Received 3 April 2023, accepted 1 May 2023, date of publication 3 May 2023, date of current version 10 May 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3272864

RESEARCH ARTICLE

Parallel Heuristic Methods to Accelerate Best Equivocation Code Generation

YANCHEN LI[1], (Student Member, IEEE), KE ZHANG[2], AND FUMIHIKO INO[1], (Member, IEEE)
[1]Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565-0871, Japan
[2]School of Information Engineering, Wuhan University of Technology, Wuhan, Hubei 430070, China

Corresponding author: Yanchen Li (yc-li@ist.osaka-u.ac.jp)

This work was supported in part by the "Program for Leading Graduate Schools" of the Ministry of Education, Culture, Sports, Science and Technology, Japan, and the Japan Society for the Promotion of Science KAKENHI under Grant 23H03371; and in part by the National Nature Science Foundation of China under Grant 61801340.

ABSTRACT In this paper, we propose parallel heuristic methods to accelerate the generation of (n, m) best equivocation code (BEC), where n and m are code and message lengths, respectively. The proposed dynamic programming (DP) method and greedy method extend a previous heuristics method by reducing the time complexity of the code generation process. The DP method produces the same codes as the previous method but incurs an overhead for data reuse. In contrast, the greedy method avoids this overhead but generates slightly different codes due to its heuristic approach. We parallelize the proposed methods by exploiting coarse-grained and fine-grained parallelisms, which achieve further acceleration on multicore CPU and graphics processing unit (GPU) systems, respectively. Experimental results demonstrate that the proposed DP and greedy methods reduce the sequential generation time to a quarter, as indicated by theoretical complexity analysis. In addition, the parallel implementation achieves linear speedup on a multicore CPU system, and the GPU implementation realizes coalesced memory accesses, resulting in $17\times$ acceleration over the eight-core CPU implementation. We found that the greedy method produced different codes that differ from the previous and DP methods; however, the generated codes had higher equivocation rates than those generated by a naive random method. We believe that the proposed parallel methods can effectively accelerate BEC generation for large m and n values, especially with larger values of n relative to m.

INDEX TERMS Dynamic programming, GPU, greedy algorithm, multicore CPU, parallel processing, syndrome coding.

I. INTRODUCTION

Wyner's wiretap channel [1] is a typical model deployed to realize physical layer security [2]. Physical layer security exploits the physical properties of the communication channel to secure confidential information against an eavesdropper. This approach is useful for establishing security in wireless networks with noisy channels [3]. Compared to well-known public key cryptography, which relies on a computationally infeasible problem, physical layer security analytically secures confidential information based on information theory regardless of the eavesdropper's computational power. In addition, physical layer security is not susceptible to key leakage, which is a drawback of public key cryptography.

The wiretap channel model assumes that a sender transmits confidential messages to a legitimate receiver in the presence of an eavesdropper. Wyner proposed a syndrome coding scheme to secure messages for a specific case where the main channel (i.e., between the sender and receiver) is noiseless, and the channel to the eavesdropper is a binary symmetric channel. Here the eavesdropper receives incorrect messages that have bits flipped with crossover probability α. Similar to Wyner's work, many studies have used linear codes with syndrome coding to accept all channels as noisy channels [4], achieve network security [5], and achieve secrecy capacity [6].

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

Zhang et al. [7] proposed the best equivocation code (BEC) for the syndrome coding scheme. One advantage of the BEC is that it has the highest *equivocation rate* [8] (an information secrecy metric) for a given code of length n and *code rate* m/n, where m is the length of the messages to be encoded. The equivocation rate is a measure of how much uncertainty the eavesdropper has. Zhang et al. [7] also presented a heuristic BEC generation method and demonstrated experimentally that the generated BECs typically provide greater secrecy than the best known error-correcting codes [9], [10] with equal parameter values for n and m [11].

However, the code generation cost is a drawback of the BEC. The time complexity of code generation is $\mathcal{O}(n^2 4^m)$, which restricts the code generation for both large m and n values. In fact, our preliminary estimation indicates that code generation for $(n, m) = (100, 25)$ requires more than one month on a single CPU core, which is simply unacceptable for practical applications. Thus, accelerating the BEC generation is necessary to generate codes with high equivocation rates.

Accelerated code generation contributes to achieving both high equivocation rates and high code rates. Given a fixed message length m, the greater the code length n, the higher the equivocation rate and the lower the code rate. In contrast, given a fixed code length n, the equivocation and code rates will increase with increasing message length m. As a result, BECs with both larger m and n values are required to realize high equivocation and high code rate.

In this paper, we focus on accelerating BEC generation with large code length n and message length m to achieve high equivocation and code rates. We propose a dynamic programming (DP) method and a greedy method that extends the previous method [7] by reducing the time complexity of BEC generation. The proposed DP method produces the same codes as the previous method but incurs overhead for data reuse. In contrast, the proposed greedy method avoids this overhead but generates codes that differ slightly compared to those generated by the previous method. In addition, the proposed methods are parallelized by exploiting coarse-grained and fine-grained parallelisms, which achieve further acceleration on multicore CPU and GPU systems, respectively.

The remainder of this paper is organized as follows. Section II presents work related to the acceleration of code generation and the calculation of the equivocation rate. Section III discusses preliminaries, including the underlying syndrome coding scheme and the previous method [7], which is the basis of the proposed methods. Section IV describes the proposed methods, and Section V explains how the proposed methods are parallelized. Experimental results are presented in Section VI. Finally, the paper is concluded in Section VII, including suggestions for future work.

II. RELATED WORK

Generally, practical linear codes are defined with a generator polynomial that generates codewords. For example, the binary Bose—Chaudhuri—Hocquenghem (BCH) code [12], [13] with $(n, m) = (15, 4)$ and a distance of at least five has the generator polynomial $G(x) = x^8 + x^7 + x^6 + x^4 + 1$. Thus, the generator polynomial gives the definition for the BCH codes, and there is no need to search codes. In contrast, a corresponding definition for BECs has not been presented, which motivates us to accelerate BEC generation, i.e., finding the code with the highest equivocation rate for the given values of m and n.

The performance of BEC generation primarily relies on the evaluation of equivocation rates. To eliminate this performance bottleneck, Zhang et al. [14] proposed a method that generates random codes rather than BECs, and they realized an effective reduction of the computational costs. However, the equivocation rates were reduced due to the random solution. Pfister et al. [15] presented a Monte Carlo simulation approach to estimate the precise amount of equivocation for coset-based wiretap codes. Their approach is similar to Zhang et al. [14] in terms of deploying a random simulation. In contrast, rather than employing random approaches, the proposed solution employs a heuristic approach to generate BECs in a feasible time.

Another approach to address the performance issue involves incorporating restrictions on the deployed codes such that their equivocation rates can be obtained precisely with reduced computational costs. For example, Harrison and Bloch [16] demonstrated that the equivocation ensured by coset coding over a binary erasure wiretap channel could be calculated precisely using the knowledge of the generator matrix. Harrison [17] also extended the work [16] to present exact expressions for the equivocation. In the current study, we focus on reducing the time complexity of the equivocation rate calculation using algorithmic techniques, such as DP and greedy methods. Note that further improvements may be realized by exploiting the advantages of coding theory techniques [18].

In terms of BEC generation, Al-Hassan et al. [19] proposed a variant of Zhang's method [7]. The variant generates BECs by extending the code length n by two rather than one, which produces higher equivocation rates than the original method [7]. In addition, they proposed a code generation method [20] for BECs with the highest minimum distance. Note that all of these variants can be accelerated using the parallel methods proposed in this paper because these variants are sequential algorithms with a structure that is similar to that of Zhang's method [7].

III. PRELIMINARIES

In this section, we provide a brief overview of coding theory to facilitate a general understanding of linear codes and BECs.

A. LINEAR CODE

A linear code is an error-correcting code that constructs a codeword from a linear combination of codewords. Here, an (n, k) code encodes a k-bit message $\mathbf{x} \in \{0, 1\}^k$ into an n-bit codeword $\mathbf{x}G \in \{0, 1\}^n$, where G is the $k \times n$ generator matrix. Then, code C is given by set $C = \{\mathbf{x}G \mid \mathbf{x} \in \{0,1\}^k\}$. For a linear code C, we have $\forall \mathbf{y}_1, \mathbf{y}_2 \in C \Rightarrow \mathbf{y}_1 + \mathbf{y}_2 \in C$.

An n-bit row vector $\mathbf{y} \in \{0,1\}^n$ is a codeword of C if and only if the following holds:

$$\mathbf{y}H^\top = \mathbf{0}, \tag{1}$$

where H is the parity check matrix for C. The parity check matrix H is an $(n-k)\times n$ matrix that can be computed from the generator matrix G. In the following, we assume that H is in the standard form $H = [I_{n-k} \mid P]$, where I_{n-k} is the $(n-k)\times(n-k)$ identity matrix, and $P \in \{0,1\}^{(n-k)\times k}$ is an $(n-k)\times k$ matrix. For any data $\mathbf{y} \in \{0,1\}^n$, $\mathbf{y}H^\top \in \{0,1\}^{n-k}$ is referred to as *the syndrome* of $\mathbf{y}$. The receiver can detect errors by computing the syndrome $\mathbf{y}H^\top$ for the received data $\mathbf{y}$, and the receiver will obtain $\mathbf{y}H^\top \neq \mathbf{0}$ if $\mathbf{y}$ includes any errors.

Here, h_{ij} is the element of matrix H located at the i-th row and the j-th column, where $0 \le i < n-k$ and $0 \le j < n$. Notice here that we deploy zero-based numbering for indexing elements in a matrix. To simplify the notation for matrix H, we represent the matrix in vector form:

$$H = (\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{n-1}), \tag{2}$$

where $\mathbf{h}_j$ represents the j-th column of H. Note that the j-th column $\mathbf{h}_j$ can be further represented as an integer value h_j by summing all elements of the column: $h_j = \sum_{i=0}^{n-k-1} h_{ij}2^i$. For example, the parity check matrix

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{3}$$

can be represented in the following vector form:

$$H = (1, 2, 4, 8, 6, 11, 15, 5). \tag{4}$$

B. BEST EQUIVOCATION CODE (BEC)

The BEC [7] is based on a syndrome coding scheme designed for the wiretap channel model. As mentioned previously, the syndrome of length $n-k$ is used for error detection in error-correcting codes. In contrast, the goal of syndrome coding is to embed a confidential message of length $n-k$ using an (n,k) linear code C. In the following, we use $m = n-k$ to represent the message length.

There are two restrictions between the code length n and message length m. The first is $n > m$ because the code length must be greater than the message length. The second restriction is $n < 2^m$. This ensures the linear independence of the columns in matrix H, which is required for linear coding.

Figure 1 shows an overview of the (n,m) BEC scheme [7]. This scheme employs an (n,k) linear code C to encode an m-bit message $\mathbf{x}$ as an n-bit vector $\mathbf{v}$ such that

$$\mathbf{x} = \mathbf{v}H^\top. \tag{5}$$

For each message, the scheme also generates a random n-bit codeword $\mathbf{y} \in C$ from a random, uniformly distributed k-bit vector $\mathbf{r} \in \{0,1\}^k$:

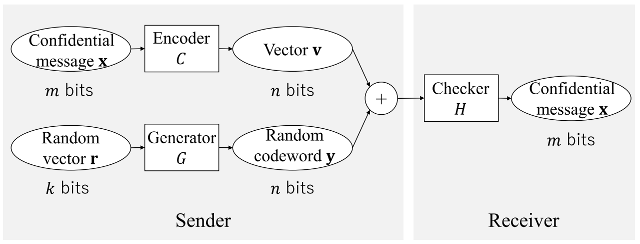


FIGURE 1. Overview of the (n, m) BEC scheme, where $m = n - k$, based on an (n, k) linear code C.

$$\mathbf{y} = \mathbf{r}G. \tag{6}$$

Then, the sender transmits an n-bit vector $\mathbf{v}+\mathbf{y}$ to the receiver.

The received vector is identical to the transmitted vector because the main channel is noise-free in the wiretap channel model. According to Eqs. (1) and (5), the receiver can decode the received vector as follows:

$$(\mathbf{v}+\mathbf{y})H^\top = \mathbf{x} + \mathbf{0} = \mathbf{x}. \tag{7}$$

In contrast, the eavesdropper's channel produces errors in the wiretap channel model. Here, let $\mathbf{e} \in \{0,1\}^n$ be the error vector, which represents an error pattern among 2^n possible patterns. Given the crossover probability α, an error pattern $\mathbf{e}$ occurs with probability $\alpha^{w(\mathbf{e})}(1-\alpha)^{n-w(\mathbf{e})}$, where $w(\mathbf{e})$ is the Hamming weight of $\mathbf{e}$. The eavesdropper decodes the received vector $\mathbf{v}+\mathbf{y}+\mathbf{e}$ as follows:

$$(\mathbf{v}+\mathbf{y}+\mathbf{e})H^\top = \mathbf{x} + \mathbf{s}, \tag{8}$$

where $\mathbf{s} = \mathbf{e}H^\top \in \{0,1\}^m$ represents the syndrome for the error vector $\mathbf{e}$. Thus, in the wiretap channel model, the syndrome can be used as an error amplifier rather than an error checker.

Zhang et al. [7] demonstrated that the syndrome $\mathbf{s} \in \{0,1\}^m$ for $\mathbf{e} = (e_0, e_1, \ldots, e_{n-1}) \in \{0,1\}^n$ can be calculated as follows:

$$\mathbf{s} = \mathbf{e}H^\top = \bigoplus_{j=0}^{n-1} \delta(e_j - 1)\mathbf{h}_j^\top, \tag{9}$$

where δ is the Dirac function, and $\oplus$ is the bitwise modulo-2 addition, i.e., the bitwise XOR operation. Eq. (9) implies the following: (1) an error pattern generates one of the 2^m syndromes, (2) different error patterns may produce the same syndrome, and (3) the parity check matrix H determines the distribution of syndromes.

C. SECRECY OF BEC

Assume that we have a syndrome encoding scheme whose parity check matrix is H. Given an estimated message $\bar{\mathbf{x}}$ for the original message $\mathbf{x}$, the equivocation rate E for the syndrome encoding scheme can be measured by the eavesdropper decoder output equivocation:

$$E(H) = \mathcal{H}(\mathbf{x}|\bar{\mathbf{x}}), \tag{10}$$

where $\mathcal{H}(\mathbf{x}|\bar{\mathbf{x}})$ represents the conditional entropy of $\mathbf{x}$ given $\bar{\mathbf{x}}$. According to Eq. (8), the equivocation rate can be obtained as follows by calculating the probabilities of all possible syndromes:

$$\begin{aligned}\mathcal{H}(\mathbf{x}|\bar{\mathbf{x}}) &= \mathcal{H}(\mathbf{s}) \\ &= -\sum_{j=0}^{2^m-1} p_j \log p_j,\end{aligned} \tag{11}$$

where p_j is the probability of the j-th syndrome $\mathbf{s}_j$ for the given parity check matrix H. Thus, the secrecy of BECs, $E(H)$, relies on the parity check matrix H, which determines the distribution of syndromes.

Zhang et al. [7] proposed a recursive greedy method to evaluate the secrecy of a BEC. Their method was based on a code extension approach in which the i-th recursive step generates codes of length $i+1$ from those of length i, where $0 \leq i < n$. A code of length $i+1$ can be obtained from the first i-th columns of the parity check matrix H. Here, let $p_j^{(i)}$ be the probability of the j-th syndrome, where $0 \leq j < 2^m$, that can be generated from the code of length $i+1$. This probability can be calculated according to the following recurrence relation:

$$p_j^{(i)} = \begin{cases} 1-\alpha, & i=0, j=0, \\ \alpha, & i=0, j=1, \\ 0, & i=0, j>1, \\ (1-\alpha)p_j^{(i-1)} + \alpha p_{j\oplus h_i}^{(i-1)}, & i>0. \end{cases} \tag{12}$$

For example, when $i=0$, there are only two error patterns, $\mathbf{e} \in \{\mathbf{0}, \mathbf{1}\}$, which produce the first ($j=0$) and second ($j=1$) syndromes, where $\mathbf{e}=\mathbf{0}$ and $\mathbf{e}=\mathbf{1}$ correspond to the errorless and one-bit error cases that occur with probabilities of $1-\alpha$ and α, respectively. The remaining syndromes ($j>1$) will not be produced when $i=0$. In contrast, when $i>0$, we have two cases according to whether the extended i-th bit causes an error or not. The first term of Eq. (12) ($i>0$) corresponds to the no error case, in which the j-th syndrome at length i reappears at length $i+1$. The second term covers the error case, where the extended i-th bit causes an error. This error bit produces a different syndrome as indicated by Eq. (9). In other words, the extended column $\mathbf{h}_i$ of the parity check matrix H increases the probability of the $(j \oplus h_i)$-th syndrome, where $0 \leq j < 2^m$.

D. BEC GENERATION

In the BEC generation problem, the goal is to find a parity check matrix H that maximizes the equivocation rate for the given m and n:

$$\underset{H\in\{0,1\}^{m\times n}}{\arg\max} = E(H). \tag{13}$$

Here, H is given in the standard form; thus, Eq. (13) can be rewritten as follows:

$$\underset{P\in\{0,1\}^{m\times(n-m)}}{\arg\max} = E([I_m \mid P]). \tag{14}$$

Eq. (14) indicates that an exhaustive search requires the worst time complexity of $\mathcal{O}(2^{m(n-m)})$ to find the highest equivocation rate for the given m and n.

As described in Section I, we focus on accelerating BEC generation for the given m and n. Note that finding appropriate values for m and n is also a practical issue when using BECs; however, this is beyond the scope of the current study.

E. PREVIOUS HEURISTIC METHOD

Zhang et al. [7] proposed a variant of a greedy strategy to reduce the search space. Before describing that method, we first introduce the greedy strategy. Here, let $H^{(i)}$ denote a parity check matrix of length $i+1$, where $0 \leq i < n$. As explained in Section III-C, the idea of the greedy strategy is to generate a parity check matrix $H^{(i)}$ by extending $H^{(i-1)}$ with an additional column:

$$H^{(i)} = [H^{(i-1)} \mid Q], \tag{15}$$

where $m \leq i < n$ and $Q \in \{0,1\}^{1\times m}$ is the column to be added. According to this greedy strategy, the problem of n-bit BEC generation can be expressed as follows:

$$\underset{H\in\mathbb{H}^{(n-1)}}{\arg\max}\, E(H), \tag{16}$$

where $\mathbb{H}^{(n-1)}$ is the set of parity check matrices to be searched at the $(n-1)$-th recursive step. The set $\mathbb{H}^{(i)}$ at the i-th step can be written by a recurrence relation as follows:

$$\mathbb{H}^{(i)} = \begin{cases} \{I_m\}, & 0 \leq i < m, \\ \{[H^{(i-1)} \mid Q], \\ \quad H^{(i-1)} \in \underset{H\in\mathbb{H}^{(i-1)}}{\arg\max}\, E(H), \\ \quad Q \in \{0,1\}^{1\times m}\}, & m \leq i < n. \end{cases} \tag{17}$$

The greedy method considers the best matrix for the extension at each step. Under the assumption that $\forall i\ (m \leq i < n)$, $|\arg\max_{H\in\mathbb{H}^{(i-1)}} E(H)| = 1$, the greedy method significantly reduces the time complexity from $\mathcal{O}(2^{m(n-m)})$ to $\mathcal{O}((n-m)2^m)$ because the search space at each of $n-m$ steps is given by $\mathcal{O}(2^m)$.

In contrast, Zhang's previous method [7] reduces equivocation rate loss by considering the top 10 matrices rather than the best matrix. In other words, their method replaces the arg max operator with an operator that returns the arguments of the top 10 values. Algorithm 1 shows the previous recursive method [7], which outputs a set $\mathbb{H}^{(n-1)}$ of the top t BECs from four inputs: (1) message length m, (2) code length n, (3) crossover probability α, and (4) the number t of matrices saved in each step. The algorithm first initializes the set $\mathbb{H}^{(m-1)}$ of BECs with the minimum set $\{I_m\}$ (line 1) and then iteratively extends the set by adding a column to the parity check matrix. The loop structure comprises the following four parts.

1) Matrix extension (Algorithm 1, line 4). The top t matrices saved in the previous iteration are extended by adding a column. All of the extended matrices are added to set $\mathbb{H}$ for evaluation.

Algorithm 1 BECgeneration (m, n, α, t)

Input: message length m, code length n, crossover probability α, and the number t of matrices to save.
Output: A set $\mathbb{H}^{(n-1)}$ of top t BECs of length n.

```
1: $\mathbb{H}^{(m-1)} \leftarrow \{I_m\}$;                                        ▷ Eq. (17)
2: for $i \leftarrow m$ to $n-1$ do
3:     $\mathbb{E} \leftarrow \emptyset$;                                           ▷ Emptyset
4:     $\mathbb{H} \leftarrow \{[H^{(i-1)} \mid Q], H^{(i-1)} \in \mathbb{H}^{(i-1)}, Q \in \{0,1\}^{1\times m}\}$;   ▷ Eq. (17)
5:     for each $H \in \mathbb{H}$ do                                              ▷ Loop 1
6:         $\mathbf{p} \leftarrow$ DistrCalc$(m, \alpha, H, i)$;                     ▷ Alg. 2
7:         $\mathbb{E} \leftarrow \mathbb{E} \cup \{$ EquivRateEval$(m, \mathbf{p})$ $\}$;   ▷ Alg. 3
8:     end for
9:     $\mathbb{H}^{(i)} \leftarrow$ top $t$ matrices in $\mathbb{H}$;              ▷ According to $\mathbb{E}$
10: end for
```

Algorithm 2 DistrCalc (m, α, H, l)

Input: message length m, crossover probability α, parity check matrix H, current column index l in H.
Output: Probability distribution $\mathbf{p}$ of syndromes, where $\mathbf{p} = (p_0, p_1, \ldots, p_{2^m-1})$.

```
1: $p_0^{(0)} \leftarrow 1-\alpha$; $p_1^{(0)} \leftarrow \alpha$;                   ▷ Eq. (12)
2: for $i \leftarrow 1$ to $l$ do
3:     for $j \leftarrow 0$ to $2^m - 1$ do                                        ▷ Loop 2
4:         $p_j^{(i)} \leftarrow (1-\alpha)p_j^{(i-1)} + \alpha p_{j\oplus h_i}^{(i-1)}$;   ▷ Eq. (12)
5:     end for
6: end for
7: $\mathbf{p} \leftarrow \mathbf{p}^{(l)}$;
```

Algorithm 3 EquivRateEval $(m, \mathbf{p})$

Input: message length m and probability distribution $\mathbf{p}$ of syndromes, where $\mathbf{p} = (p_0, p_1, \ldots, p_{2^m-1})$.
Output: Equivocation rate E.

```
1: $E \leftarrow 0$;
2: for $j \leftarrow 0$ to $2^m - 1$ do
3:     $E \leftarrow E - p_j \log p_j$;                                             ▷ Eq. (11)
4: end for
```

2) Probability distribution calculation (Algorithm 2). For each matrix in $\mathbb{H}$, the probabilities of all possible syndromes are calculated and stored in vector $\mathbf{p}$.
3) Equivocation rate evaluation (Algorithm 3). For each matrix in $\mathbb{H}$, the equivocation rate is calculated from $\mathbf{p}$ and added to set $\mathbb{E}$.
4) Top t selection (Algorithm 1, line 9). According to the set $\mathbb{E}$ of equivocation rates, the matrices with the top t highest equivocation rates are selected for the extension at the next iteration.

After the iterated extensions are performed, Algorithm 1 returns $\mathbb{H}^{(n-1)}$ as a set of BECs with the top t equivocation rates.

In practice, the message length m is much smaller than the code length n, and parameter t is typically set to less than 10.

TABLE 1. Time complexities of BEC generation methods assuming $m \ll n$ and $t = \mathcal{O}(1)$. The DP method reduces the overall time but incurs a search overhead for data reuse. In contrast, the greedy method implements a heuristic approach with greedy parameter g to realize a similar complexity to that of the DP method.

Procedure	Previous [7]	DP	Greedy
Search	—	$O(n2^m)$	—
Overall	$O(n^2 4^m)$	$O(n4^m)$	$O(n^2 2^m + gn4^m)$

Assuming these practical conditions, we obtain the following theorem.

Theorem 1: Algorithm 1 can be processed in $\mathcal{O}(n^2 4^m)$ time if $m \ll n$ and $t = \mathcal{O}(1)$.

Proof: The loop at line 5 iterates $t2^m$ times because $|\mathbb{H}| = t2^m$: 2^m different columns are added to each of the t matrices. Algorithms 2 and 3 take $\mathcal{O}(l2^m)$ time and $\mathcal{O}(2^m)$ time, respectively. Note that Algorithm 2 can be rewritten as $\mathcal{O}(n2^m)$ time because $l < n$. As a result, the body for the loop at line 2 of Algorithm 1 takes $\mathcal{O}(n2^m t2^m)$ time per iteration. In addition, the loop at line 2 iterates $n - m$ times; thus, the time complexity of Algorithm 1 is given as $\mathcal{O}((n-m)nt4^m)$. This complexity can be rewritten as $O(n^2 4^m)$ if $m \ll n$ and $t = \mathcal{O}(1)$. □

IV. PROPOSED METHODS

Here, we describe the proposed methods in detail. Our contributions over the previous method [7] are summarized as follows. We propose a DP method that performs data reuse to reduce the time complexity of BEC generation from $O(n^2 4^m)$ to $O(n4^m)$ if $m \ll n$ and $t = \mathcal{O}(1)$. We also propose a greedy method that realizes a similar time complexity as the DP method by implementing a heuristic approach that can be parallelized efficiently on GPU. Table 1 summarizes the contributions of our methods.

A. DYNAMIC PROGRAMMING METHOD

The previous method can be improved by eliminating the duplicate computations in Algorithm 2. In other words, Algorithm 2 calculates the entire distribution, $\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \ldots, \mathbf{p}^{(l)}$, each time it is called from Algorithm 1; however, all distributions except the last $\mathbf{p}^{(l)}$ have already been computed at the previous call of Algorithm 2. Such duplicate computations can be eliminated using a DP method that exploits the advantage of the recurrence relation in Eq. (12).

Algorithm 4 describes the proposed DP method that reuses the computed distributions. To realize this efficient reuse, the proposed DP method extends the previous method as follows.

- Storing computed distributions (line 14). Algorithm 4 holds a set $\hat{\mathbb{P}}$ of computed distributions that can be reused in the next iteration.
- Searching computed distributions (line 8). Algorithm 4 finds the appropriate distribution $\hat{\mathbf{p}}$ to be reused at line 9.

The search procedure is described in Algorithm 5, which finds the computed distribution from $\hat{\mathbb{P}}$ that can be reused for H. To facilitate this discussion, we introduce notation $H_{\le i}$,

Algorithm 4 BECgenerationDP (m, n, α, t)

Input: message length m, code length n, crossover probability α, and the number t of matrices to save.

Output: A set $\mathbb{H}^{(n-1)}$ of top t BECs of length n.

```
1:  $\mathbb{H}^{(m-1)} \leftarrow \{I_m\}$;
2:  $\hat{\mathbb{P}} \leftarrow \{$ DistrCalc$(m, \alpha, I_m, m-1)$ $\}$;
3:  for $i \leftarrow m$ to $n-1$ do
4:      $\mathbb{E} \leftarrow \emptyset$;
5:      $\mathbb{H} \leftarrow \{[H^{(i-1)} \mid Q], H^{(i-1)} \in \mathbb{H}^{(i-1)}, Q \in \{0,1\}^{1\times m}\}$;   ▷ Eq. (17)
6:      $\mathbb{P} \leftarrow \emptyset$;
7:      for each $H \in \mathbb{H}$ do
8:          $\hat{\mathbf{p}} \leftarrow$ SearchDistr$(H, \mathbb{H}^{(i-1)}, \hat{\mathbb{P}})$;   ▷ Alg. 5
9:          $\mathbf{p} \leftarrow$ DistrCalcDP$(m, \alpha, H, i, \hat{\mathbf{p}})$;   ▷ Alg. 6
10:         $\mathbb{E} \leftarrow \mathbb{E} \cup \{$ EquivRateEval$(m, \mathbf{p})$ $\}$;
11:         $\mathbb{P} \leftarrow \mathbb{P} \cup \{\mathbf{p}\}$;
12:     end for
13:     $\mathbb{H}^{(i)} \leftarrow$ top $t$ matrices in $\mathbb{H}$;   ▷ According to $\mathbb{E}$
14:     $\hat{\mathbb{P}} \leftarrow \mathbb{P}$;
15: end for
```

which represents the matrix comprising the first i-th columns of matrix H. Given matrix H, Algorithm 5 first finds the corresponding matrix $H_{\leq i-1}$ from $\mathbb{H}^{(i-1)}$, where $\mathbb{H}^{(i-1)}$ is the set of the top t matrices of length i. The appropriate distribution $\hat{\mathbf{p}}$ can be returned without an additional search process because $\mathbb{H}^{(i-1)}$ and $\hat{\mathbb{P}}$ have the same indexing scheme, in which matrix $H_{\leq i-1}$ and its corresponding distribution $\hat{\mathbf{p}}$ can be accessed with the same offset in $\mathbb{H}^{(i-1)}$ and $\hat{\mathbb{P}}$, respectively.

Note that the vector form given in Eq. (4) is useful in terms of realizing an efficient comparison of different matrices at line 2 of Algorithm 5. With this form, the matrix comparison can be realized by comparing the $i-1$ integer values rather than that of the $n(i-1)$ boolean values.

Algorithm 6 shows the distribution calculation extended for the proposed DP method. Here, rather than calculating the entire distribution, Algorithm 6 reuses the distribution at the previous step, which allows us to replace the double nested loop in Algorithm 2 with a single loop.

The time complexity of the DP method can be given by the following theorem:

Theorem 2: Algorithm 4 can be processed in $\mathcal{O}(n4^m)$ time if $m \ll n$ and $t = \mathcal{O}(1)$.

Proof: The loop at line 7 in Algorithm 4 iterates $t2^m$ times because $|\mathbb{H}| = t2^m$; 2^m different columns are added to each of t matrices. In addition, the body for the loop at line 3 takes $\mathcal{O}(t2^m(t+2^m))$ time per iteration because Algorithms 5 and 6 require $\mathcal{O}(t)$ time and $\mathcal{O}(2^m)$ time, respectively. This loop iterates $n-m$ times; thus, the time complexity of Algorithm 4 is $\mathcal{O}((n-m)t2^m(t+2^m))$, which can be rewritten as $\mathcal{O}(n4^m)$ if $m \ll n$ and $t = \mathcal{O}(1)$. □

Compared with the previous method [7], the proposed DP method reduces the time complexity from $\mathcal{O}(n^2 4^m)$ to $O(n4^m)$ if $m \ll n$ and $t = \mathcal{O}(1)$ (Table 1). However, the DP method

Algorithm 5 SearchDistr $(H, \mathbb{H}^{(i-1)}, \hat{\mathbb{P}})$

Input: Parity check matrix H, set $\mathbb{H}^{(i-1)}$ of BECs of length i, and set $\hat{\mathbb{P}}$ of computed distributions.

Output: Probability distribution $\hat{\mathbf{p}}$ of syndromes, where $\hat{\mathbf{p}} = (p_0, p_1, \ldots, p_{2^m-1})$.

```
1: for each $J \in \mathbb{H}^{(i-1)}$ do
2:     if $J == H_{\leq i-1}$ then
3:         $\hat{\mathbf{p}} \leftarrow$ distribution in $\hat{\mathbb{P}}$ computed for $J$;
4:     end if
5: end for
```

Algorithm 6 DistrCalcDP $(m, \alpha, H, l, \hat{\mathbf{p}})$

Input: message length m, crossover probability α, parity check matrix H, current column index l in H, and reused distribution $\hat{\mathbf{p}}$.

Output: Probability distribution $\mathbf{p}$ of syndromes, where $\mathbf{p} = (p_0, p_1, \ldots, p_{2^m-1})$.

```
1: $\mathbf{p}^{(l-1)} \leftarrow \hat{\mathbf{p}}$;
2: for $j \leftarrow 0$ to $2^m - 1$ do
3:     $p_j^{(l)} \leftarrow (1-\alpha)p_j^{(l-1)} + \alpha p_{j\oplus h_l}^{(l-1)}$;   ▷ Eq. (12)
4: end for
5: $\mathbf{p} \leftarrow \mathbf{p}^{(l)}$;
```

processes Algorithm 5, which incurs a search overhead for data reuse. This overhead incurred by processing Algorithm 5 is given as $\mathcal{O}((n-m)t^2 2^m)$, which can then be given as $\mathcal{O}(n2^m)$ if $m \ll n$ and $t = \mathcal{O}(1)$.

Note that some modifications are required to implement Algorithm 4 on real machines due to memory capacity limitations. Specifically, in line 14, Algorithm 4 assumes that all previous distributions $\hat{\mathbb{P}}$ and new distributions $\mathbb{P}$ are stored in main memory. The space complexity of these distributions is $\mathcal{O}(t4^m)$, which is too large for modern systems. Thus, we trade time complexity for space complexity by (1) avoiding updating the distributions $\mathbb{P}$ (line 11), (2) recalculating the distributions of top t BECs (line 14) using Algorithms 5 and 6, and (3) storing only the distributions of the top t BECs rather than all distributions (line 14). This solution, which reduces the space complexity to $\mathcal{O}(t2^m)$, is reasonable because only the top t BECs are extended for code generation. Note that the recalculation step (4), which processes Algorithms 5 and 6 t times, occurs place outside the loop at line 7; thus, the overhead at line 14 is $\mathcal{O}(t(t+2^m))$ per iteration, which is negligible in terms of the bottleneck part, i.e., $\mathcal{O}(t2^m(t+2^m))$ per iteration.

B. GREEDY METHOD

The proposed greedy method accepts a heuristic approach that facilitates parallelization but may produce different BECs compared to the previous [7] and the proposed DP methods (Section IV-A). The proposed greedy method realizes efficient data reuse under the assumption that higher equivocation rates are generated from specific matrices that

TABLE 2. Top 10 BECs and their equivocation rates generated by the previous method [7] with $m = 5$, $n = 9$, and $t = 10$. Parity check matrices are presented in the vector form.

Parity check matrix	Equivocation rate
(1, 2, 4, 8, 16, 15, 19, 21, 25)	0.4918
(1, 2, 4, 8, 16, 15, 19, 21, 22)	0.4910
(1, 2, 4, 8, 16, 31, 7, 11, 21)	0.4887
(1, 2, 4, 8, 16, 15, 19, 21, 9)	0.4880
(1, 2, 4, 8, 16, 31, 7, 11, 13)	0.4875
(1, 2, 4, 8, 16, 15, 19, 21, 7)	0.4858
(1, 2, 4, 8, 16, 15, 19, 5, 24)	0.4853
(1, 2, 4, 8, 16, 15, 19, 21, 3)	0.4843
(1, 2, 4, 8, 16, 7, 11, 13, 14)	0.4831
(1, 2, 4, 8, 16, 31, 7, 11, 5)	0.4831

have common columns. Based on this assumption, the proposed greedy method discards some of the top t matrices in the i-th step such that matrices $H \in \mathbb{H}^{(i)}$ include the common submatrix $H_{\leq i-g}$, where $g \geq 0$ is a greedy parameter that determines the width of the common submatrix. Note that the greedy method with $g = n$ corresponds to the previous method [7].

One concern about the proposed greedy method is the assumption required to realize data reuse. This assumption was confirmed experimentally, i.e., we confirmed that the top t matrices tend to have common columns. Table 2 shows the top 10 matrices and their equivocation rates generated by Algorithm 1 with $m = 5$, $n = 9$, and $t = 10$. As can be seen in Table 2, six codes have a common prefix $(1, 2, 4, 8, 16, 15, 19)$ of length seven; thus, the last two columns can be ignored by using $g = 2$. In other words, the matrices with the most common prefix are saved for the next iteration, and the remaining four matrices with different prefixes are discarded prior to executing the subsequent iteration.

Algorithm 7 shows the proposed greedy method that reuses computed distributions under the assumption of the common columns in the top t matrices. The greedy method extends the previous method [7] as follows.

- Precomputation of the distribution for the common columns $H_{\leq i-g-1}$ (Algorithm 7, line 5). Note here that "-1" is required because all matrices have been extended by one column at line 4. The proposed method stores the precomputed distribution in vector $\hat{\mathbf{p}}$.
- Reuse of the precomputed distribution (Algorithm 8). The precomputed distribution $\hat{\mathbf{p}}$ is reused to avoid redundant iterations in Algorithm 2. Consequently, the loop at line 2 in Algorithm 8 begins from $l - g$ rather than 1.
- Top t selection (Algorithm 7, line 11). At most t matrices with the most common columns $H_{\leq i-g}$ are saved for the next iteration.

Given extensions mentioned above, the greedy method reduces the time complexity as follows:

Theorem 3: Algorithm 7 can be processed in $\mathcal{O}(n^2 2^m + gn4^m)$ time if $m \ll n$ and $t = \mathcal{O}(1)$.

Proof: Algorithm 7 first calculates the distribution for the common columns $H_{\leq i-g-1}$ at line 5. The time complexity of this calculation is $O((n - m)(n + m - g)2^m)$, which can be derived by $\sum_{i=m}^{n-1}(i - g - 1)2^m$. The time complexity of Algorithm 8 is $\mathcal{O}(g2^m)$ because this algorithm calculates the distributions from $\mathbf{p}^{(l-g)}$ to $\mathbf{p}^{(l)}$. The loop at line 6 in Algorithm 7 iterates at most $t2^m$ time because $|\mathbb{H}| \leq t2^m$. Here, we consider only t matrices; thus, the most common columns at line 11 in Algorithm 7 can be identified in $\mathcal{O}(t)$ time. Therefore, the time complexity of Algorithm 7 is $\mathcal{O}((n-m)((n+m-g)2^m+t2^m g2^m)) = \mathcal{O}((n-m)((n+m)2^m+tg4^m)$. As a result, the time complexity of Algorithm 7 is $\mathcal{O}(n^2 2^m + gn4^m)$ if $m \ll n$ and $t = \mathcal{O}(1)$, and this completes the proof. □

Algorithm 7 BECgenerationG (m, n, α, t, g)

Input: message length m, code length n, crossover probability α, the number t of matrices to save, and greedy parameter g.
Output: A set $\mathbb{H}^{(n-1)}$ of top t BECs of length n.

```
1:  $\mathbb{H}^{(m-1)} \leftarrow \{I_m\}$;
2:  for $i \leftarrow m$ to $n-1$ do
3:      $\mathbb{E} \leftarrow \emptyset$;
4:      $\mathbb{H} \leftarrow \{[H^{(i-1)} \mid Q] \mid H^{(i-1)} \in \mathbb{H}^{(i-1)}, Q \in \{0,1\}^{1\times m}\}$;   ▷ Eq. (17)
5:      $\hat{\mathbf{p}} \leftarrow$ DistrCalc($m, \alpha, H_{\leq i-g-1}, i-g-1$), $H \in \mathbb{H}$;   ▷ Alg. 2
6:      for each $H \in \mathbb{H}$ do
7:          $\mathbf{p} \leftarrow$ DistrCalcG($m, \alpha, H, i, \hat{\mathbf{p}}, g$);   ▷ Alg. 8
8:          $\mathbb{E} \leftarrow \mathbb{E} \cup \{$ EquivRateEval($m, \mathbf{p}$) $\}$;
9:      end for
10:     $\mathbb{H}^{(i)} \leftarrow$ top $t$ matrices in $\mathbb{H}$;   ▷ According to $\mathbb{E}$
11:     Discard matrices in $\mathbb{H}^{(i)}$ that do not have the most common columns $H_{\leq i-g}$;
12: end for
```

Algorithm 8 DistrCalcG $(m, \alpha, H, l, \hat{\mathbf{p}}, g)$

Input: message length m, crossover probability α, parity check matrix H, current column index l in H, reused distribution $\hat{\mathbf{p}}$, and greedy parameter g.
Output: Probability distribution $\mathbf{p}$ of syndromes, where $\mathbf{p} = (p_0, p_1, \ldots, p_{2^m-1})$.

```
1:  $\mathbf{p}^{(l-g)} \leftarrow \hat{\mathbf{p}}$;
2:  for $i \leftarrow l-g$ to $l$ do
3:      for $j \leftarrow 0$ to $2^m - 1$ do
4:          $p_j^{(i)} \leftarrow (1-\alpha)p_j^{(i-1)} + \alpha p_{j\oplus h_i}^{(i-1)}$;   ▷ Eq. (12)
5:      end for
6:  end for
7:  $\mathbf{p} \leftarrow \mathbf{p}^{(l)}$;
```

Compared to the proposed DP method, the proposed greedy method eliminates the overhead incurred by the distribution search (Algorithm 5) but incurs an overhead in the distribution calculation (Table 1). In fact, the time complexity of Algorithm 8 is $\mathcal{O}(g2^m)$, whereas that of Algorithm 6 is $\mathcal{O}(2^m)$. The distribution calculation can be parallelized efficiently using 2^m GPU threads, whereas the distribution search cannot be parallelized efficiently on many threads because search tasks inherently require a reduction operation to unify the search results from the threads.

V. PARALLELIZATION OF PROPOSED METHODS

The proposed DP and greedy methods have a similar loop structure as the previous method [7], whose performance bottleneck occurs at the quadruple-nested loop comprising two double-nested loops at lines 2 and 5 of Algorithm 1 and lines 2 and 3 of Algorithm 2. A similar triple-nested loop also limits the performance of the proposed DP method, as can be found in Algorithm 4 (lines 3 and 7) and Algorithm 6 (line 2). Given this similar loop structure, we focus on the simplest method, i.e., the previous method, to simplify the explanation of how the proposed methods are parallelized.

Among the for loops mentioned above, there are two for loops that can be parallelized efficiently on CPU/GPU cores. These parallelizable loops have the following characteristics: (1) each iteration is independent of all other iterations, and (2) there is a large number of iterations during execution. According to these characteristics, we targeted the following loops for parallelization.

- Loop 1 in Algorithm 1 (lines 5–8). Given different matrices from $\mathbb{H}$, their probability distribution calculation and equivocation rate evaluation can be processed independently on independent cores.
- Loop 2 in Algorithm 2 (lines 3–5). As indicated by Eq. (12), different elements of the vector $\mathbf{p}^{(l)}$ can be processed independently on different cores.

Note that loops 1 and 2 have coarse- and fine-grained parallelisms, respectively. Due to the coarse granularity, loop 1 involves more complicated access patterns than loop 2. Given such characteristics, we decided to assign loops 1 and 2 to CPU and GPU cores, respectively, because GPU cores are better suited to exploit fine-grained parallelism with simple access patterns.

Unfortunately, the outermost loop at line 2 of Algorithm 1 cannot be parallelized due to the data dependence, i.e., code generation for length $i + 1$ depends on that for length i. Similarly, the loop at line 2 of Algorithm 2 must be processed in sequence.

A. COARSE-GRAINED PARALLELIZATION FOR CPU CORES

We developed a main-worker scheme to allow CPU cores to exploit coarse-grained parallelism in loop 1. Using c threads on c physical CPU cores, the scheme processes Algorithm 1 in $\mathcal{O}(n4^m/c)$ time if $m \ll n$ and $t = \mathcal{O}(1)$. The coarse-grained method realizes multithreaded execution that iteratively assigns a task to an idle thread. Here, a task corresponds to the probability distribution calculation or the equivocation rate evaluation for a matrix $H \in \mathbb{H}$. This method is summarized as follows.

- Thread creation. Given c physical CPU cores, the main thread forks c worker threads before iterating the loop at line 2 of Algorithm 1.
- Task management. The main and worker threads share a buffer that holds the current step i and the index for the matrix H to be processed next. Here, the step i is initialized with m, as indicated at line 2 of Algorithm 1. In contrast, the matrix index is initialized with $|\mathbb{H}|$, i.e., the number of matrices in $\mathbb{H}$, and decremented until the matrix index reaches zero, which means that no matrices are left for the current step i. Then, the main thread resets the matrix index, increments the step i, and iterates the same operations while $i < n$.
- Task assignment. The worker threads exclusively access the buffer to find the task to be executed next. Once an idle worker thread finds a task, it decrements the matrix index in the buffer.
- Thread termination. All worker threads are joined to the main thread when the buffer holds the matrix index of 0 for code length n.

In this study, we implemented the abovementioned method using the std::thread class in the 2011 C++ language standard. Besides, we deployed the function `SetThreadAffinityMask` for better cache utilization.

B. FINE-GRAINED PARALLELIZATION FOR GPU CORES

The fine-grained parallel method exploits the data parallelism in loop 2 using 2^m GPU threads. Given 2^m GPU threads, the method processes Algorithm 1 in $\mathcal{O}(n2^m)$ time if $m \ll n$ and $t = \mathcal{O}(1)$. Here, the j-th thread is responsible for calculating $p_j^{(i)}$, where $1 \leq i \leq l$ and $0 \leq j < 2^m$. The CPU is responsible for iterating the i loop at line 2 of Algorithm 2 and calling a kernel function [21] that processes loop 2 in parallel on the GPU. The CPU also manages the data transfer between the CPU and GPU. The initial probabilities $p_0^{(0)}$ and $p_1^{(0)}$ are transferred from the CPU to GPU prior to entering the i loop at line 2. After completing the kernel execution, the calculated probability vector $\mathbf{p}^{(l)}$ is transferred from the GPU to the CPU. During the iterative kernel calls, there is no need to exchange data between the CPU and the GPU if Algorithm 3 is executed on the GPU.

We implemented these methods using the compute unified device architecture (CUDA) [21]. There are two important aspects relative to maximizing the application performance on CUDA-enabled GPUs: (1) minimizing data transfer between the CPU and GPU, and (2) realizing coalesced memory access [21].

In terms of minimizing the data transfer between the CPU and GPU, we decided to parallelize Algorithm 3 on the GPU. This allows us to avoid the iterative transfer of vector $\mathbf{p}$, i.e., the input to Algorithm 3, because Algorithm 3 can access $\mathbf{p}$ directly after completing Algorithm 2 on the GPU. Otherwise, the total amount of data transfer would be $\mathcal{O}(n4^m)$, which cannot be ignored compared to the time complexity of the proposed DP and greedy methods. The computational part of Algorithm 3 performs a reduction operation that merges multiple values into a single value. Here, we parallelized this operation using a well-known tree-based algorithm, in which values are combined along with the paths from leaves to the root in a binary tree. Consequently, Algorithm 3 can be

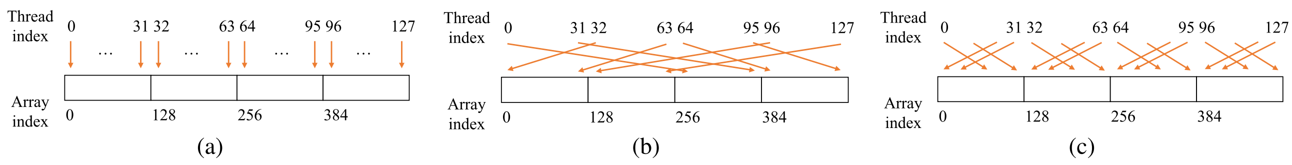


FIGURE 2. Examples of coalesced memory access. (a) single strided access, (b) permutation between warps, and (c) permutation between threads in the same warp. Warps in these examples access a 128-byte aligned segment.

processed in $\mathcal{O}(\log m)$ time with 2^m threads, implying that Algorithm 2 still determines the time complexity of the fine-grained parallel method.

Memory access coalescing is an important technique to maximize the kernel performance on the GPU. Here, we assume that the corresponding data are stored in the float data type, and the data are cached in both L1 and L2 cache memory. As shown in Fig. 2(a), memory transactions from 32 threads in a warp [21] can be coalesced into a 128-byte memory transaction if the threads access a 128-byte aligned data segment on global memory [21]. Based on this assumption, loop 2 of Algorithm 2 can be parallelized with realizing coalesced memory access on the GPU. In the following, we demonstrate this by analyzing each memory access required by line 4 of Algorithm 2.

Lemma 1: Reading $p_j^{(i-1)}$ at line 4 of Algorithm 2 can be performed with a 128-byte memory transaction on the GPU, and writing $p_j^{(i)}$ at line 4 can be performed with a 128-byte memory transaction.

Proof: Consider a task assignment scheme where the j-th thread is responsible for reading $p_j^{(i-1)}$ and writing $p_j^{(i)}$. With this scheme, 32 threads in the same warp access 32 contiguous elements if dimension j is contiguous in the array. These 32 elements exist in a 128-byte aligned segment, as shown in Fig. 2(a); thus, each of the reading and writing operations from a warp can be performed with a 128-byte memory transaction. □

Before analyzing the remaining access to $p_{j\oplus h_i}^{(i-1)}$ at line 4, we introduce the following lemmas.

Lemma 2: Let X be a finite set of non-negative integers, where $X = \{0, 1, \ldots, 2^m - 1\}$ and $m \geq 1$. In addition, let $f : X \to X$ be a function given as $f(x) = x \oplus h$, where $h \in X$ is a constant value. Function f is then a permutation.

Proof: The property of the bitwise XOR operator $\oplus$ gives the proof. Function f is injection because $f^{-1}(x) = f(x)$, for all $x \in X$. This explains that f is bijection because f is defined over a finite set. A bijective function from a set to itself is a permutation. □

Lemma 3: Reading $p_{j\oplus h_i}^{(i-1)}$ at line 4 of Algorithm 2 can be performed with a 128-byte memory transaction on the GPU.

Proof: Consider a task assignment scheme where the j-th thread is responsible for reading $p_{j\oplus h_i}^{(i-1)}$. All threads created from a kernel launch have the same value for variable i because the value is given as an argument to the kernel function; thus, we replace h_i with constant value $h \in X$ to simplify the formulation. Then, the index for $p_{j\oplus h_i}^{(i-1)}$ can be expressed as $f(j) = j \oplus h$. Given such an access pattern, an arbitrary warp accesses a 128-byte aligned segment, which can be classified into the following two cases.

- $h \geq 32$. For all j, both j and $f(j)$ have the same values at the five least-significant bits, which indicates that (1) the warp causes a single strided access and (2) the accessed data exist in a 128-byte aligned segment, as shown in Fig. 2(b).
- $0 \leq h < 32$. Lemma 2 indicates that the warp performs a permutation operation; thus, the data in a 128-byte aligned segment will be swapped between threads in the warp, as shown in Fig. 2(c).

As a result, the reading operation can be processed with a 128-byte memory transaction. □

Finally, we present the following theorem, which implies that loop 2 can be parallelized on the GPU with efficient memory access.

Theorem 4: Loop 2 of Algorithm 2 can be processed with three 128-byte memory transactions.

Proof: The body of loop 2 comprises two memory read, and one memory write operations. According to Lemmas 1 and 3, these operations can be processed with three 128-byte memory transactions. □

VI. EXPERIMENTS

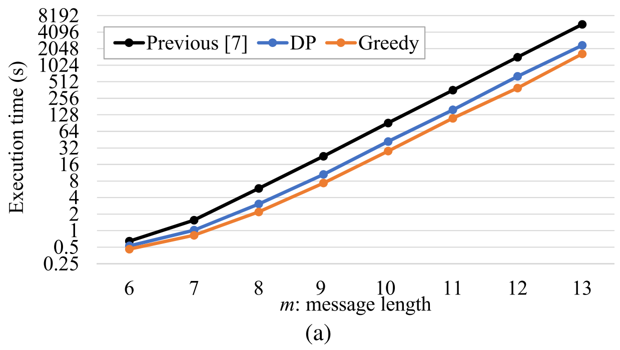
In this section, we present experimental results of the proposed DP and greedy methods compared to the previous method [7] in terms of the execution time and parallel efficiency. We also evaluate the equivocation rate of the generated codes.

We used two different systems to evaluate our coarse-grained and fine-grained parallel methods. Table 3 shows specifications of the experimental systems #1 and #2.

For GPU execution, we determined the number of threads in a block experimentally. Here, we used 64 threads for the thread block size, which demonstrated better performance than other typical configurations, e.g., 128 and 256 threads.

A. SEQUENTIAL PERFORMANCE

Figure 3 shows the sequential execution time of the compared methods with different m and n values on system #1. As shown in Fig. 3(a), where the message length m varied with fixed n, t, and g, the execution time of the three methods increased roughly four times as the message length m was increased by one. Theoretically, these results are reasonable because the measured behaviors agree with



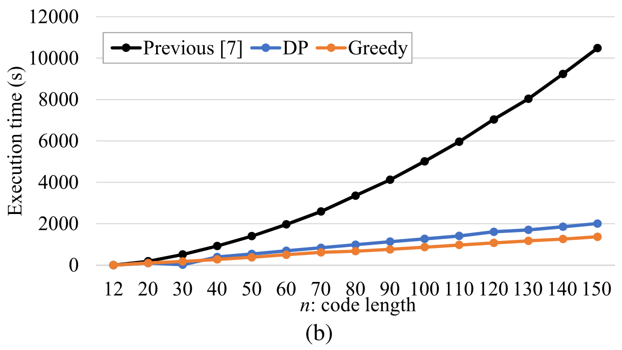


FIGURE 3. Execution time of sequential BEC generation methods on CPU system #1: (a) results for different m with fixed $n = 50$, $t = 20$, and $g = 2$, shown in log scale, (b) results for different n with fixed $m = 12$, $t = 20$, and $g = 2$.

TABLE 3. Specifications of experimental systems.

Item	System #1	System #2
CPU	Intel Core i7-8700K	AMD Ryzen 9 5900X
	6 cores, 3.7 GHz	12 cores, 3.7 GHz
Main memory	DDR4-2666 32GB	DDR4-3200 32GB
GPU	NVIDIA GeForce RTX 2080	N/A
	2944 cores, 1.5 GHz	
OS	Ubuntu 16.04	Windows 10
Compiler	Visual Studio C/C++ and CUDA 10.0 [21]	

TABLE 4. Breakdown analysis of sequential BEC generation methods on CPU system #1. Times in seconds are shown for $m = 12$, $n = 150$, $t = 20$, and $g = 2$.

Item	Previous [7]	DP	Greedy
Distribution calculation	8567	176	393
Equivocation rate evaluation	1820	1819	974
Others	97	15	4
Total	10484	2010	1371

Theorems 1, 2, and 3, which indicate that the three methods have a time complexity of $\mathcal{O}(4^m)$ if n, t, and g are fixed as constant values. Note that the time complexity of the proposed greedy method involves two terms, i.e., $n^2 2^m$ and $gn4^m$; however, the latter term dominates the execution time in Fig. 3(a), where $n^2 \ll 4^m$.

Figure 3(b) shows the results obtained with different code lengths n and fixed values for m, t, and g. The previous method roughly increased the execution time four times when n was doubled. In contrast, the proposed DP and greedy methods were more robust against increasing n. Similar to the results shown in Fig. 3(a), these results are also theoretically reasonable. Note that the time complexity of the proposed greedy method can be considered $\mathcal{O}(n^2)$ if m, g, and t are fixed as constant values. However, the greedy method demonstrated a similar tendency as the DP method, which has a time complexity of $\mathcal{O}(n)$ if m, g, and t are fixed as constant values. The reason for this behavior can be explained by the value of m, which dominates the time complexity of the proposed greedy method. As mentioned previously, the second term $gn4^m$ determines the execution time in Fig. 3(b).

Figure 3 also shows that the proposed DP and greedy methods reduced the execution time compared to the previous method. This advantage comes from the reduced time complexity as summarized in Table 1. As a result, as shown in Fig. 3(b), the proposed DP method increased the speedup over the previous method from 1.8× to 5.2× as n was increased from 20 to 150. In contrast, the speedups shown in Fig. 3(a) were in the range of 2.1× to 2.4× when $m > 8$. Thus, the proposed methods can effectively accelerate BEC generation for large m and n values, especially with larger values of n relative to m.

As shown in Fig. 3, we found that the proposed greedy method was approximately 1.5 times faster than the proposed DP method. The greedy method achieved a faster execution time because it dropped some of the top t matrices, whereas the DP method extended all of the top t matrices in each step. This can be confirmed by Table 4, which shows the breakdown of execution time for $m = 12$, $n = 150$, $t = 20$, and $g = 2$. As shown in this table, the acceleration of the greedy method mainly comes from that of equivocation rate evaluation, which means that set $\mathbb{H}$ in Algorithm 7 has less matrices than that in Algorithm 1. Table 4 also shows that both the DP and greedy methods significantly reduced the time for distribution calculation, which is the bottleneck of the previous method.

B. PARALLEL PERFORMANCE

Table 5 shows the speedup measured on multicore CPU system #1. Here, we varied the number c of worker threads for the previous method, proposed DP method, and proposed greedy method. We found that all three parallel methods achieved linear speedup when $m \geq 8$. In contrast, the methods failed to achieve a linear speedup for small problems where $m < 8$ because the fraction of the parallelizable part in sequential time was not sufficiently high to achieve linear speedup, as described by Amdahl's law [22].

We also evaluated the impact of the algorithmic improvement. The previous method required 16,047 s on $c = 8$ worker threads to generate codes for $m = 15$. In contrast, the proposed DP and greedy methods completed the same task in 17,091 s and 12,633 s on $c = 2$ worker threads,

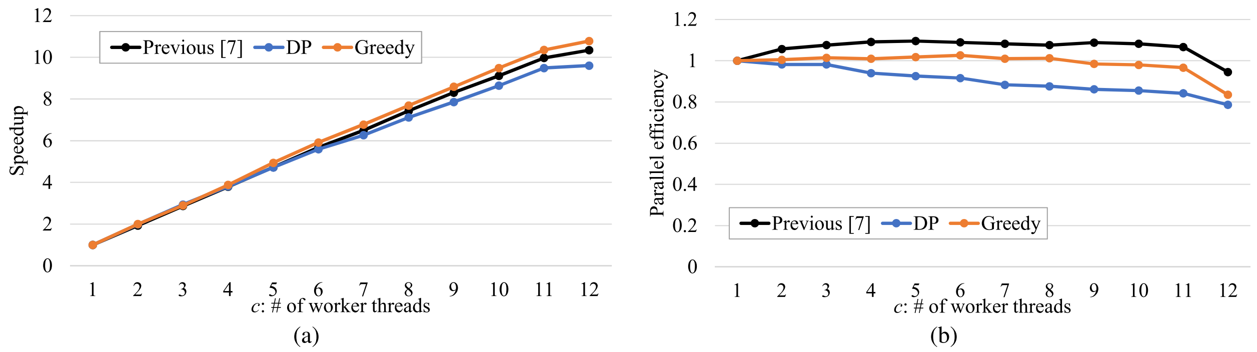


FIGURE 4. Strong and weak scaling analysis of coarse-grained parallel methods on multicore CPU system #2 with different number c of worker threads: (a) speedup for fixed $m = 13$, $n = 50$, $t = 20$ and $g = 2$, (b) parallel efficiency, i.e., the ratio of speedup on c, for fixed amount of work per worker thread. The value for n was appropriately scaled according to the time complexity analysis in Sections III and IV.

TABLE 5. Speedup of parallel BEC generation methods on multicore CPU system #1 with different number c of worker threads. Results are shown for different m values with fixed $n = 50$, $t = 20$, and $g = 2$.

m	Previous [7]			DP			Greedy		
	$c = 2$	$c = 4$	$c = 8$	$c = 2$	$c = 4$	$c = 8$	$c = 2$	$c = 4$	$c = 8$
6	1.2	1.3	1.5	1.0	1.2	1.2	1.0	1.1	1.1
7	1.3	1.9	2.4	1.4	1.6	1.8	1.3	1.4	1.6
8	1.6	2.6	4.1	1.6	2.4	3.3	1.6	2.3	2.9
9	1.7	3.1	4.8	1.8	3.2	4.5	1.8	3.1	4.7
10	1.7	3.2	5.3	1.9	3.5	6.0	1.9	3.3	5.7
11	1.8	2.4	5.4	1.8	3.7	6.4	1.9	3.6	6.2
12	1.7	3.3	5.4	2.1	3.9	6.9	1.9	3.6	6.3
13	1.8	3.4	5.4	1.9	3.6	6.6	1.9	3.6	6.3

respectively. Thus, the proposed methods reduced the number of worker threads to one-quarter of that used by the previous method. Note that this impact increases with the code length n, as described in Section VI-A.

We next evaluated the coarse-grained parallel method on multicore CPU system #2. Figure 4 shows strong and weak scaling results obtained with the previous method, proposed DP method, and proposed greedy method. For the strong scaling in Fig. 4(a), we varied the number c of worker threads with fixed $m = 13$, $n = 50$, $t = 20$, and $g = 2$. For the weak scaling in Fig. 4(b), we varied both n and c so that every thread processed fixed amount of work for different n. In more detail, given c worker threads, we set $n = \sqrt{c(\bar{n}^2 - \bar{m}^2) + \bar{m}^2}$ for the previous method and $n = c(\bar{n} - \bar{m}) + \bar{m}$ for the DP and the greedy methods, where $\bar{m} = 13$ and $\bar{n} = 50$ are values used for the single worker configuration ($c = 1$).

As shown in Fig. 4(a), the previous and the greedy methods had better strong scaling than the DP method. A similar tendency can be found at the weak scaling results in Fig. 4(b); the parallel efficiency of the DP method was around 0.9 whereas the other methods were around 1.0. The DP method had lower efficiency because it accelerated the equivocation rate calculation on worker threads. Therefore, the sequential part, i.e., task assignment on the main thread, becomes relatively larger than the parallel part, degrading the parallel efficiency of the DP method. In fact, the three methods assign tasks in the same manner.

Note that there are $c+1$ threads in total because the master thread forks c worker threads. Consequently, the main thread can disturb one of the worker threads if they run on the same core. Because our system #2 had 12 CPU cores, this resource conflict degraded both strong and weak scaling performance when $c = 12$ (Fig. 4).

We then measured the execution time of the fine-grained parallel method on the GPU system #1. Figure 5(a) shows that our fine-grained parallel method successfully accelerated the BEC generation task on the GPU. For the proposed greedy method, the GPU implementation was 17 times faster than the eight-core CPU implementation. The execution time shown in Fig. 5(a) doubled as we increased m by one, which implies that the time complexity is $\mathcal{O}(2^m)$ if n, t, and g are fixed. In addition, the execution time shown in Fig. 5(b) increased linearly with n. In fact, given fixed t and g values, the fine-grained parallel method deployed 2^m GPU threads to process $\mathcal{O}(n4^m)$ algorithms, which means that each thread was responsible for $\mathcal{O}(n2^m)$ computations.

We also analyzed the kernel performance in terms of achieved occupancy and memory throughput, which can be obtained by NVIDIA Nsight Compute. As for the target kernel, we instrumented the distribution calculation kernel that processes Loop 2 of Algorithm 2. Figure 6 shows the execution time (per kernel launch), achieved occupancy, and memory throughput on GPU system #1. When $m = 18$, the achieved occupancy and memory throughput reached 80% and 60%, respectively, demonstrating efficient execution on the GPU. In contrast, both metric values were less than 7% when $m \leq 11$, where the fine-grained parallel method was slower than the coarse-grained parallel method; the former and latter took 101 s and 66 s to process the previous method on the GPU and eight-core CPU, respectively. Such small problem instances generate at most 2^{11} GPU threads, which are not sufficient to maximize the performance on 2944 cores on the experimental GPU. Consequently, the memory throughput steadily increased with m if GPU threads are enough to hide the memory access latency with GPU computation on the single instruction, multiple thread (SIMT) architecture [21]. It also should be noted that coalesced

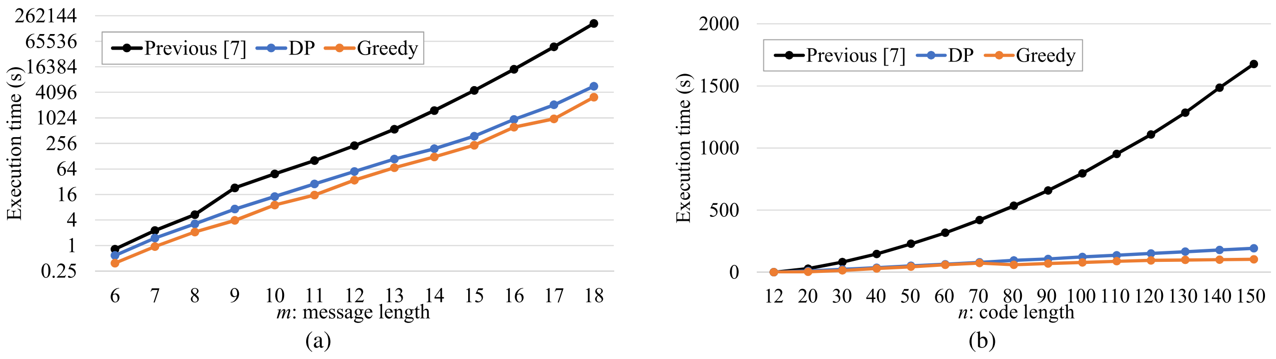


FIGURE 5. Execution time of parallel BEC generation methods on GPU system #1: (a) results for different m values with fixed $n = 50$, $t = 20$, and $g = 2$ (shown in log scale), (b) results for different n with fixed $m = 12$, $t = 20$, and $g = 2$.

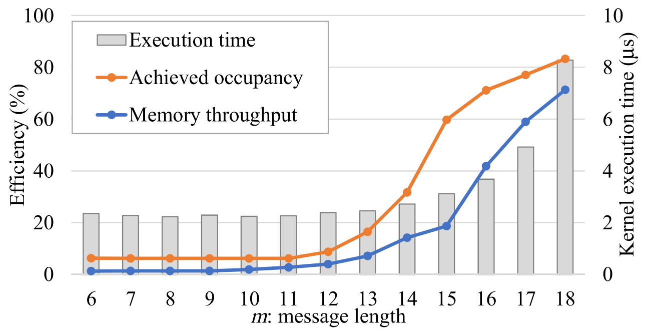


FIGURE 6. Efficiency analysis of distribution calculation kernel for different m values on GPU system #1. Execution time in μs was measured for a single launch of the kernel, which processes loop 2 in Algorithm 1, with $n = m + 1$.

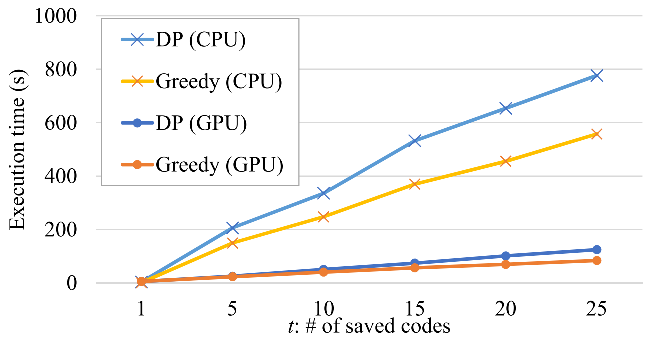


FIGURE 7. Execution time of the proposed DP and greedy methods for different t values with $m = 13$, $n = 50$, and $g = 2$.

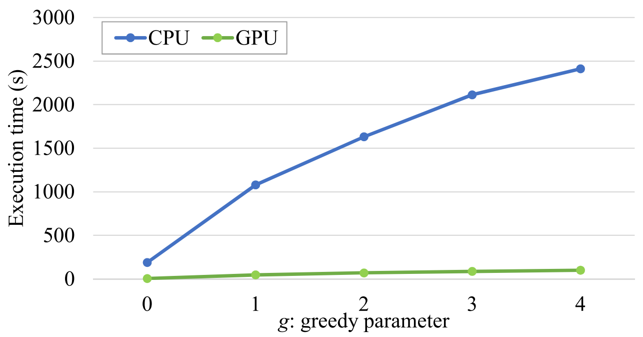


FIGURE 8. Execution time of the proposed greedy method for different g values with $m = 13$, $n = 50$, and $t = 20$.

memory accesses are useful to maximize the memory throughput.

C. IMPACT OF EXECUTION PARAMETERS

We investigated the performance of the proposed methods in terms of execution parameters t and g. Figure 7 shows the execution time of the proposed methods with different execution parameter t on CPU system #1. In this figure, the execution time increased linearly with t. For the proposed DP method, this linear behavior can be explained by Algorithm 4, which extends matrices from the top t matrices. In contrast, the proposed greedy method saves some of the top t matrices. Figure 4 shows that in this experiment, the number of matrices for extension was proportional to t when $m = 13$, $n = 50$, and $g = 2$.

Figure 8 shows the execution time of the proposed greedy method with different values for parameter g on CPU system #1. As shown, the greedy method increased its execution time with g. This behavior is reasonable with the theoretical analysis summarized in Table 1. To be more specific, the execution time was proportional to the number of matrices saved at line 11 of Algorithm 7. Although the greedy parameter g cannot specify this number directly, our timing results indicate that g controls the execution time indirectly.

Another interesting behavior that can be observed in Fig. 8 is that the execution time with $g = 0$ was relatively fast on both the CPU and GPU systems. Although the configuration of $g = 0$ minimizes the execution time for BEC generation, it is equivalent to that of $t = 1$, which strictly limits the search space for matrix H. In other words, the proposed greedy method with $g = 0$ saves only the highest equivocation rate at line 11 of Algorithm 7 because $H^{(i)}_{\le i-g} = H^{(i)}_{\le i}$ when $g = 0$.

D. EQUIVOCATION RATE

Here, we discuss how t and g affect the equivocation rate of BECs. Table 6 shows the equivocation rates of the BECs

TABLE 6. Equivocation rates of the BECs generated by the compared methods. The proposed DP method generated the same code as the previous method [7]. Bold numbers show the highest equivocation rate for specific *m*, *n*, and *t* values.

m	n	t	Previous [7] and DP	Greedy		Random
				$g = 2$	$g = 4$	
10	50	1	**0.931497**	**0.931497**	**0.931497**	0.924632
		3	**0.931809**	0.931699	0.931803	
		5	0.931612	**0.931618**	0.931504	
		10	**0.931872**	0.931701	0.931522	
		15	**0.931872**	0.931823	0.931608	
		20	**0.931872**	0.931706	0.931533	
11	60	1	**0.955632**	**0.955632**	**0.955632**	0.950971
		3	0.955833	**0.955863**	0.955833	
		5	**0.955863**	0.955853	0.955817	
		10	0.955867	**0.955869**	0.955783	
		15	**0.955873**	0.955854	**0.955873**	
		20	**0.955910**	0.955779	0.955873	
12	70	1	**0.971006**	**0.971006**	**0.971006**	0.967819
		3	0.970938	**0.970978**	0.970963	
		5	**0.971042**	0.970995	0.970995	
		10	**0.971051**	0.970954	0.971045	
		15	0.971042	0.971005	**0.971052**	
		20	0.971042	0.971012	**0.971050**	
13	80	1	**0.980716**	**0.980716**	**0.980716**	0.978675
		3	**0.980763**	0.980692	0.980740	
		5	0.980742	**0.980745**	0.980714	
		10	**0.980725**	0.980714	0.980713	
		15	0.980747	0.980714	**0.980752**	
		20	**0.980753**	0.980751	0.980734	

generated by the previous method, the proposed DP method, the proposed greedy method, and a random method. Here, the random method generated matrix $H = [I_m \mid Q]$ 100 times, where $Q \in \{0, 1, \}^{m\times(m-n)}$ is the random vector for given m and n, and calculated the average equivocation rate from the generated matrices. Note that the BECs with the same equivocation rate shown in Table 6 were different codes.

Table 6 shows that higher t sometimes failed to produce a higher equivocation rate. For example, the previous method and the proposed DP method obtained the highest equivocation rate for $m = 12$ and $n = 70$ when $t = 10$. Similar behavior was observed for the greedy method, which obtained the highest rate for $m = 10$, $n = 50$, and $g = 4$ when $t = 3$. Note that the execution time linearly increased with t; thus, we consider that smaller t, e.g., $t = 10$, is an efficient setting for BEC generation methods.

We also suggest using a smaller g value, e.g., $g = 2$, for the proposed greedy algorithm. As shown in Table 6, there is no significant difference between $g = 2$ and $g = 4$ in terms of the equivocation rate. In contrast, the proposed greedy method exhibited a longer execution time with g (Fig. 8). Thus, we consider that a smaller g (≥ 0) is preferred to obtain a high equivocation rates with shorter execution time. Here, that $g = 0$ must be avoided to obtain high equivocation rates because it strictly limits the search space for matrix H, as mentioned previously.

Given specific n, m, and t values, the greedy algorithm tended to yield lower equivocation rates than the previous method and the proposed DP method; however, it produced much higher equivocation rates than the random codes. In addition, the proposed greedy method produced the highest equivocation rate for some cases. Thus, we think that the greedy method is useful for generating competitive codes compared to the previous method.

VII. CONCLUSION

In this paper, we proposed DP and greedy methods for the BEC generation task. Under the assumption that $m \ll n$ and $t = \mathcal{O}(1)$, the proposed DP method reduces the time complexity from $O(n^2 4^m)$ to $O(n4^m)$ while generating the same codes as the previous method. In contrast, the proposed greedy method accepts a heuristic approach that produces different codes but reduces the time complexity to $\mathcal{O}(n^2 2^m + gn4^m)$.

We also parallelized the proposed DP and greedy methods for multicore CPU and GPU systems. These methods exploit coarse-grained parallelism on multicore CPU systems and fine-grained parallelism on GPU systems, which decreases the time complexity by a factor of c, i.e., the number of threads. We theoretically showed that the GPU-based methods can realize efficient coalesced memory accesses with GPU threads.

In our experiments, we also found that the proposed DP and greedy methods successfully reduced execution times, as indicated by a theoretical complexity analysis. The impact of time complexity reduction was equivalent to deploying four times as many cores. Exploiting the coarse-grained parallelism realized a linear speedup on the multicore CPU system, and exploiting fine-grained parallelism realized memory access coalescing on the GPU, yielding 17 times acceleration over the eight-core CPU implementation. In addition, in terms of the quality of the generated codes, we found that the proposed greedy method generated codes that differ from those generated by the previous method; however, the generated codes exhibited a much higher equivocation rate than the random codes. We believe that the proposed parallel methods are useful in terms of accelerating the BEC generation task for messages transmitted with relatively long codes.

In the future, we plan to improve the greedy method using the advantages of coding theory techniques. For example, the search space could be reduced by finding upper bounds on the equivocation rate.

REFERENCES

[1] A. D. Wyner, "The wire-tap channel," *Bell Syst. Tech. J.*, vol. 54, no. 8, pp. 1355–1387, Oct. 1975.

[2] M. Bloch and J. Barros, *Physical-Layer Security: From Information Theory to Security Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[3] H. V. Poor and R. F. Schaefer, "Wireless physical layer security," *Proc. Nat. Acad. Sci. USA*, vol. 114, no. 1, pp. 19–26, 2017.

[4] G. Cohen and G. Zémor, "Generalized coset schemes for the wire-tap channel: Application to biometrics," in *Proc. Int. Symp. Inf. Theory (ISIT)*, 2004, p. 46.

[5] S. Y. El Rouayheb and E. Soljanin, "On wiretap networks II," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2007, pp. 551–555.

[6] Y. Chen and A. J. H. Vinck, "Secrecy coding for the binary symmetric wiretap channel," *Secur. Commun. Netw.*, vol. 4, no. 8, pp. 966–978, Aug. 2011.

[7] K. Zhang, M. Tomlinson, M. Z. Ahmed, M. Ambroze, and M. R. D. Rodrigues, "Best binary equivocation code construction for syndrome coding," *IET Commun.*, vol. 8, no. 10, pp. 1696–1704, Jul. 2014.

[8] L. H. Ozarow and A. D. Wyner, "Wire-tap channel II," *AT&T Bell Lab. Tech. J.*, vol. 63, no. 10, pp. 2135–2157, Dec. 1984.

[9] M. Grassl. (2019). *Bounds on the Minimum Distance of Linear Codes and Quantum Codes*. [Online]. Available: http://www.codetables.de/

[10] M. Grassl, "Searching for linear codes with large minimum distance," in *Discovering Mathematics With Magma*. Heidelberg, Germany: Springer, 2006, pp. 287–313.

[11] S. Al-Hassan, M. Z. Ahmed, and M. Tomlinson, "New best equivocation codes for syndrome coding," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2014, pp. 669–674.

[12] A. Hocquenghem, "Codes correcteurs d'erreurs," Chiffres, Paris, France, (in French), Tech. Rep., 1959, pp. 147–156, vol. 2.

[13] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Inf. Control*, vol. 3, no. 1, pp. 68–79, Mar. 1960.

[14] K. Zhang, M. Tomlinson, and M. Z. Ahmed, "Best random codes construction for syndrome coding scheme," in *Proc. IEEE Int. Conf. Comput. Inf. Technol. (CIT)*, Aug. 2017, pp. 210–214.

[15] J. Pfister, M. A. C. Gomes, J. P. Vilela, and W. K. Harrison, "Quantifying equivocation for finite blocklength wiretap codes," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2017, pp. 1–6.

[16] W. K. Harrison and M. R. Bloch, "Attributes of generators for best finite blocklength coset wiretap codes over erasure channels," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2019, pp. 827–831.

[17] W. K. Harrison, "Exact equivocation expressions for wiretap coding over erasure channel models," *IEEE Commun. Lett.*, vol. 24, no. 12, pp. 2687–2691, Dec. 2020.

[18] S. Ando, F. Ino, T. Fujiwara, and K. Hagihara, "Enumerating joint weight of a binary linear code using parallel architectures: Multi-core CPUs and GPUs," *Int. J. Netw. Comput.*, vol. 5, no. 2, pp. 290–303, 2015.

[19] S. Al-Hassan, M. Z. Ahmed, and M. Tomlinson, "Extension of the parity check matrix to construct the best equivocation codes for syndrome coding," in *Proc. Global Inf. Infrastruct. Netw. Symp. (GIIS)*, Sep. 2014, pp. 1–3.

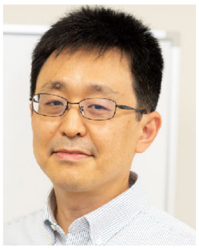
[20] S. Al-Hassan, M. Z. Ahmed, and M. Tomlinson, "Construction of best equivocation codes with highest minimum distance for syndrome coding," in *Proc. IEEE Int. Conf. Commun. Workshop (ICCW)*, Jun. 2015, pp. 496–501.

[21] NVIDIA Corporation. (Sep. 2018). *CUDA C Programming Guide Version 10.0* [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[22] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. AFIPS Conf.*, 1967, pp. 483–485.

YANCHEN LI (Student Member, IEEE) is currently pursuing the Ph.D. degree in computer science with Osaka University. His current research interests include randomized algorithms and high performance computing.

KE ZHANG received the Ph.D. degree in science of communication from the University of Porto, in 2014. She is currently an Associate Professor with the School of Information Engineering, Wuhan University of Technology.



FUMIHIKO INO (Member, IEEE) received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently a Professor with the Graduate School of Information Science and Technology, Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation.