (s \times a_{i,j})$
13: $\quad\quad prod \leftarrow prod \times \mathbf{x}[i]$
14: $\quad p \leftarrow p + ((-1)^g \times prod)$
15: **return** $p \times (4 \times (n \bmod 2) - 2)$

	BINARY	GRAY
GRAY_0	0000	0000
GRAY_1	0001	0001
GRAY_2	0010	0011
GRAY_3	0011	0010
GRAY_4	0100	0110
GRAY_5	0101	0111
GRAY_6	0110	0101
GRAY_7	0111	0100
GRAY_8	1000	1100
GRAY_9	1001	1101
GRAY_{10}	1010	1111
GRAY_{11}	1011	1110
GRAY_{12}	1100	1010
GRAY_{13}	1101	1011
GRAY_{14}	1110	1001
GRAY_{15}	1111	1000

As mentioned above, Ryser's algorithm is currently the most efficient one for the exact computation of permanents of generic, dense matrices. In the pseudocode above, the outer loop (line 7) iterates exactly $(2^{n-1}-1)$ times and the inner loop (line 11) iterates n times. Hence, the overall complexity is $\mathcal{O}(2^{n-1}n)$.

3. SpaRyser: An algorithm for sparse permanents

A sparse matrix has many zero entries; for permanents, and hence for Ryser, such entries do not have a practical impact on the final value. When the selected transversal contains even a single zero, its whole contribution becomes zero. Hence, such entries can be removed from the matrix and the remaining part can be stored with an appropriate data structure. The most widely used sparse-matrix data structures in the literature are *Compressed Row/Column Storage* (CRS and CCS). To store the sparse nonzero pattern, these structures use two arrays where the entries of the first one denote the start locations of each sparse row/column whose column/row ids are consecutively stored in the second array. An additional array, whose index structure is the same as the second array, is used to store the nonzero values. A toy matrix and its CRS/CCS representations are given in Figure 1. Using such a structure for permanent computation was proposed by Mittal and Al-Kurdi [16]. However, their approach computes the permanent via enumeration and can be used only for matrices when the number of all transversals is small. That is, their algorithm is based on (1), whereas the proposed algorithms SpaRyser and SkipPer in this work are based on (2).

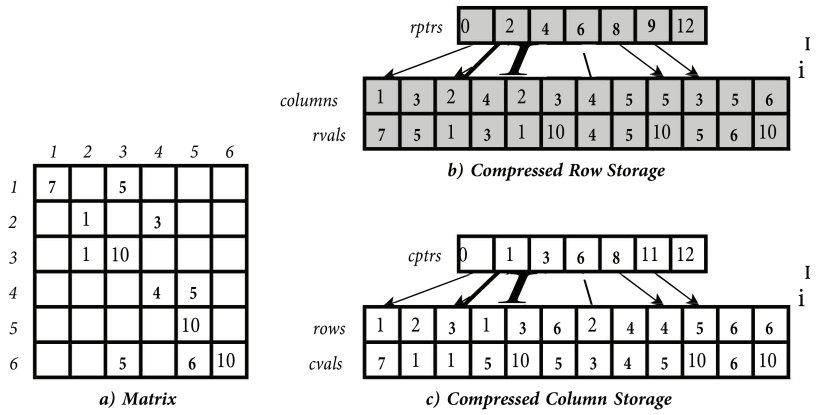


Figure 1: (a) A 6×6 matrix and its (b) CRS and (c) CCS representations.

For a dense/full matrix, regardless of the flipped bit, there will be n updates on the vector $\mathbf{x}$. However, for a sparse matrix, one can exploit the sparsity with CCS representation, since the zero entries, which have zero contribution on the $\mathbf{x}$ vector, can be skipped. Thanks to CCS, when the jth bit is flipped in the Gray code, the number of updates to the vector will exactly be equal to the number of nonzero elements on column j. Note that Ryser also performs the same number of useful updates, but it also performs updates with no impact. Hence, in terms of the number of floating point operations, using CRS and CCS can significantly increase the efficiency based on the level of sparsity in the matrix. The pseudocode of SpaRyser is given in Algorithm 2.

In addition to the exploitation of sparsity, SpaRyser performs two other techniques for fast permanent computation:

1. In Algorithm 1, Ryser, the inner loop (at line 11) has two statements. Using CRS and CCS in SpaRyser only reduces the number of $\mathbf{x}$ updates on the first line but the $\Theta(n)$ cost of the second line to compute *prod* is still there. The preliminary experiments showed that for a significant number of iterations of the main loop, i.e. for many g values, the contribution, and hence the value of the *prod*, can be zero in sparse

Algorithm 2 : SpaRyser

Input: $(rptrs, columns, rvals)$ - CRS of $\mathbf{A}$
$(cptrs, rows, cvals)$ - CCS of $\mathbf{A}$
Output: perm($\mathbf{A}$)

```
1: nzeros ← 0
2: for i = 1 to n do
3:     sum = 0
4:     for ptr = rptrs[i] to (rptrs[i + 1] − 1) do
5:         sum ← sum + rvals[ptr]
6:     x[i] ← rvals[rptrs[i + 1] − 1] − sum/2
7:     if x[i] = 0 then
8:         nzeros ← nzeros + 1
9: if nzeros > 0 then
10:    p ← ∏_{i=1}^{n} x[i]
11: else
12:    p ← 0
13: for g = 1 to 2^{n−1} − 1 do
14:    j ← log2(Gray_g ⊕ Gray_{g−1})
15:    s ← 2× Gray_g[j] −1
16:    for ptr = cptrs[j] to (cptrs[j + 1] − 1) do
17:        row ← rows[ptr]
18:        val ← cvals[ptr]
19:        if x[row] = 0 then
20:            nzeros ← nzeros − 1
21:        x[row] ← x[row] + (s × val)
22:        if x[row] = 0 then
23:            nzeros ← nzeros + 1
24:    if nzeros = 0 then
25:        prod ← 1
26:        for i = 1 to n do
27:            prod ← prod × x[i]
28:        p ← p + ((−1)^g × prod)
29: return p × (4 × (n mod 2) − 2)
```

matrices. However, even for these g values, Ryser performs all the multiplications. To get rid of these unnecessary operations, SpaRyser counts the number of zeros with a variable $nzeros$, and when a zero is detected, i.e. when $nzeros$ is positive, it does not compute the product and does not update p.

2. Both Ryser and SpaRyser use a binary-reflected Gray code. For an n-bit, binary-reflected Gray code, the jth bit changes 2^{n-j} times for $1 \leq j \leq n$. Hence, there is a significant imbalance on the numbers of bit flips for different bit locations. Such an imbalance yields an avenue for further optimization. To increase the performance of the algorithm, a preprocessing step is applied and the columns are ordered in increasing number of nonzeros. This ordering is expected to increase the performance, since sparser columns are processed more frequently compared to the denser ones. Clearly, the amount of improvement depends on the imbalance on the number of column nonzeros. This ordering scheme is called SortOrd.

4. SkipPer: Faster sparse permanents with Gray skipping and parallelization

With SpaRyser, the sparsity is exploited and the unnecessary multiplications are omitted. However, one can further avoid them by explicitly skipping these iterations, which will yield an even faster algorithm. That is, instead of checking a zero vector-product for every iteration, when a zero-contributing Gray code is detected, one can skip as many iterations as possible with a single jump. Nevertheless, the jumps must not skip a contributing iteration. To jump as long as possible, the proposed algorithm SkipPer uses the following lemma.

Lemma 1 *In an n-bit binary-reflected Gray code, the value of the jth bit of* Gray_g*, for $1 \leq j \leq n$, is*

$$\text{Gray}_g[j] = \begin{cases} 0 & \text{if } g < 2^{j-1} \\ \left(\lfloor \frac{(g-2^{j-1})}{2^j} \rfloor \bmod 2 \right) + 1 & \text{otherwise.} \end{cases}$$

The lemma is exploited as follows: when a zero entry $\mathbf{x}[i]$ is detected throughout the execution, at least one of the columns of $\mathbf{A}$ having a nonzero at its ith position needs to be processed to make $\mathbf{x}[i]$ nonzero. In Ryser's approach, a column is processed when the corresponding bit in the Gray code changes. Let $next(g, j)$ be the first iteration after the gth iteration where $\text{Gray}_g[j] \neq \text{Gray}_{next(g,j)}[j]$ for $1 \leq j \leq n$. Based on the lemma, this iteration can be computed as:

$$next(g, j) = \begin{cases} 2^j & \text{if } g < 2^j \\ g + 2^{j+1} - ((g - 2^j) \bmod (2^{j+1})) & \text{otherwise.} \end{cases}$$

For completeness, note that in an n-bit Gray code, when $next(g, j) \geq 2^n$, $\text{Gray}_g[j]$ is already the last value for the jth bit and it will never be changed.

Using $next(g, j)$, for a vector entry $\mathbf{x}[i] = 0$ encountered at iteration g, SkipPer computes

$$g^{(i)} = \min \{next(g, j) : a_{i,j} \neq 0\}$$

to find the earliest iteration that can make $\mathbf{x}[i] \neq 0$. Hence, assuming $\mathbf{x}$ is the current vector at iteration g, the first iteration that can produce a nonzero *prod* value is

$$next(g) = \begin{cases} g + 1 & \text{if } \mathbf{x}[i] \neq 0, \quad \text{for } 1 \leq i \leq n \\ \max\{g^{(i)} : \mathbf{x}[i] = 0\} & \text{otherwise.} \end{cases}$$

It is indeed safe to jump from iteration g to $next(g)$ for each $1 \leq g < 2^{n-1} - 1$ since all nonzero contributions of all the iterations will still be taken into account. However, one also needs an efficient mechanism to find the entries in vector $\mathbf{x}$ at iteration $next(g)$; due to skipping, the algorithm only computes $\mathbf{x}$ for iteration g and not for the iterations in between. Fortunately, this can be efficiently done by computing the bitwise difference of Gray_g and $\text{Gray}_{next(g)}$ as Algorithm 3 shows (lines 10–18).

By using the $next(.)$ function, SkipPer skips the zero-contributing transversals as much as possible. To increase the length of skips, an ordering is applied as a preprocessing step whose pseudocode is shown in Algorithm 4. As described in the previous section, the ordering technique in SpaRyser was sorting the columns based on their number of nonzeros. For SkipPer, a similar, sorting-like column ordering, SkipOrd, is used, which dynamically updates the degree counts (*degs* in the pseudocode) after each column selection in such

a way that the degree of each unselected column is always equal to the number of its rows that are not yet touched by previously chosen columns. That is, $degs[j]$ is the additional number of rows touched for the first time if column j is chosen. The ordering process always continues with the column having minimum $degs$ value (line 14). To ignore already chosen columns during this selection, $degs[j]$ is immediately set to ∞ once column j is selected (line 15 of Algorithm 4).

Algorithm 3 : SkipPer

Input: $(rptrs, columns, rvals)$ - CRS of $\mathbf{A}$
$(cptrs, rows, cvals)$ - CCS of $\mathbf{A}$
Output: $\texttt{perm}(\mathbf{A})$

1: **for** $i = 1$ to n **do**
2: $sum = 0$
3: **for** $ptr = rptrs[i]$ to $(rptrs[i+1] - 1)$ **do**
4: $sum \leftarrow sum + rvals[ptr]$
5: $\mathbf{x}[i] \leftarrow rvals[rptrs[i+1] - 1] - \frac{sum}{2}$
6: $p \leftarrow \prod_{i=1}^{n} \mathbf{x}[i]$
7: $g' \leftarrow 0$
8: $g \leftarrow 1$
9: **while** $g < 2^{n-1}$ **do**
10: $gr_{diff} = g \oplus g'$
11: **for** $j = 1$ to n **do**
12: **if** $gr_{diff}[j] = 1$ **then**
13: $gr_{diff}[j] \leftarrow 0$
14: $s \leftarrow 2 \times \text{Gray}_g[j] - 1$
15: **for** $ptr = cptrs[j]$ to $cptrs[j+1] - 1$ **do**
16: $row \leftarrow rows[ptr]$
17: $val \leftarrow cvals[ptr]$
18: $\mathbf{x}[row] \leftarrow \mathbf{x}[row] + (s \times val)$
19: $prod \leftarrow 1$
20: **for** $i = 1$ to n **do**
21: $prod \leftarrow prod \times \mathbf{x}[i]$
22: $p \leftarrow p + ((-1)^g \times prod)$
23: $g' \leftarrow g$
24: $g \leftarrow next(g)$
25: **return** $p \times (4 \times (n \bmod 2) - 2)$

Algorithm 4 : SkipOrd

Input: $\mathbf{A}$ ($n \times n$ matrix)
Output: $\mathbf{A}'$ (row/col permuted $\mathbf{A}$)

1: $rowPerm \leftarrow [.]$
2: $colPerm \leftarrow [.]$
3: $rowVisited \leftarrow [.]$
4: $degs \leftarrow [.]$
5: **for** $j = 1$ to n **do**
6: $degs[j] \leftarrow 0$
7: $rowVisited[j] \leftarrow$ **false**
8: **for** $i = 1$ to n **do**
9: **for** $j = 1$ to n **do**
10: **if** $a_{i,j} \neq 0$ **then**
11: $degs[j] \leftarrow degs[j] + 1$
12: $i \leftarrow 1$
13: **for** $j = 1$ to n **do**
14: $curCol \leftarrow \underset{\ell}{argmin}\{degs[\ell]\}$
15: $degs[curCol] \leftarrow \infty$
16: $colPerm[j] \leftarrow curCol$
17: **for** all ℓ s.t. $a_{\ell,curCol} \neq 0$ **do**
18: **if** $rowVisited[\ell] =$ **false then**
19: $rowVisited[\ell] \leftarrow$ **true**
20: $rowPerm[i] \leftarrow \ell$
21: $i \leftarrow i + 1$
22: **for** all k s.t. $a_{\ell,k} \neq 0$ **do**
23: **if** $degs[k] \neq \infty$ **then**
24: $degs[k] \leftarrow degs[k] - 1$
25: $\mathbf{A}' \leftarrow \mathbf{A}[rowPerm, colPerm]$

In SkipOrd, the steps where the rows are touched for the first time are also taken into account (lines 17–24 of Algorithm 4). Such a row ordering creates a structure with entries close to the diagonal in the lower triangular part of the matrix. This is expected to increase the performance; when a vector entry $\mathbf{x}[i] = 0$ with a large i (close to n) is encountered, to increase the length of the jumps, one hopes that $\mathbf{x}[i]$ keeps its value as long as possible. Since the Gray-code is unbalanced and the later bits are less frequently flipped, $\mathbf{x}[i]$ will be zero for a long time if flipping the earlier bits does not yield an update on $\mathbf{x}[i]$. This is what the row-ordering in SkipOrd provides and this is why this ordering is expected to make SkipPer faster.

4.1. Parallel permanent computation with SkipPer

As Algorithm 3 shows, there are loop workloads at two levels where the size of the iteration space is $2^{n-1}-1$ for the first level (line 9) and n for the second level (lines 11 and 20). Since 2^{n-1} much larger than n and a coarse-grain parallelism usually create less overhead, it is better to divide the permanent computation into a number of tasks through the outer loop. Hence, the threads will process different iterations, i.e. different chunks of consecutive Gray codes. Although it seems hard to start processing arbitrary chunks at first glance, by using the technique in SkipPer, the threads can skip all the iterations until the first iteration of any chunk. Hence, the previous approach can be used to divide all the iteration space into τ threads.

Let t_i be the ith thread for $1 \leq i \leq \tau$. To parallelize SkipPer, one can simply divide the iteration space into equisized chunks of size $c = \lfloor 2^{n-1}-1)/\tau \rfloor$ (the last thread may perform a few more iterations if τ does not divide 2^{n-1}). Each thread computes the first iteration of its chunk by computing $g^i_{first} = c \times (i-1)+1$ and process the Gray codes in the iteration space $[g^i_{first}, g^{i+1}_{first})$ where for completeness, $g^{\tau+1}_{first} = 2^{n-1}$. The $\mathbf{x}$ vector (as computed in line 5 of Algorithm 3) is copied to the private memory of each thread and the private copy $\mathbf{x}^i$ is updated based on g^i_{first} by t_i. After the initialization step, the threads process their chunks as in the sequential SkipPer. Since the parallelized variant is quite similar, a separate pseudocode is not provided.

One problem with this simple approach using static scheduling is that although each iteration seems to be equal in terms of its computational load, especially for sparse matrices, an unexpected number of iterations are skipped in a single chunk. Since the number of skipped iterations can greatly vary for different chunks, when static scheduling is used, the workload distribution among the threads can be highly irregular. To better balance the loads, one can use more, smaller chunks and assign the chunks to the idle threads one by one throughout the execution, i.e. perform dynamic scheduling.

5. Related work

There are a few studies on computing exact permanents for sparse matrices: Mittal and Al-Kurdi proposed an algorithm that enumerates all the permanents by exploiting the efficiency of CRS and CCS representations [16]. Their algorithm is efficient for very sparse matrices with a small number of transversals. When the number of transversals is large, which is usually the case in practice, the execution time increases significantly. Unlike their algorithm, SkipPer does not enumerate the transversals and runs along the same lines as Ryser's algorithms.

Forbert and Marx proposed an approximation algorithm for sparse permanents [22]. In the same study, they also presented a decomposition scheme based on the following equation:

$$\texttt{perm}\left(\begin{bmatrix} a & b & \mathbf{c} \\ \mathbf{d} & \mathbf{e} & \mathbf{A}' \end{bmatrix}\right) = \texttt{perm}\left(\begin{bmatrix} 0 & 0 & \mathbf{c} \\ \mathbf{d} & \mathbf{e} & \mathbf{A}' \end{bmatrix}\right) + \texttt{perm}\left(\begin{bmatrix} a\mathbf{e}+b\mathbf{d} & \mathbf{A}' \end{bmatrix}\right), \tag{3}$$

where for an $n \times n$ input matrix on the left side of equality, a and b are scalars, $\mathbf{c}$ is an $(n-2)$-dimensional row vector, $\mathbf{d}$ and $\mathbf{e}$ are $(n-1)$-dimensional column vectors, and $\mathbf{A}'$ is the $(n-1) \times (n-2)$ left-over matrix. This decomposition scheme was exploited by Liang et al. [23] and a hybrid algorithm was proposed, which recursively decomposes the matrices via (3) if there exists a row/column with less than or equal to four nonzeros. If this is not the case, i.e. when all the rows/columns have more than four nonzeros, the algorithm calls Ryser. This decomposition is also employed in the experiments of this work. The proposed algorithms in this study, especially SkipPer, can be used instead of Ryser in this hybrid algorithm.

The experiments of Liang et al. showed that when the density of the matrix, i.e. the ratio of the number of nonzeros to n^2, is 40%, the execution times of the hybrid algorithm and Ryser are comparable. They also reported that for a 25×25 matrix having 25% density, the execution time ratio of Ryser and the hybrid algorithm is 1.5. In the preliminary experiments, SkipPer alone, even without decomposition, provided much better speedups, especially when the sparsity was higher. Note that as n increases with a constant sparsity, the decomposition technique becomes less and less effective since the expected number of nonzeros in a row/column increases and becomes more than four. The hybrid algorithm in [23] was also used by Yue et al. [19], where the matrix was first partitioned into two for a more efficient permanent computation. Similar to the aforementioned studies, SkipPer can be used instead of the hybrid algorithm, or inside the hybrid algorithm, to improve the performance.

Since the decomposition can yield many matrices, a trivial parallelization strategy would be assigning all the final submatrices to a different thread/processor/node/etc. This strategy was considered in [24] with a load-balancing technique based on estimating the hardness of the permanent computation for each submatrix. Similar to the above, if decomposition is not applicable, one cannot divide the task into subtasks with this strategy. However, even if this is the case, the proposed parallelization of SkipPer can be employed. Furthermore, as the experiments will show, with dynamic scheduling this strategy yields almost linear speedups and hence it can be used without hesitation even when decomposition is possible.

Computing permanents of special matrix classes, e.g., (0,1)-matrices, has been investigated in the literature [18]. As mentioned before, the permanent of a (0,1)-matrix is related to the number of perfect matchings in the corresponding graph structure. A more recent study introduced a different approach specialized to compute the number of perfect matchings in complex, weighted, undirected graphs [25]. The approach in [25] does not directly exploit the sparsity; however, it uses an approach different than Ryser's while computing the permanent. Although SkipPer can work for any matrix, it can be specialized for (0,1)-matrices with a better memory layout and further optimized by adapting the techniques in [18] and [25].

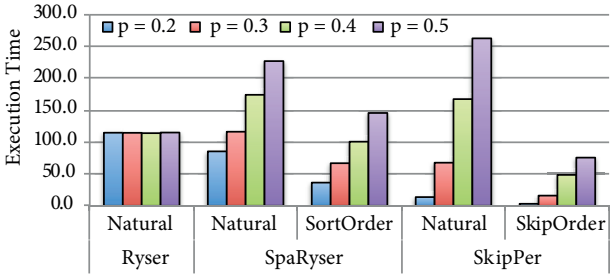
6. Experimental results

To analyze the performance of Ryser, SpaRyser, and SkipPer and measure the performance improvements obtained due to exploitation of sparsity, Gray skipping, and parallelization, various experiments are conducted on synthetic and real-life matrices. All the experiments are performed on a server running on 64-bit CentOS 6.5 equipped with 64GB RAM and two Intel Xeon E7-4870 v2 clocked at 2.30 GHz and having 15 cores each. Each core has a 32KB L1 and a 256KB L2 cache, and the size of the L3 cache is 30MB. For the multicore implementations, `C++` and `OpenMP 4.5` are used with `gcc 8.2.0` and `-O3` optimization flag.

6.1. Experiments with synthetic data

To generate synthetic data, the Erdös–Renyi model is employed; for an $n \times n$ matrix, each entry $a_{i,j}$ is independently chosen to be a nonzero with a constant probability p, a parameter to the model. Hence, each row/column is expected to have $p \times n$ nonzeros and the expected number of nonzeros is $p \times n^2$. The probability values $p = \{0.2, 0.3, 0.4, 0.5\}$ and the matrix dimensions $n = \{32, 34, 36\}$ are used in the experiments. For each $(n,\ p)$ tuple, 5 random matrices are created. Figure 2 (left) shows the execution times of the algorithms with different orderings for $n = 32$. Each bar is an average of five values, i.e. execution times for five matrices. On the right of the figure, the coefficients of variation (CV) are given for each bar. Note that the CV values

are usually at most 0.25 and more than 0.30 only when the average execution time is small. As the results show, SpaRyser is better than the naive algorithm Ryser especially when the sparsity is high. However, when $p = 0.5$, there is not much sparsity to exploit and SpaRyser becomes slower. As the results confirm, the impact of the ordering (SortOrder) on the performance of SpaRyser is significant.



	Ryser	SpaRyser		SkipPer	
p	Nat.	Nat.	Ord	Nat.	Ord.
0.2	0.01	0.28	0.29	0.32	0.39
0.3	0.01	0.10	0.23	0.20	0.33
0.4	0.00	0.05	0.12	0.15	0.25
0.5	0.01	0.08	0.01	0.09	0.18

Figure 2: The execution times (in seconds) for $n = 32$ and $p = \{0.2, 0.3, 0.4, 0.5\}$ of the algorithms Ryser, SpaRyser, and SkipPer with and without ordering on randomly generated matrices (left) and variations of coefficient for the execution times on five matrices used for each (n, p) pair (right). The bars labeled as *Natural* show the executions where no ordering is applied to the given matrix and the natural order is used.

The proposed algorithm SkipPer improves the performance drastically; although it can perform much worse when the matrix is unordered, with SkipOrder its performance improves significantly. Even for $p = 0.5$, SkipPer is 1.6× faster than Ryser. For $p = 0.2$, the speedup is around 40×. The speedups of SkipPer with SkipOrd over Ryser with different n and p values are given in Table 1.

Table 1: Speedups of to SkipPer over Ryser with different n and p values.

	p			
n	0.2	0.3	0.4	0.5
32	40.1×	10.8×	2.5×	1.6×
34	119.7×	11.2×	3.8×	1.6×
36	140.9×	13.2×	4.2×	1.6×

Table 2 compares the algorithms in a pairwise manner for $n = 32$ and $p = \{0.2, 0.3, 0.4, 0.5\}$. As described above, 20 matrices are generated for $n = 32$ with different p values. To compare two algorithms, for each matrix, the execution time of the former is divided into that of the latter and the averages of these 20 values are reported in the table. As the results confirm, SkipPer performs much better than the other two algorithms. Furthermore, SortOrd is a better ordering for SpaRyser, and SkipOrd yields around 20% better performance for SkipPer compared to SortOrd. Although better orderings may still exist, these two observations show the validity of the rationale behind the orderings designed specially for SpaRyser and SkipPer.

The performance of the proposed approach described in the previous section to parallelize SkipPer is evaluated with 16 threads and coarse-grain parallelism. The parallelism is employed first by dividing the iteration space of length $2^{n-1} - 1$ into 16 static chunks. The performance is also measured with much smaller chunks created and assigned by dynamic scheduling. The speedup results with 16 chunks, i.e. static scheduling (columns 10–13), and 512 chunks with dynamic scheduling (columns 6–9) are presented in Table 3. The results confirm that with smaller chunks, one can have almost linear speedup, i.e. between 14.4 and 15.2 with $\tau = 16$ threads. However, especially for sparser matrices, where the number of skipped iterations can

Table 2: Pairwise performance comparison of the algorithms for $n = 32$. For each row, the performances are normalized w.r.t. to the algorithm whose name is given in the leftmost column at that row. Each value in the table is the average of normalized execution times for 20 different executions since four different p values are used and five matrices are generated with each p.

		Ryser	SpaRyser			SkipPer		
Algorithm	Ordering	Nat.	Nat.	Sort Ord	Skip Ord	Nat.	Sort Ord	Skip Ord
Ryser	Natural	1.0	-	-	-	-	-	-
SpaRyser	Natural	0.9	1.0	-	-	-	-	-
	SortOrd	1.8	1.9	1.0	-	-	-	-
	SkipOrd	1.7	1.8	0.9	1.0	-	-	-
SkipPer	Natural	2.9	2.6	1.3	1.4	1.0	-	-
	SortOrd	11.0	9.5	4.7	5.0	3.4	1.0	-
	SkipOrd	13.7	12.1	5.9	6.3	4.1	1.2	1.0

greatly vary, the parallel efficiency is low with static scheduling. Columns 2-5 in the table present the speedups w.r.t. sequential Ryser when $\tau = 16$ threads are used. Overall, parallel SkipPer is around $600\times - 2000\times$ faster than Ryser and the speedups decrease with decreasing sparsity.

Table 3: Speedups of parallel SkipPer-SkipOrd with $\tau = 16$ threads: columns 2–5 show the improvement with dynamic scheduling w.r.t. the traditional Ryser. Columns 6–9 present the speedups w.r.t. the sequential algorithm. To show the benefits of dynamic scheduling, columns 10–13 show the speedup values with static scheduling.

	Speedup w.r.t. seq. Ryser (dyn., #chunks = 512)				Speedup w.r.t. seq. SkipPer-SkipOrd (dyn., #chunks = 512)				Speedup w.r.t. seq. SkipPer-SkipOrd (stat., #chunks = 16)			
	p				p				p			
n	0.2	0.3	0.4	0.5	0.2	0.3	0.4	0.5	0.2	0.3	0.4	0.5
32	605	161	37.2	23.7	15.2	15.1	14.8	15.3	6.6	10.3	11.1	11.7
34	1790	172	57.4	24.4	14.4	15.2	15.2	15.5	6.1	9.8	9.8	10.7
36	2050	180	49.2	24.2	14.6	15.0	15.5	15.6	6.8	9.0	10.6	13.7

6.2. Experiments with real-life matrices

In addition to synthetic matrices, five real-life matrices are used to further evaluate the performance of the proposed algorithms. The properties of the matrices and the results of the experiments are given in Table 4. For each matrix, the decomposition technique (3) in [22, 23] is used to recursively decompose the matrix into two. The decomposition is performed until there is no row/column having less than 5 nonzeros or $n \leq 32$. Each algorithm is run with a 30-min time limit and computes the permanent of as many submatrices as it can. For each algorithm alg, Table 4 reports the number of submatrices alg processes, the number of submatrices on which alg is the fastest, and alg's average relative performance with respect to that of SkipPer-SkipOrd. The relative performance is computed only over the matrices processed by alg. Since SkipPer is the fastest algorithm and the time limit is the same for all the algorithms, SkipPer's execution times are known on these matrices. An algorithm is considered best for a matrix if its execution time does not exceed $1.05\times$ the minimum permanent computation time for that matrix. As the table shows, although there are matrices for

which SpaRyser is better than SkipPer, i.e. 6 submatrices for `c100` and one submatrix for `chesapeake`, the average performance of SkipPer is much better than the other two algorithms.

Table 4: This table shows the number of matrices processed, number of matrices with the fastest execution time, and the relative performance with respect to SkipPer for each algorithm. For each row, several submatrices are generated by using the decomposition given in Eq. 3. The last two columns present the speedup values for SkipPer with static and dynamic scheduling, respectively.

	Prop.		Ryser			SpaRyser-SortOrd			SkipPer-SkipOrd			SkipPer-SkipOrd $\tau = 16$, speedup	
Matrix	n	nnz	Solved	Best	Perf.	Solved	Best	Perf.	Solved	Best	Perf.	static	dynamic
`bfwb62`	62	202	15	-	37.8	49	-	11.7	520	520	1.0	7.1×	14.7×
`c100`	100	300	15	-	4.3	38	14	1.2	42	36	1.0	14.9×	15.9×
`cage5`	37	233	10	-	22.2	10	-	6.2	10	10	1.0	6.7×	13.6×
`chesapeake`	39	340	15	-	11.8	44	7	5.5	84	83	1.0	5.6×	14.8×
`will57`	57	281	15	-	69.2	64	-	13.4	609	609	1.0	8.3×	14.7×

In addition to the individual performances of Ryser, SpaRyser, and SkipPer, the last two columns of the table present the speedups for SkipPer both with static and dynamic scheduling and $\tau = 16$ w.r.t. sequential SkipPer. The results confirm the previous experiments; static scheduling can fail to balance the load distribution, whereas parallelization with dynamic scheduling yields close to linear speedups.

7. Conclusion and future work

In this work, a parallel algorithm for efficient computation of permanents of sparse matrices is proposed. Compared to the original algorithm Ryser, around 2000× speedups are obtained for matrices with high sparsity. The proposed techniques can be used to improve the performance of the existing solutions in the literature since at one point they almost all use Ryser. The best algorithm, SkipPer, uses Gray skipping, whose impact heavily depends on the nonzero pattern in the matrix. Although SkipOrder's effectiveness is shown empirically, it would be an interesting future work to generate the best ordering that enables the algorithm to cover the iteration space with the fewest number of skips and yield the best theoretical complexity. In addition, one can also combine the current work with the technique proposed by Servedio and Wan [17], which adds a row/column to the original matrix to obtain an augmented matrix to encounter many zero terms in Ryser. In addition to these, it could be interesting to analyze further optimizations within SkipPer for matrices with a specific pattern/structure such as structurally and numerically symmetric matrices.

Another possible research avenue is parallelizing SkipPer on many-core devices such as GPUs. The problem is harder indeed; balancing the load with nonuniform and dynamic task sizes is not a straightforward job when thousands of cores are available. However, the Gray skipping technique as well as the proposed ordering technique can be easily adopted since they create independent tasks with possibly varying but small computational loads; one can easily increase the number of chunks to millions. Unfortunately, a possible architectural problem with this approach can be the existence of intervals with no impact on the permanent. Especially when these intervals are distributed evenly among all the intervals, a naive task-to-thread assignment will weaken the utilization of the GPU. This problem can be solved by computing the permanent in a warp-centric approach instead of a thread-centric one. In the future, this path is planned to be investigated to have an efficient GPU parallelization of SkipPer.

References

[1] Kılıç E, Tasçı D. On the permanents of some tridiagonal matrices with applications to the Fibonacci and Lucas numbers. Rocky Mountain Journal of Mathematics 2007; 37: 1953-1969. doi: 10.1216/rmjm/1199649832

[2] Balakrishnan N. Permanents, order statistics, outliers, and robustness. Revista Matematica Complutense 2007; 20 (1): 7-107. doi: 10.5209/rev_REMA.2007.v20.n1.16528

[3] Aaronson S, Arkhipov A. The computational complexity of linear optics. In: Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing; New York, NY, USA; 2011. pp. 333-342.

[4] Wu J, Liu Y, Zhang B, Jin X, Wang Y et al. A benchmark test of boson sampling on Tianhe-2 supercomputer. National Science Review 2018; 5 (5): 715-720. doi: 10.1093/nsr/nwy079

[5] Narahara M, Tamaki K, Yamada R. Application of permanents of square matrices for DNA identification in multiple-fatality cases. BMC Genetics 2013; 14 (1): 72. doi: 10.1186/1471-2156-14-72

[6] Mahajan M, Raghavendra RBV, Sreenivasaiah K. Monomials, multilinearity and identity testing in simple read-restricted circuits. Theoretical Computer Science 2014; 524: 90-102. doi: 10.1016/j.tcs.2014.01.005

[7] Minc H. Permanents (Encyclopedia of Mathematics and Its Applications). Cambridge, UK: Cambridge University Press, 1984.

[8] Valiant LG. The complexity of computing the permanent. Theoretical Computer Science 1979; 8 (2): 189-201. doi: 10.1016/0304-3975(79)90044-6

[9] Jerrum M, Sinclair A, Vigoda E. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. Journal of the ACM 2004; 51 (4): 671-697. doi: 10.1145/1008731.1008738

[10] Štefankovič D, Vigoda E, Wilmes J. On counting perfect matchings in general graphs. In: Theoretical Informatics, LATIN 2018; Cham, Switzerland; 2018. pp. 873-885. doi: 10.1007/978-3-319-77404-6_63

[11] Gurvits L, Samorodnitsky A. Bounds on the permanent and some applications. In: FOCS '14 Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science; Washington, DC, USA; 2014. pp. 90-99. doi: 10.1109/FOCS.2014.18

[12] Linial N, Samorodnitsky A, Wigderson A. A deterministic strongly polynomial algorithm for matrix scaling and approximate permanents. Combinatorica 2000; 20: 545-568. doi: 10.1007/s004930070007

[13] Dufossé F, Kaya K, Panagiotas I, Uçar B. Scaling Matrices and Counting the Perfect Matchings in Graphs. Research Report RR-9161. Inria Grenoble Rhône-Alpes, 2018.

[14] Anari N, Gurvits K, Gharan SO, Saberi A. Simply exponential approximation of the permanent of positive semidefinite matrices. In: 58th IEEE Annual Symposium on Foundations of Computer Science; Berkeley, CA, USA; 2017. pp. 914-925.

[15] Eldar L, Mehraban S. Approximating the permanent of a random matrix with vanishing mean. In: 59th IEEE Annual Symposium on Foundations of Computer Science; Paris, France; 2018. pp. 23-34.

[16] Mittal RC, Al-Kurdi A. Efficient computation of the permanent of a sparse matrix. International Journal of Computer Mathematics 2001; 77 (2): 189-199. doi: 10.1080/00207160108805061

[17] Servedio S, Wan A. Computing sparse permanents faster. Information Processing Letters 2005; 96 (3): 89-92. doi: 10.1016/j.ipl.2005.06.007

[18] Bax E, Franklin J. A permanent algorithm with $\exp[\omega(n^{1/3}/2\ln n)]$ expected speedup for 0-1 matrices. Algorithmica 2008; 32: 157-162. doi: 10.1007/s00453-001-0072-0

[19] Yue B, Liang H, Bai F. Improved algorithms for permanent and permanental polynomial of sparse graph. Communications in Mathematical and in Computer Chemistry 2013; 69: 831-842.

[20] Ryser HJ. Combinatorial Mathematics. New York, NY, USA: Mathematical Association of America, 1963, doi: 10.5948/UPO9781614440147

[21] Nijenhuis A, Wilf HS. Combinatorial Algorithms. New York, NY, USA: Academic Press, 1978.

[22] Forbert H, Marx D. Calculation of the permanent of a sparse positive matrix. Computer Physics Communications 2003; 150 (3): 267-273. doi: 10.1016/S0010-4655(02)00683-5

[23] Liang H, Huang S, Bai F. A hybrid algorithm for computing permanents of sparse matrices. Applied Mathematics and Computation 2006; 172 (2): 708-716. doi: 10.1016/j.amc.2004.11.020

[24] Wang L, Liang H, Bai F, Huo Y. A load balancing strategy for parallel computation of sparse permanents. Numerical Linear Algebra with Applications 2011; 19: 1017-1030. doi: 10.1002/nla.1844

[25] Björklund A, Gupt B, Quesada N. A faster Hafnian formula for complex matrices and its benchmarking on a supercomputer. Journal of Experimental Algorithmics 2019; 24 (1): 1.11:1-1.11:17. doi: 10.1145/3325111