Electronic Research Archive

http://www.aimspress.com/journal/era

ERA, 31(9): 5313–5339.
DOI: 10.3934/era.2023270
Received: 31 May 2023
Revised: 15 July 2023
Accepted: 18 July 2023
Published: 26 July 2023

Research article

Optimization of designing multiple genes encoding the same protein based on NSGA-II for efficient execution on GPUs

Donghyeon Kim and Jinsung Kim*

School of Computer Science and Engineering, Chung-Ang University, Seoul, South Korea

* **Correspondence:** Email: kimjsung@cau.ac.kr; Tel: +8228205554.

Abstract: In synthetic biology, it is a challenge to increase the production of target proteins by maximizing their expression levels. In order to augment expression levels, we need to focus on both homologous recombination and codon adaptation, which are estimated by three objective functions, namely HD (Hamming distance), LRCS (length of repeated or common substring) and CAI (codon adaptation index). Optimizing these objective functions simultaneously becomes a multi-objective optimization problem. The aim is to find satisfying solutions that have high codon adaptation and a low incidence of homologous recombination. However, obtaining satisfactory solutions requires calculating the objective functions multiple times with many cycles and solutions. In this paper, we propose an approach to accelerate the method of designing a set of CDSs (CoDing sequences) based on NSGA-II (non-dominated sorting genetic algorithm II) on NVIDIA GPUs. The implementation accelerated by GPUs improves overall performance by 187.5× using 100 cycles and 128 solutions. Our implementation allows us to use larger solutions and more cycles, leading to outstanding solution quality. The improved implementation provides much better solutions in a similar amount of time compared to other available methods by 1.22× improvements in hypervolume. Furthermore, our approach on GPUs also suggests how to efficiently utilize the latest computational resources in bioinformatics. Finally, we discuss the impacts of the number of cycles and the number of solutions on designing a set of CDSs.

Keywords: protein encoding; multi-objective optimization; bioengineering; GPU computing; NSGA-II

1. Introduction

The production of proteins is a fundamental process for many applications in medicine and biotechnology such as the development of drugs, vaccines and diagnostic tests [1–6]. According to the [7], methods for the efficient production of mammalian proteins for drugs have continuously been developed to support clinical evaluations and this market size exceeds US $12 billion annually. Moreover,

the S-glycoprotein, a key component of antiviral vaccines for severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2), has predominantly been produced in mammalian cells, incurring significant costs. Therefore, efficient protein production could contribute to the development of COVID-19 vaccines and biopharmaceuticals [8]. In addition, improving the production efficiency of enzyme proteins used for production of food and feed has been addressed in [9]. Thus, it is crucial to efficiently produce proteins across various fields. A common method used to boost the production of target proteins is the integration of numerous copies of the target gene into the host genome. However, this method is often expensive and time-consuming. For example, CRIM (conditional replication, integration and modular) [10], which is widely used for DNA integration into the host chromosome, takes around one to two weeks to complete due to multiple recombination steps. To address this, CIGMC (chromosomal integration of gene[s] with multiple copies), which reduces the number of recombination steps to a single step [11] and other approaches [12, 13] have been explored. Although they provide some efficient ways to integrate multiple gene copies to the host genome, the integration of gene copies in close proximity can lead to problems such as homologous recombination [14]. Consequently, the production of proteins may not necessarily be proportional to the number of integrated copies as protein expression levels are affected by both homologous recombination and codon adaptation [15]. The challenge lies in creating a list of gene copies that maximize the production of a target protein. To achieve this, both homologous recombination and codon adaptation should be considered. The integration of gene copies physically close to each other in the host genome can result in homologous recombination, which can lead to the loss of genes. Homologous recombination tends to occur in repetitive sequences in close proximity, leading to a reduction in the copy number of the target gene [16–18]. These findings highlight the importance of minimizing the length of identical sequences to minimize the potential for homologous recombination.

In addition to homologous recombination, the expression levels of proteins are also influenced by codon adaptation. A single CDS (CoDing sequence) that encodes a protein usually consists of several amino acids, each of which can be encoded by multiple synonymous codons. In other words, different codons can encode the same amino acid. Accordingly, for a given target gene there can be several CDSs differing in their synonymous codons. Codon usage bias refers to the tendency for genes to preferentially use certain codons more frequently than others. It reflects how well a codon is adapted to a particular host organism. Codons that are better adapted to the host organism generally result in higher expression levels [19–23]. Therefore, optimizing codon usage in a CDS can effectively enhance protein expression. However, if the same well-optimized codon is used frequently within the same host organism solely to increase the degree of codon adaptation, it can also increase the likelihood of homologous recombination. Therefore, it is crucial to consider homologous recombination and codon adaptation as part of multi-objective optimization approach. This approach aims to generate alternative CDSs that produce the same protein in a host organism using codons with different sequences and are better adapted to the specific host.

Terai et al. [14] proposed a method for designing a set of CDSs that simultaneously takes into account both homologous recombination and codon adaptation. Their approach is based on NSGA-II (non-dominated sorting genetic algorithm II), a widely used genetic algorithm for multi-objective optimization [24]. Three objective functions are defined to estimate the expression level of CDSs and four mutation operators are employed to revise them with the goal of discovering alternative CDSs that exhibit high codon adaptation and a low incidence of homologous recombination at the same

time. In a separate study, Gonzalez-Sanchez *et al.* [25] introduced a method based on MOABC (multi-objective artificial bee colony), a multi-objective optimization algorithm based on artificial bee colony (ABC) swarm intelligence. Although the MOABC-based method shows some improvement compared to Terai et al.'s method, this method will be labelled as NSGA-II-CPU in the rest of the paper, it required more computational resources. The MOABC-based method in [25] requires $2N$ computations of three objective functions per cycle, which are time-consuming components, where N is the number of solutions. On the other hand, NSGA-II-CPU requires N computations of three objective functions per cycle. Therefore, when considering computational costs, the MOABC-based method may not be more efficient than NSGA-II-CPU. Subsequent research led to the development of AP-MOABC (asynchronous parallel-MOABC), a parallelized version of the original method using the OpenMP standard for CPUs [26], resulting in a significant performance boost. However, the execution times of AP-MOABC are still relatively high, and the method was not suitable for the SIMT model on GPUs due to its master-worker parallelization model. In contrast, NSGA-II-CPU is capable of distributing computations evenly across threads on GPUs without relying on the master-worker parallelization model.

In this paper, we propose an approach to accelerate the optimization process in the multi-objective genetic algorithm, specifically focusing on optimizing the NSGA-II-based method, NSGA-II-CPU, proposed by Terai et al. Our approach aims to efficiently perform the three multi-objective functions and four mutation operations on NVIDIA GPUs, significantly reducing execution time required for the multi-objective optimization problem of protein encoding and enabling the handling of larger datasets and increased computations. To achieve this, we optimized the three multi-objective functions using a divide-and-conquer and dynamic programming approach, efficiently utilizing features of GPU architecture. Furthermore, we investigated the impact of the number of solutions and cycles on the expression levels of designing CDSs. The experimental results demonstrated a marked improvement in performance compared to existing alternatives.

On an NVIDIA GeForce RTX 4090, we achieved an average speedup of approximately 187.5× using 100 cycles and 128 solutions. While our implementation slightly lagged behind AP-MOABC in terms of solution quality measured by hypervolume and minimum distance, it overcame the execution time restriction by efficiently executing the method of designing a set of CDSs on GPUs. Consequently, within a similar timeframe to the execution time of AP-MOABC, we were able to design CDSs with larger datasets and more cycles than 128 solutions and 100 cycles, leading to outstanding solution quality. The implementation achieved up to a 1.22× improvement in hypervolume and a 1.20× improvements in minimum distance compared to AP-MOABC.

This paper is organized as follows. Section 2 provides background information. Section 3 details our approach to optimize Terai *et al.*'s method based on NSGA-II for designing multiple genes on GPUs. Section 4 presents the experimental results. Section 5 discusses related research on protein encoding. Finally, Section 6 concludes the paper and Section 7 outlines potential future work.

2. Background and related works

2.1. Multi-objective optimization

A multi-objective optimization problem involves optimizing multiple objective functions simultaneously. In contrast to single-objective optimization, which focuses on a single objective function,

multi-objective optimization requires the optimization of two or more objective functions that may compete or conflict with each other. The goal of multi-objective optimization is to find a set of solutions that represent the best trade-off between the objectives. In single-objective optimization, evaluating the superiority of a solution is straightforward because it involves comparing the values of a single objective function. However, in multi-objective optimization, the quality of a solution is assessed by considering all the values of the objective functions simultaneously. Specifically, a solution dominates another solution if it performs better in at least one objective and is not worse in others. For example, considering the two solution vectors s_1 and s_2 of length m, with each element representing a value of an objective function. Assuming that all objective functions are to be minimized and s_1 is a better solution than s_2— $s_1 \prec s_2$ *if and only if* $\forall i \in \{1, 2, \ldots, m\} : f_i(s_1) \leq f_i(s_2) \bigwedge \exists j \in \{1, 2, \ldots, m\} : f_j(s_1) < f_j(s_2)$; $\prec$ indicates the Pareto dominance relationship between two solutions [24, 27]. In other words, s_1 dominates s_2 or s_1 is not dominated by s_2. A set of solutions that represent the best trade-off solutions between the objectives is called the Pareto front or Pareto optimal solutions. Although it is possible to find all true Pareto optimal solutions by processing all cases, it requires significant amount of computational resources and time. For example, if a solution consists of 500 amino acids and each amino acid can be encoded by 3 synonymous codons, 3^{500} cases must be compared to find all true Pareto optimal solutions. In practice, many real-world problems do not have a known true Pareto front. Instead, multi-objective optimization aims to obtain multiple solutions that are close to the true Pareto front within a reasonable amount of time. In addition, it is crucial to consider the divergence between solutions to ensure a good spread of solutions in the objective space. In summary, the goal of multi-objective optimization is to obtain solutions that are distributed and close to the true Pareto optimal front within a reasonable computational timeframe.

2.2. NSGA-II

The NSGA-II is a multi-objective optimization algorithm based on genetic algorithms. It consists of two steps, the initialization step and the generation step. The generation step, which includes the crossover step, mutation step and selection step, can be repeated multiple times. Each generation step is called a cycle. In the initialization step, a set of N solutions are generated and these N solutions are used as the original solutions in the first generation. Then, in the crossover step of the generation step, pairs of solutions randomly selected from the original solutions with the crossover probability. These pairs exchanges the same parts of their solutions, introducing variations. In the mutation step, each solution is mutated from its original form to generate a new solution, resulting in a total of $2N$ solutions (N original and N generated). Next, the NSGA-II calculates the multi-objective functions of the N solutions. In order to select the better N solutions from the $2N$ solutions, the algorithm performs non-dominated sorting with the calculated multi-objective functions. In non-dominated sorting, the rank value of each solution is determined through the dominant tests and the solutions are sorted in ascending order based on their rank values. For example, solutions with a rank value of 0 dominate solutions with rank values greater than 0. However, among solutions with equal rank values, it is not possible to determine which solution dominates others. To address this, the algorithm performs crowding distance sorting, which measures the divergence of solutions in the same rank in the objective space [24] and sorts solutions in descending order based on their crowding distance. Solutions with lower ranks are preferred and among solutions whose rank values are equal, those with higher crowding distances are preferred. Finally, the selected N solutions are used as the original solutions for the next generation

step or cycle. However, according to NSGA-II-CPU, the crossover step is deemed ineffective because the multi-objective functions for the given problems do not have rugged local optima and the mutation step is solely used to generate new solutions in proximity to the true Pareto front. Therefore, we also excluded the crossover step in our implementation.

2.3. Three objective functions

In the multi-objective optimization problem for designing a set of CDSs based on NSGA-II, three objective functions are employed: *mCAI* (*minimum value of codon adaptation index*), *mHD* (*minimum value of Hamming distance*) and *MLRCS* (*maximum length of repeated or common substring*). These objectives aim to identify solutions that simultaneously exhibit a low incidence of homologous recombination and high codon adaptation. In other words, the goal is to maximize protein expression levels by utilizing solutions composed of codons that are well-adapted to the host organism while minimizing similarity among CDSs.

The objective function *mCAI* evaluates the level of codon adaption of a solution in relation to the host organism, while the remaining two objective functions, *mHD* and *MLRCS*, assess the similarity among CDSs within a solution. However, a conflict typically arises between the objective function *mCAI* and the other two objective functions, *mHD* and *MLRCS*, due to the presence of numerous identical codons within CDSs, leading to a high degree of similarity among them when *mCAI* has a high value. The value of *mCAI* is determined by the weight of the codons. However, for a given amino acid, there exists only one codon with the highest weight among multiple synonymous codons. Consequently, if the value of *mCAI* improves, the values of *mHD* and *MLRCS* will worsen.

2.3.1. Minimum value of codon adaptation index (mCAI)

Depending on the specific host organism, certain codons among the synonymous codons are more frequently used. Such codons demonstrate a higher degree of adaptation to the host organism. Consequently, when designing CDSs, if the amino acids are encoded using the more frequently used codons, the resulting CDSs will have a high protein expression level. For each CDS in a solution, the value of the CAI (codon adaptation index) indicates the extent to which highly adapted codons are encoded in the CDS.

$$CAI(CDS_i) = \sqrt[K]{\prod_{k=1}^{K} weight(codon_{i,k})} \tag{1}$$

In Eq (1), CDS_i indicates the i-th CDS in the solution; K is the total number of codons in a single CDS; and $codon_{i,k}$ indicates k-th codon of i-th CDS in the solution. The weight assigned to each codon is calculated based on its frequency of use within the synonymous codons in the host organism. It is calculated by dividing its frequency of use by the frequency of the most frequently used codon among its synonymous codons. In this paper, the weight assigned to each codon is consistent with the values used in previous studies [14,25,28] and these weight values were calculated by analyzing the top 1000 highly expressed genes in the host organism *S.cerevisiae* [29].

$$mCAI = \min_{1 \le i \le S} CAI(CDS_i) \tag{2}$$

In Eq (2), S is the total number of CDSs in a solution. The minimum value of the CAIs of the CDSs in a solution is used as the value of the mCAI objective function. This approach is employed because using the average CAI value of CDSs as the objective function may not effectively capture the presence of CDSs with CAIs much lower than the average. It is preferred to have a large value for mCAI in the solution to maximize protein expression levels.

2.3.2. Minimum value of Hamming distance (mHD)

$$HD(CDS_i, CDS_j) = \sum_{1 \le l \le L} \sigma(CDS_{i,l}, CDS_{j,l}) \tag{3}$$

The HD values are calculated for all possible pairs of CDSs within a solution and each HD value indicates the number of different bases between the paired of CDSs. In Eq (3), CDS_i and CDS_j indicate the i-th CDS and j-th CDS in the solution, respectively, and L is the length of a CDS. If the l-th base of the i-th CDS and the j-th CDS are different, the value of $\sigma(CDS_{i,l}, CDS_{j,l})$ is 1. Conversely, if they are the same, the value is 0.

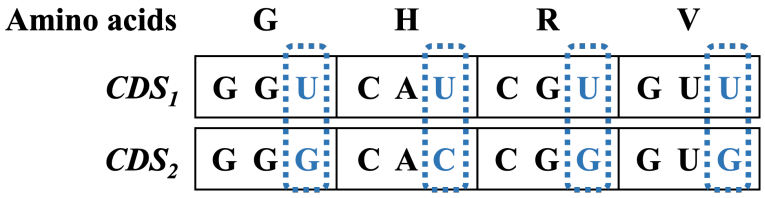


Figure 1. Example of the Hamming distance between CDS_1 and CDS_2 (G: Glycine, H: Histidine, R: Arginine and V: Valine).

For example, let us assume the first CDS in the solution is "GGUCAUCGUGUU" and the second CDS is "GGGCACCGGGUG" as shown in Figure 1. The codons in the CDSs encode the corresponding amino acids. The first codon of CDS_1, GGU and the first codon of CDS_2, GGG, are synonymous codons of glycine. The HD value between the first CDS and the second CDS is 4 because the 3rd, 6th, 9th and 12th bases differ between the pair. The HD value increases as the number of differing bases at the same positions within the CDS pair increases.

$$mHD = \min_{1 \le i < j \le S} \frac{HD(CDS_i, CDS_j)}{L} \tag{4}$$

In Eq (4), S is the total number of CDSs in a solution. Similar to the rationale behind the $mCAI$ objective function, the smallest HD value among the HD values within the solution is used to calculate the mHD objective function. The mHD value is calculated by dividing the minimum HD value by the length of the CDS. It is preferred to have a larger mHD value in the solution to maximize the protein expression levels.

2.3.3. Maximum length of repeated or common substring (MLRCS)

In a solution, the longest common substring between all pairs of CDSs or within the same CDS is determined to calculate the $MLRCS$ objective function. The value of $MLRCS$ objective function is obtained by dividing the length of the longest common substring among the CDSs by the length

of a single CDS. $S_{i,p,l}$ indicates a substring of length l starting from the p-th base of the i-th CDS in a solution. Similarly, $S_{j,q,l}$ is a substring of length l starting from the q-th base of the j-th CDS in a solution.

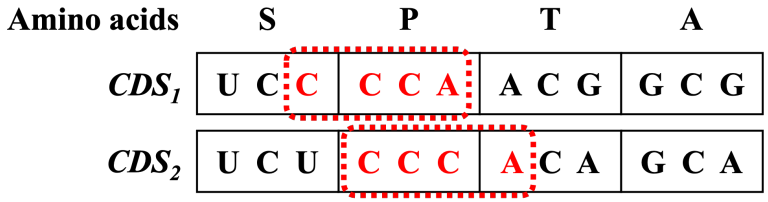


Figure 2. Example of the longest substring between CDS_1 and CDS_2 (S: Serine, P: Proline, T: Threonine and A: Alanine).

For example, let us consider two CDSs shown in Figure 2: CDS_1 is "UCCCCAACGGCG" and CDS_2 is "UCUCCCACAGCA". The longest common substring between them is CCCA (highlighted in Figure 2). In this case, $S_{1,3,4}$ and $S_{2,4,4}$ indicate the longest common substring CCCA for CDS_1 and CDS_2, respectively. Therefore, the length of the longest substring between CDS_1 and CDS_2 is 4. Note that the starting point of each substring must be different ($p \neq q$ if $i = j$) when calculating the longest common substring within a single CDS. It is preferred to have a smaller value for the *MLRCS* objective function in the solution to maximize protein expression levels.

2.4. Four mutation operators

Amino acids are typically encoded by multiple synonymous codons, with the exception of two amino acids, *methionine* and *tryptophan*, which have only one synonymous codon. For example, the amino acid *leucine* can be encoded by six different codons: UUA, UUG, CUU, CUC, CUA and CUG. In the context of CDSs, the mutation process involves replacing the existing codons with their synonymous counterparts. The mutation operators determine which codons in a solution will be mutated and employ specific methods to mutate the codons.

Once the codons to be mutated are determined by the mutation operator, a random number in the range $(0, 1]$ is generated for each selected codon. If the generated random number is smaller than the given mutation probability, the corresponding codon is mutated by the selected mutation operator. For example, assuming that 100 codons in a solution are selected by the mutation operator and the mutation probability is 5%, approximately five codons will be mutated. In our method, the mutation step involves randomly selecting one of four mutation operators for each solution. Three of these operators are designed to improve the values of the objective function, while the remaining operator randomly mutates some codons in the solution. The four mutation operators are described as follows:

1) The codons of a single CDS with the *mCAI* value in the solution can be replaced with their synonymous codons with a higher weight. However, if the codon already has the highest weight, it is not be replaced.
2) The codons of a pair of CDSs with the *mHD* value in the solution can be randomly replaced with their synonymous codons.
3) The codons belonging to the two substrings with the *MLRCS* value in the solution can be randomly replaced with their synonymous codons.

4) The codons of all CDSs in the solution can be randomly replaced with their synonymous codons.

In the mutation operators 2, 3 and 4, when codons are randomly replaced with synonymous codons, they will not be replaced with themselves to ensure variation within the solution.

3. Efficient parallel implementation on GPUs

3.1. Overview of our implementation

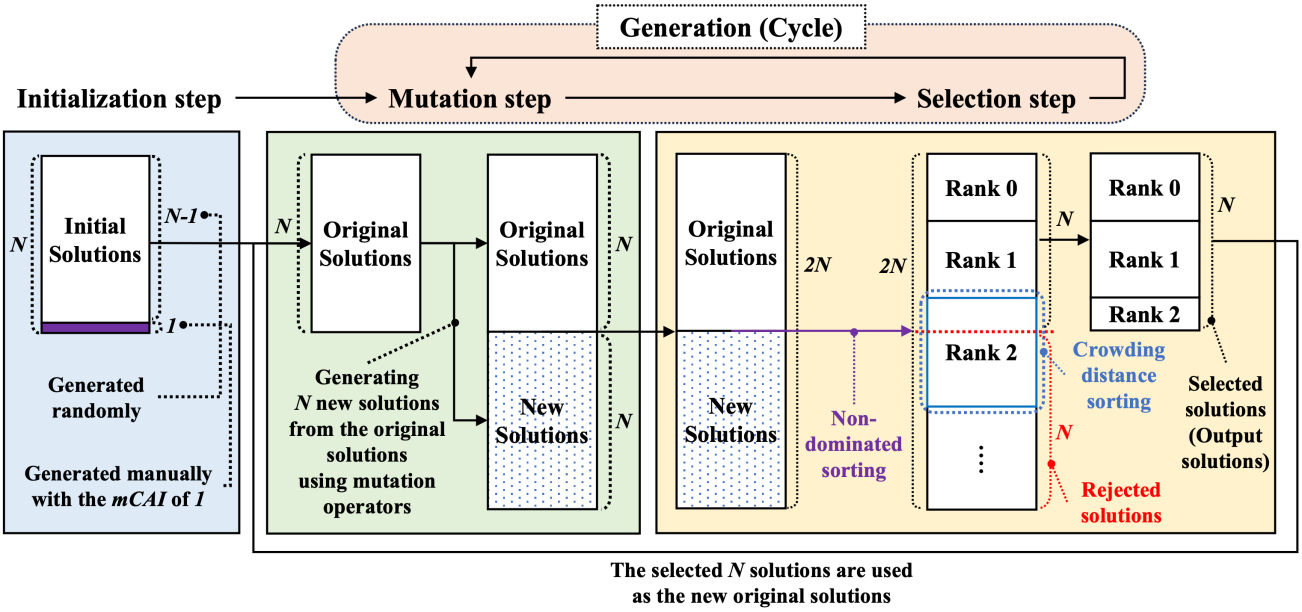


Figure 3. Procedure of our implementation based on NSGA-II-CPU.

Figure 3 illustrates the procedure of our approach, which aims to optimize the NSGA-II-based method, NSGA-II-CPU, for designing a set of CDSs in order to increase the production of target proteins. This approach consists of three steps: initialization step, mutation step and selection step. Both the mutation step and the selection step are designed to generate numerous solutions with high protein expression levels.

The initialization step, shown at the leftmost part of Figure 3, is the first step where N solutions are generated. To generate N solutions, $N - 1$ solutions are generated by randomly selecting codons, while the remaining one solution is generated by manually selecting codons with the highest weight. This is possible because the weight of codons in the host organism is experimentally provided. The solution composed of codons with the highest weight is used to generate solutions with high values of the *mCAI* objective function by undergoing mutation in the mutation step. In the mutation step shown in the middle part of Figure 3, N solutions are newly generated by mutating the N original solutions, resulting in a total of $2N$ solutions. In the first generation, the N original solutions are generated in the initialization step. After the first generation, the N original solutions are generated in the previous generation. When a new solution is generated from the original solution, one of the four mutation operators mentioned in Section 2.4 is randomly used. Finally, in the selection step shown at the rightmost part of Figure 3, non-dominated sorting and crowding distance sorting are performed in

order to select N good solutions out of the $2N$ solutions. These selected N solutions serve as the new original solutions for the subsequent mutation step. This process is repeated for multiple generations and the resulting $2N$ solutions are stored in a file.

In our implementation, each step is implemented as a separate CUDA kernel. Additionally, there is another CUDA kernel responsible for initializing the state of the random number generators to ensure that each thread can generate different random numbers. The CUDA kernels for the random number generators and the initialization step are only executed once at the beginning of the process, while the CUDA kernels of the mutation step and the selection step are executed multiple times. Furthermore, our implementation allows users to provide three input parameters: N, G and P_m. N represents the number of solutions in a population, G represents the number of generations (cycles) and P_m represents the mutation probability. These input parameters are used to generate the population with G cycles and $2N$ solutions based on the mutation probability P_m.

3.2. How to map the population to thread blocks

According to the approach used by NSGA-II-CPU, there are no interactions between solutions but there are significant interactions between CDSs. When designing a set of CDSs, the calculation of the three objective functions is too time-consuming and requires numerous computations between codons and between CDSs within a solution. To address this, our implementation utilizes shared memory and maps a solution to a thread block, except for the selection step in Figure 3. By mapping a solution to a thread block on shared memory, all threads within a thread block have access to the CDSs within that solution.

To handle a population composed of N solutions, N thread blocks are employed for both the initialization step and the mutation step. Initially, each thread block generates a solution during the initialization step, resulting in the creation of N solutions called *original solutions*. In the mutation kernel, each thread block contains two solutions: (1) an original solution generated during the initialization step and (2) a new solution created by mutating the original solution using one of the four mutation operators in shared memory. This design is based on the fact that all threads within a thread block have access to a solution in shared memory, which offers significantly faster performance (over 10×) compared to global memory [30].

Unlike the initialization and mutation steps, the selection step involves using $2N$ threads to sort $2N$ solutions and the number of thread blocks depends on the number of threads within each block. If a thread block consists of T threads then $\lceil 2N/T \rceil$ thread blocks are required. For example, assuming that we have 256 threads and 2048 solutions, we would need 8 thread blocks because of $\lceil 2048/256 \rceil = 8$. Given that all solutions in a population need to be accessed for comparison in the selection step, they should be stored in global memory, which is accessible by all threads. Furthermore, all information about the codons of all amino acids is stored in constant memory, which is designed for read-only data that is accessed repeatedly within a kernel.

3.3. How to mutate codons on GPUs

In order to generate a new solution from the original solution, we mutate the codons within the original solution using one of the four mutation operators described in Section 2.4. After determining which codons to mutate, these codons are evenly distributed among the threads in a thread block as

much as possible. Then, each thread generates random numbers according to the number of codons assigned to it. If the generated random number for a codon is smaller than the mutation probability, this codon is mutated.

In our implementation, when a thread generates random numbers, it utilizes the random number generator based on the XORWOW algorithm in the cuRAND library. We use *curand_uniform()* function, which returns a sequence of pseudo-random floats uniformly distributed in range $(0.0, 1.0]$. Furthermore, as discussed in Section 2.4, depending on the mutation operators, codons can be mutated randomly with any synonymous codons or they can be mutated randomly by selecting one of the synonymous codons with a higher weight than the original codon. For the latter case, the generated random number is multiplied by the number of synonymous codons with a higher weight compared to the codon to be mutated. The calculated value is then rounded down and this number is used to select one of the synonymous codons with a higher weight. However, for the former case, instead of considering only the synonymous codons with a higher weight, the generated random number is multiplied by the total number of synonymous codons. The calculated value may indicate the original codon. In this case, the above process is repeated until a different codon is selected.

3.4. How to calculate three objective functions on GPUs

To evaluate the quality of solutions, the values of the three objective functions are used. Therefore, the values of the objective functions should be calculated for the newly generated solutions. These values are computed during the initialization step and the mutation step. The initialization step calculates the objective function values for the initialized solutions, while the mutation step calculates the objective function values for the mutated solutions (newly generated solutions). The calculation of the objective functions is performed for only N solutions per cycle.

Since each thread block is responsible for generating one solution in both steps, the threads within a block are used to calculate the objective function values of the solution. For the three objective functions, mHD, $mCAI$ and $MLRCS$, the values CAI, HD and $LRCS$ for CDSs should be calculated. Each thread within a thread block performs its own parts of the computations for these values and store the intermediate results in shared memory, which can be accessed by all threads within the block. Then, we exploit the divide-and-conquer strategy through shared memory to calculate the values of CAI, HD and $LRCS$. Finally, the value of the corresponding objective function may be updated by comparing the existing value with the calculated value.

For example, assuming that a solution is composed of two CDSs, the CAI_1 value of the first CDS is calculated first and the $mCAI$ objective function value is updated with CAI_1. Then, the CAI_2 value for the second CDS is calculated. If the CAI_2 is smaller than the existing $mCAI$ objective function value, $mCAI$ is replaced with CAI_2. Otherwise, $mCAI$ remains unchanged. The objective function values mHD and $MLRCS$ are also calculated similarly.

3.4.1. Calculation of HD

Values of HD are calculated for all pairs of CDSs in a solution. Each HD value corresponds to a pair of two CDSs in a solution. Let us assume that the length of a CDS is L and the number of threads in a thread block is T. As shown in Figure 4, each thread handles L/T pairs of bases within a pair of CDSs in order to calculate the HD value for that pair. It is important to note that, when comparing two

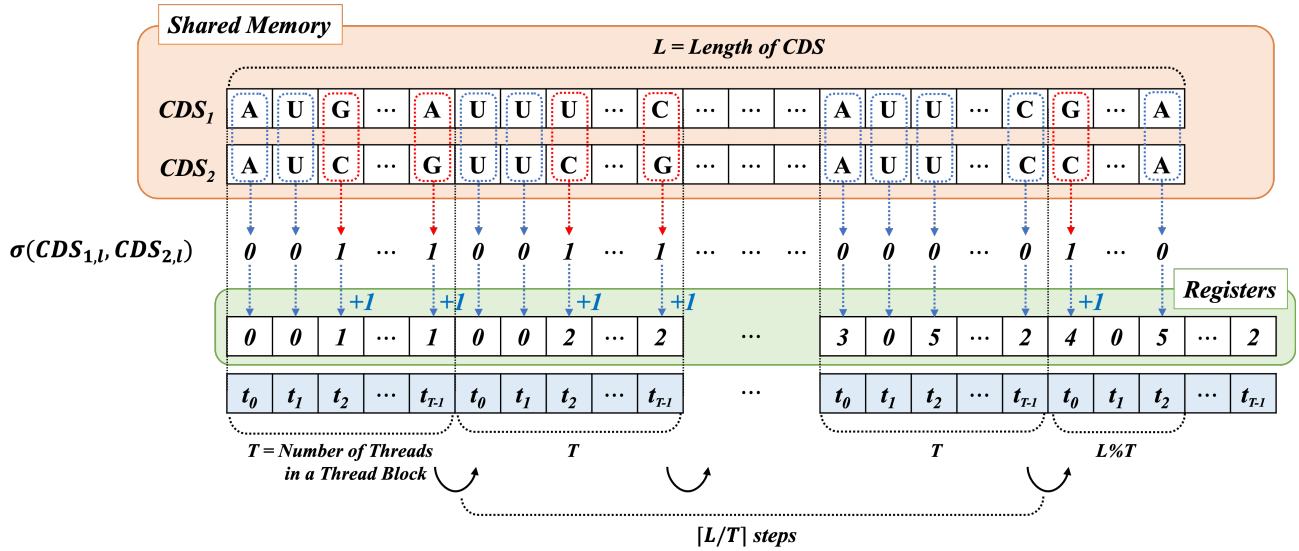


Figure 4. Example of our approach to calculate the intermediate HD values of a pairs of CDSs.

bases, they must be positioned at the same location in the CDSs.

The calculation of the HD value follows Eq (3), which is straightforward. For each pair of two bases, if they are different, the HD value is increased by 1. Otherwise, the HD value remains unchanged. In this way, the HD value indicates the number of bases that are located at the same position but differ between two CDSs. Each thread calculates the number of differing bases among L/T pairs of bases and stores this value in a register which is only visible to that particular thread. After calculating T intermediate values of HD, all threads within a thread block store their intermediate results in shared memory. Then, the divide-and-conquer strategy is employed to calculate the final value of HD as shown in Figure 5. The values at the top of Figure 5 represent the T intermediate values for the final HD value. In the first step, the half of threads, from t_0 to $t_{T/2-1}$, in a thread block accumulates their corresponding value which the other half from $t_{T/2}$ to t_{T-1}, has in shared memory. As the steps progress, the number of threads used for addition is halved and in the final step, denoted as $\lceil log_2 T \rceil$ step, a single thread, t_0, calculates the final HD value by adding the two intermediate values. Finally, the sum of the HD values is divided by L for normalization. If the calculated value is less than the existing value of mHD, the mHD value is replaced with the calculated value. Otherwise, the mHD value remains unchanged.

3.4.2. Calculation of CAI

The value of $mCAI$ represents the minimum CAI value among the CDSs in the solution. To calculate the CAI value for each CDS in the solution, we assume that each CDS consists of K codons and each thread block contains T threads. Each thread is assigned K/T codons and calculates the CAI value for one CDS. As shown in Eq (1), each thread calculates the K-th root of the weight of each codon for the assigned K/T codons. Then, it multiplies all these values together on a register, resulting in an intermediate value of CAI. All threads within a thread block store their intermediate values in shared memory. The final value of CAI is also computed using the divide-and-conquer strategy.

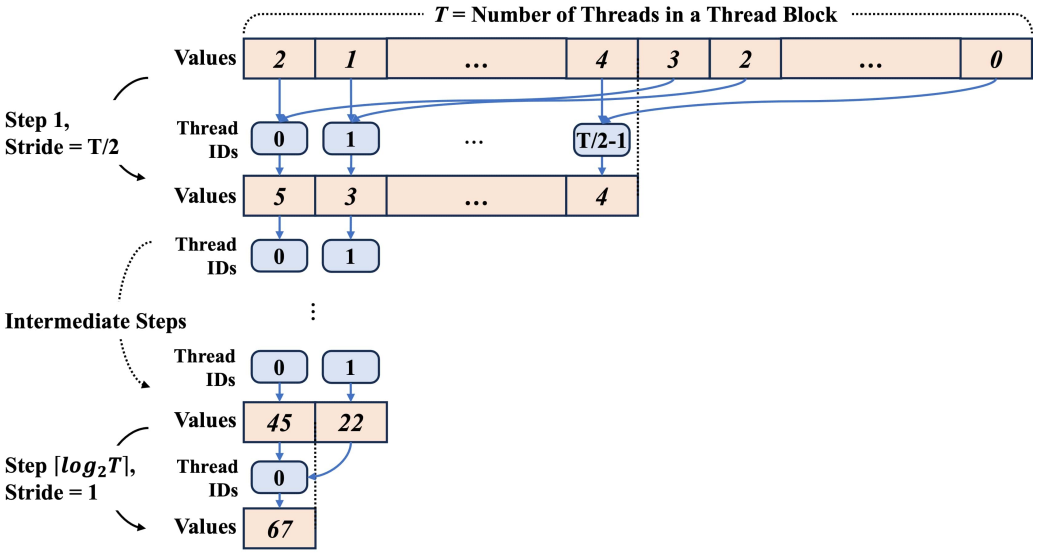


Figure 5. Example of calculating the complete HD value based on the divide and conquer strategy.

3.4.3. Calculation of LRCS

The values of LRCS are calculated for all pairs of CDSs in a solution, as well as for each individual CDS. Assuming that the length of an CDS is L, our approach to calculate the LRCS is based on the dynamic programming algorithm in Eq (5).

$$m[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ m[i-1, j-1] + 1, & \text{if } CDS_1[i] = CDS_2[j] \\ 0, & \text{if } CDS_1[i] \neq CDS_2[j] \end{cases} \tag{5}$$

The algorithm utilizes an $(L+1)\times(L+1)$ matrix denoted as m, where CDS_1 and CDS_2 are matched in rows and columns, respectively. To simplify the computations, the 0-th row and 0-th column of the matrix, $m[0,*]$ and $m[*,0]$, respectively, are padded with 0. In Eq (5), each element of $m[i, j]$ represents the length of the common substring obtained by comparing up to the i-th base of CDS_1 and the j-th base of CDS_2. For example, if the element of $m[i, j]$ is 3, it indicates that the substring from the $(i-2)$-th base to the i-th base of CDS_1 and the substring from the $(j-2)$-th base to the j-th base of CDS_2 are matched.

Since the common substring is contiguous, if the i-th base of CDS_1 and the j-th base of CDS_2 are identical when calculating the element $m[i, j]$, the element $m[i-1, j-1]$ is needed, which represents the length of the common substring obtained by comparing up to the $(i-1)$-th base of CDS_1 and the $(j-1)$-th base of CDS_2. Therefore, the element $m[i, j]$ is computed by adding 1 to the element $m[i-1, j-1]$. On the other hand, if the i-th base of CDS_1 and the j-th base of CDS_2 are different, it indicates that the common substring is interrupted at that position. Consequently, the value 0 is stored in $m[i, j]$, indicating that the element $m[i-1, j-1]$ may be required to compute the element $m[i, j]$. For example, let us consider the calculation of the element $m[3, 3]$ using the CDS_1 "AUGGCU" and CDS_2 "AUGGCC" as shown in Figure 6. To determine $m[3, 3]$, we compare the *3*rd base of the CDS_1, which

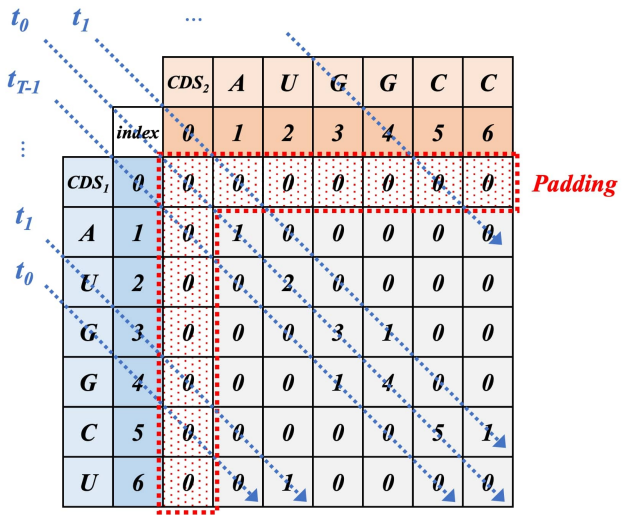


Figure 6. Example of calculating the longest length of substring.

is "G", with the 3rd base of the CDS_2, which is also "G". Since they are identical, we store the value of $m[2, 2]$ plus 1 in $m[3, 3]$. When calculating LRCS within the same CDS, if the indices of the base pair are the same, that is, when the row index and column index of the matrix element are the same, the element is computed as 0.

In Eq (5), it is important to consider the dependency. In order to calculate an element in the matrix $m[i, j]$, we need the element $m[i-1, j-1]$, which means that the element $m[i-1, j-1]$ must be calculated before calculating the element $m[i, j]$, as shown in Figure 6. The thread responsible for elements along a column $m[*, j]$ must wait for the thread handling elements along the column $m[*, j-1]$ to complete. In the SIMT architecture, this synchronization increases the execution time because threads within a warp work together. To minimize synchronization between threads, each thread performs its own calculations diagonally, as shown in Figure 6. Diagonals in the matrix are assigned to threads sequentially, starting from the bottom left and moving towards the top right, ensuring that each thread can work independently. Furthermore, due to the dependency, each thread does not need to store the entire matrix. Instead, each thread uses a single register file to store the maximum LRCS value per diagonal. Once all threads complete their calculations, they store their intermediate values of LRCS in shared memory in order to find the maximum LRCS value per matrix (a pair of CDSs). Similar to the other two objective functions, the divide-and-conquer strategy is also used to find the maximum LRCS value between a pair of CDSs. By calculating the maximum LRCS values of all pairs of CDSs as well as individual CDSs in a solution, we can determine the maximum values of LRCS denoted as $MLRCS$ in the solution.

3.5. Non-dominated sorting and crowding distance sorting

In the selection step, a total of $2N$ solutions composed of N original solutions and N new solutions are obtained from the mutation step. For selecting N solutions out of the $2N$ solutions, non-dominated sorting is performed first. In our implementation, the CUDA kernel for the selection step requires $2N$ threads because each thread handles a single solution. Although the maximum number of threads per block is normally 1024 in CUDA, we set the number of threads per block as 128 ensure better

occupancy, which is the ratio of active warps to the maximum number of warps supported on an SM (streaming multiprocessor) of the GPUs. Furthermore, each thread compares the values of the multi-objective functions (*mHD*, *mCAI* and *MLRCS*) with those of all others. Therefore, at the end of the mutation step, the values of the objective functions are stored in global memory, which is accessible by all threads on a grid.

First, each thread performs the Pareto comparison with the remaining $(2N - 1)$ solutions in order to identify a set of solutions with a rank of 0. In this process, n_p and S_p are calculated for each solution, where n_p indicates the number of other solutions that dominate the current solution and S_p is the set of solutions that are dominated by the current solution. Solutions with n_p equal to 0 are considered in the set of solutions with rank 0. This is because if n_p is 0, there are no other solutions that dominate the current solution. To determine the set of solutions with rank 1, we need to examine the solutions in the set S_p by traversing the solutions with rank 0. When visiting a solution in the set S_p of solutions with rank 0, its n_p is decreased by 1. After traversing all solutions with rank 0, we can identify the solutions in the set S_p of solutions with rank 0 that have $n_p = 0$. These solutions were only dominated by solutions with rank 0 and therefore they are assigned a rank of 1. Similarly, to find the solutions in the next rank, we visit S_p set of solutions in the previous rank and decrease the n_p value of the solutions in S_p by 1.

In our implementation, we use a $2N \times 2N$ Boolean matrix to represent the S_p set consisting of $2N$ solutions. In this matrix, each row represents the set S_p of a solution and the value of each column indicates whether the corresponding solution is dominated by the solution in the respective row. Consequently, each thread independently accesses different columns when visiting the set S_p of each solution. Furthermore, the crowding distance sorting is utilized to sort solutions with identical ranks. When selecting N solutions from $2N$ solutions, there may be cases where it is necessary to choose solutions from a group with the same rank, as shown in Figure 3. In such cases, an alternative method is required to sort solutions with identical ranks because the three objective functions cannot be utilized for sorting. After each thread computes the crowding distance(s) according to Section 2.2, the solutions are sorted in descending order based on their the crowding distance. For this purpose, we employ bitonic mergesort, which is a parallel algorithm with a time complexity of $O(log^2 n)$.

4. Experimental results and discussion

In our experiments, we used an AMD Ryzen 7 5800X 8-Core processor and an NVIDIA GeForce RTX 4090 (128 Ada SMs, 128 CUDA cores/SM, 24GB global memory). The implementation was compiled using GCC 11.3 and CUDA 12.1, with the driver version was 530.30.02, running on Ubuntu 22.04.2 LTS. For both the initialization and mutation CUDA kernels, we used 512 threads per thread block. This configuration resulted in the highest occupancy, considering the required shared memory and the number of register files as well as achieving the fastest execution time for the protein instances shown in Table 1.

On the other hand, in the selection CUDA kernel we used 128 threads per thread block, resulting in the highest occupancy. Among the four CUDA kernels, namely (1) random number generator, (2) initialization, (3) mutation, and (4) selection, the first two kernels were executed only once. On the other hand, the third and fourth CUDA kernels accounted for over 99% of the execution time because applying mutations and calculating the multi-objective functions are time-consuming tasks, which are

Table 1. Protein Instances Used in the Experiment.

Code	Name	CDSs	Length(AA)	CDS × Length
Q5VZP5	DUS27_HUMAN	2	1158	2316
A4Y1B6	FADB_SHEPC	3	716	2148
B3LS90	OCA5_YEAS1	4	679	2716
B4TWR7	CAIT_SALSV	5	505	2525
Q91X51	GORS1_MOUSE	6	446	2676
Q89BP2	DAPE_BRADU	7	388	2716

also repeated multiple times.

Unfortunately, we could not access to the implementations of Gonzalez-Sanchez et al.'s method, AP-MOABC and NSGA-II-CPU. Therefore, we relied on published experimental results [28] to compare our approach with theirs. This comparison demonstrated the accuracy of our implementation. According to [28], their experimental results are based on CPUs and its experimental results were limited to 100 cycles and 128 solutions. In Section 4.1, we therefore provide limited experimental results using the same parameters (100 cycles and 128 solutions) to facilitate a direct comparison with their methods. However, due to the improved computing performance achieved through parallel data processing in our implementation, we also present expanded experimental results with varying numbers of cycles and solutions ranging from 25 to 1600 cycles and from 128 to 2048 solutions, respectively, in Section 4.2.

Table 2. Nadir and ideal points for the calculation of quality indicators.

Objective functions	Nadir value	Ideal value
mCAI	0	1
mHD	0	0.4
MLRCS	1	0

In our experiments, we used the protein instances listed in Table 1. Additionally, Table 2 provides the nadir value and ideal value needed for the calculation of quality indicators. These indicators, such as the hypervolume indicator and the minimum distance to the ideal point, are widely used to evaluate the quality of solutions in multi-objective optimization problems. The amino acid sequences of the target proteins were obtained from the UniProt databases [31]. Furthermore, for calculating the *CAI* values of the CDSs, we utilized previously published codon usage frequencies [23].

4.1. Limited experiments with 100 cycles and 128 solutions

In this section, we compare the following three methods using a limited number of cycles (100) and solutions (128): our method, AP-MOABC (the method of parallel execution of the MOABC using OpenMP) and NSGA-II-CPU. All three methods were executed with the same parameters: 100 cycles, 128 solutions and 5% mutation probability. Our implementation was run on NVIDIA RTX 4090, while the experimental results of AP-MOABC and NSGA-II-CPU were obtained from published evaluations [28], which were conducted on four 16-core AMD Opteron Abu Dhabi 6376 processors (64 [16 × 4] physical cores, 2.3 GHz) with 96 GB DDR3 RAM. Although it is expected that their execution times

would improve if we were able to run them on faster CPUs, their execution times would still be more than a second.

Table 3. Hypervolume results and minimum distances to the ideal points.

	Hypervolume results			Minimum distances to the ideal points		
Protein	Our method	AP-MOABC	NSGA-II-CPU	Our method	AP-MOABC	NSGA-II-CPU
Q5VZP5	60.01%	59.27%	59.92%	0.508467	0.489408	0.503676
A4Y1B6	51.04%	52.71%	52.53%	0.569039	0.542613	0.551986
B3LS90	54.79%	55.59%	54.62%	0.524125	0.512751	0.512885
B4TWR7	48.67%	49.79%	48.91%	0.583531	0.563227	0.574876
Q91X51	50.88%	52.02%	50.47%	0.582617	0.574168	0.589626
Q89BP2	48.83%	50.09%	48.61%	0.593861	0.565618	0.569445
Average	52.37%	53.25%	52.51%	0.560273	0.541298	0.550416

As shown in Table 3, for most proteins, AP-MOABC slightly outperformed both our method and NSGA-II-CPU, excluding the Q5VZP5 protein. The average hypervolume of AP-MOABC was 53.25%, while our method achieved 52.37% and NSGA-II-CPU achieved 52.51%. As for the average minimum distance to the ideal points, AP-MOABC achieved 0.541298, our method achieved 0.560273 and NSGA-II-CPU achieved 0.550416. Since our method is based on NSGA-II-CPU, the experimental results of our method was similar to thosed of NSGA-II-CPU when using the same number of cycles and solutions, in terms of hypervolume and minimum distance to the ideal point. Table 4 shows the execution time for our method and AP-MOABC. For the protein A4Y1B6, our method presented the most significant speedup of 229.2× times faster than AP-MOABC. Although AP-MOABC exhibited a performance improvement of 33.28× compared to MOABC on four 16-core CPUs, our method attained a speedup of 187.5× on average compared to AP-MOABC. This indicates that our method can quickly catch up with the outputs of AP-MOABC, hypervolume and minimum distance by increasing the number of cycles as the execution time of our method on GPUs was significantly faster, although the outputs of our method were inferior to those of AP-MOABC.

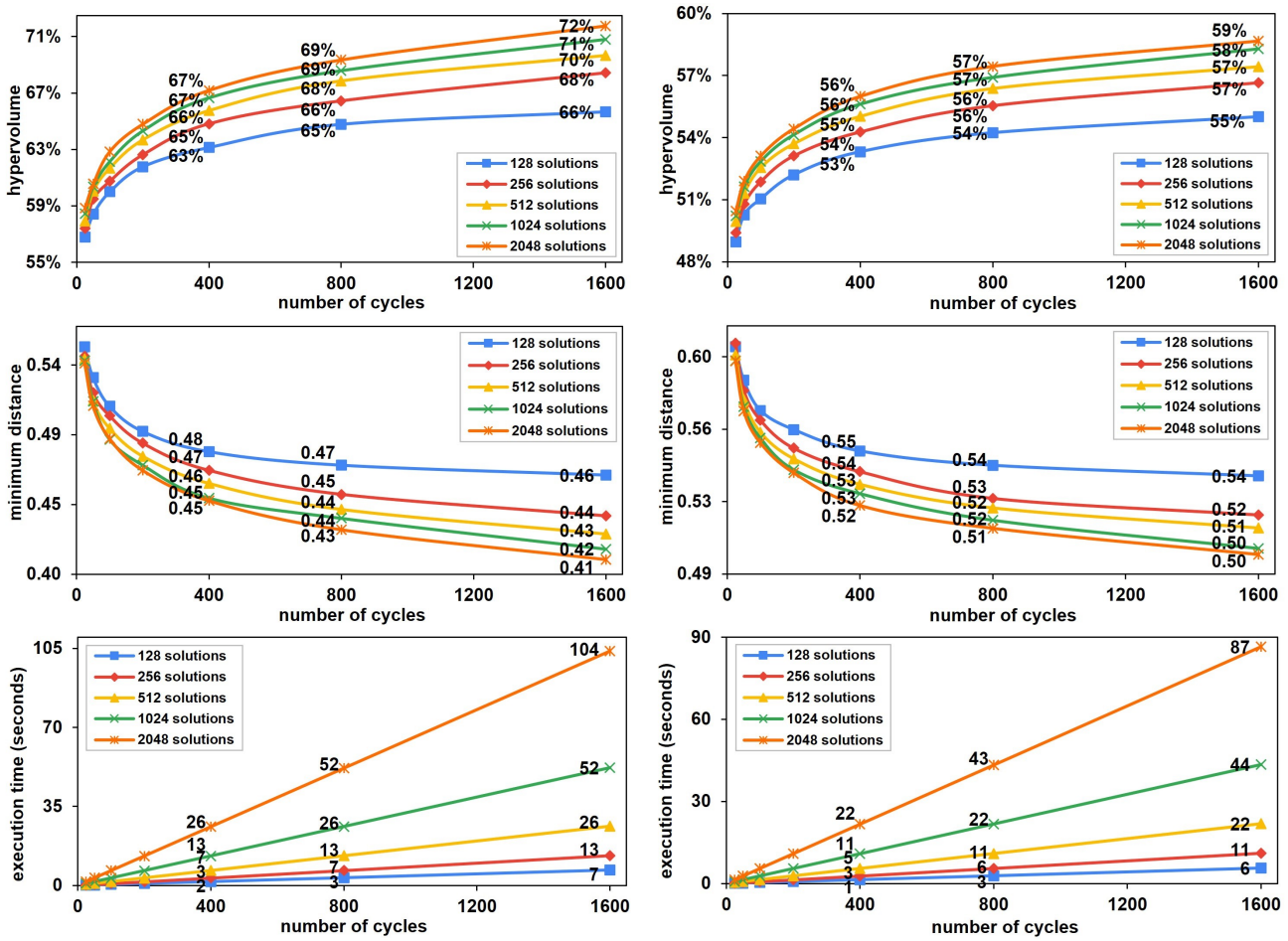
Table 4. Execution times (in seconds).

Protein	Our method	AP-MOABC
Q5VZP5	0.4284	87.937
A4Y1B6	0.3574	81.924
B3LS90	0.5217	100.989
B4TWR7	0.4672	88.211
Q91X51	0.5246	92.550
Q89BP2	0.5648	85.567
Average	0.4774	89.530

4.2. Expanded experiments with varying solutions and cycles

In this section, we present the expanded experimental results of our implementation with varying numbers of solutions and cycles. As shown in Table 4, because the execution times of our method were extremely short, we were able to test our method with various numbers of cycles and solutions in order

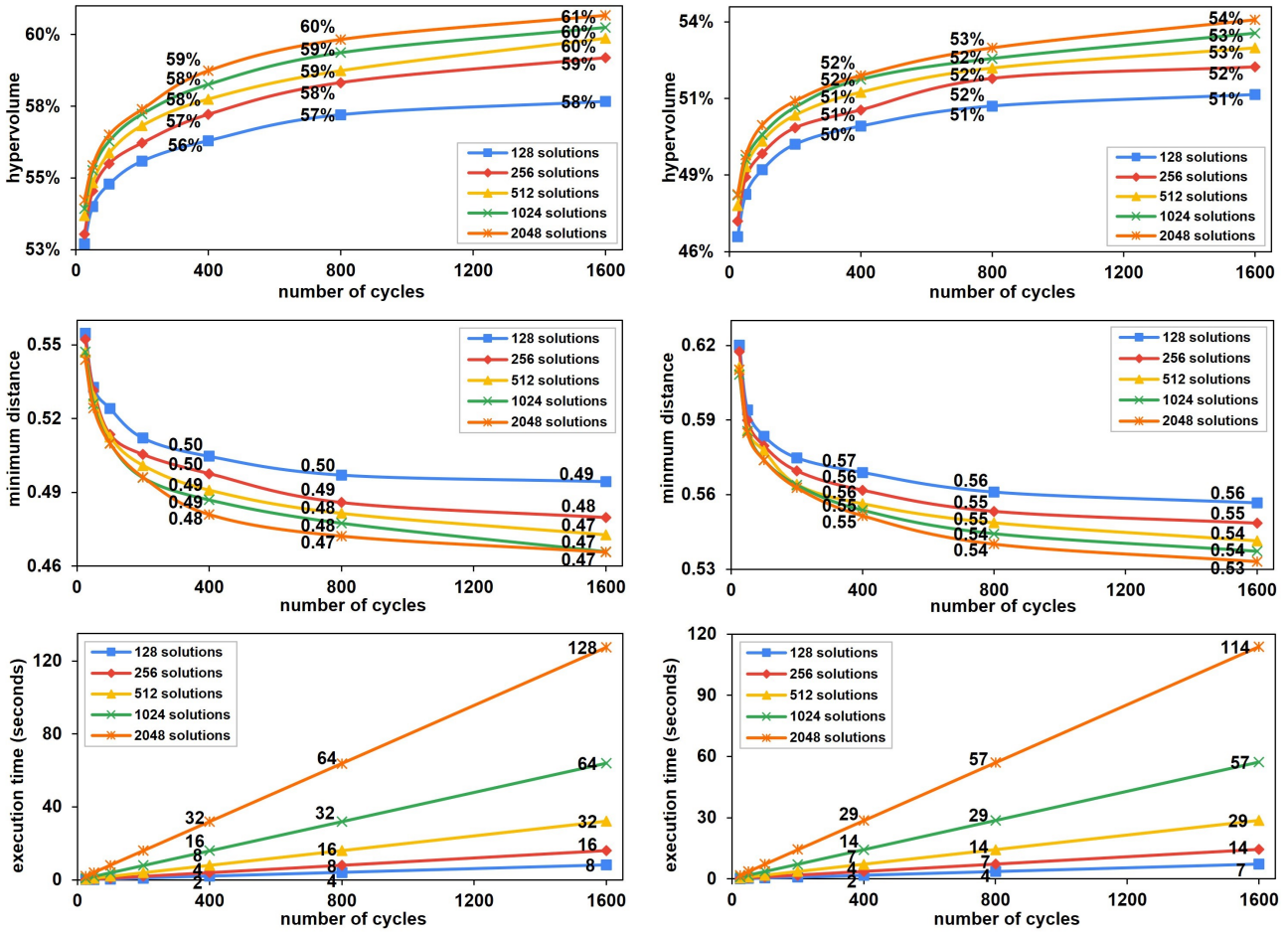
to demonstrate that our implementation has considerable potential in practice: 25, 50, 100, 200, 400, 800 and 1600 cycles, coupled with 128, 256, 512, 1024 and 2048 solutions, resulting in 35 cases for each protein. In addition to, we used a mutation probability of 5%. The experiments were performed 10 times, and the average results are presented.



(a) Q5VZP5 protein **(b)** A4Y1B6 protein

Figure 7. Experimental results of our method with Q5VZP5 7(a) and A4Y1B6 7(b).

Figures 7–9 show the hypervolume values, minimum distance values and executions times of our method with varying number of solutions and cycles. Figure 7 presents the experimental results for the proteins Q5VZP5 and A4Y1B6. The proteins B3LS90 and B4TWR7 are shown in Figure 8, while Q91X51 and Q89BP2 are shown in Figure 9. In Figures 7–9, the x-axis represents the number of cycles and the y-axis represents the hypervolume, minimum distance or execution time. The legend indicates the number of solutions. Increasing the number of solutions and cycles improved the hypervolume and minimum distance for all proteins. However, the hypervolume and minimum distance did not proportionally increase with increasing number of solutions and cycles. The trends appeared to approach their upper bounds, such as the Pareto fronts. For all proteins, the hypervolumes increased rapidly at the beginning of the cycles, followed by gradual increases. Likewise, the values of the minimum distance to ideal point rapidly decreased initially, followed by gradual decreases. For example, let us

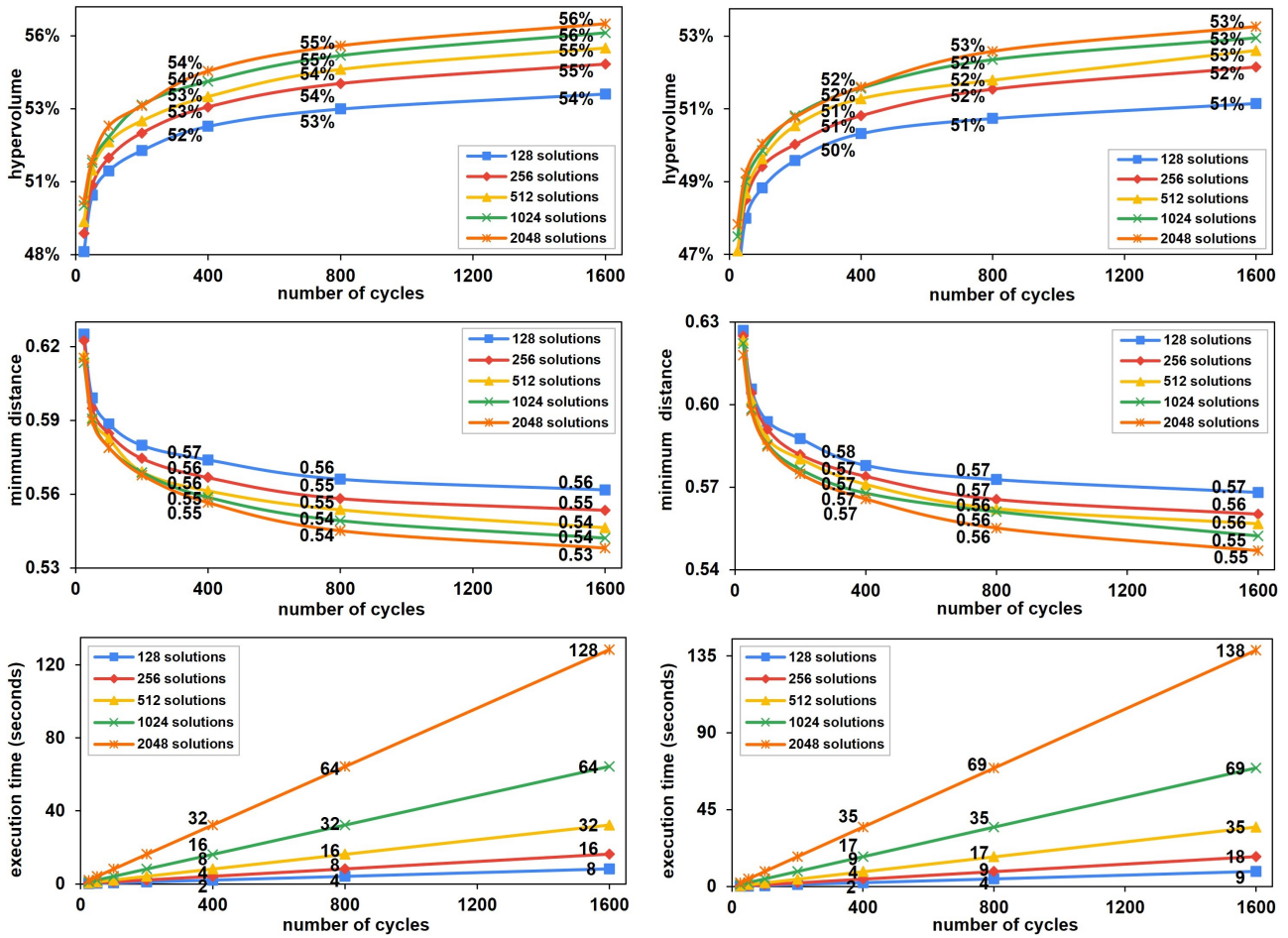


(a) B3LS90 protein **(b)** B4TWR7 protein

Figure 8. Experimental results of our method on B3LS90 8(a) and the B4TWR7 8(b).

consider the A4Y1B6 protein with 128 solutions. When the number of cycles increased from 25 to 50, the hypervolume improved by 1.32% and the minimum distance improved by 0.0162345. In contrast, when the number of cycles increased from 800 to 1600, the hypervolume improved by 0.77% and the minimum distance improved by 0.0051251. Thus, the rate of improvement in hypervolume and minimum distance was higher at the initial cycles compared to later cycles. However, the execution time showed a linear increase in proportion to the number of solutions and cycles.

In our method, doubling either the number of solutions or the number of cycles approximately doubled the execution time. However, increasing the number of solutions or cycles did not result in similar improvements in the hypervolume and minimum distance. In most of our experiments, increasing the number of cycles had a greater impact on improving both the hypervolume and minimum distance compared to increasing the number of solutions. For example, let us consider the Q5VZP5 protein. When the number of cycles was 25 and the number of solutions was 128, the hypervolume was 56.80%, the minimum distance was 0.546836 and the execution time was 0.111 seconds. Doubling the number of cycles to 50 while maintaining 128 solutions resulted in a hypervolume of 58.42%, a minimum distance of 0.527110 and an execution time of 0.216 seconds. However, when we doubled the number



(a) Q91X51 protein **(b)** Q89BP2 protein

Figure 9. Experimental results of our method with Q91X51 9(a) and Q89BP2 9(b).

of solutions to 256 while maintaining 25 cycles, the hypervolume was 57.42%, the minimum distance was 0.540800 and the execution time was 0.214 seconds. Therefore, increasing number of cycles to 50 with 128 solutions yielded better results compared to doubling the number of solutions. Let us consider the same protein, Q5VZP5, for another example. With 800 cycles and 1024 solutions, the hypervolume was 68.59%, the minimum distance was 0.436226 and the execution time was 26.143 seconds. Doubling the number of cycles to 1600 with 1024 solutions resulted in a hypervolume of 70.80%, a minimum distance of 0.416322 and an execution time of 52.255 seconds. On the other hand, when we doubled the number of solutions to 2048 while maintaining 800 cycles, the hypervolume was 69.34%, the minimum distance was 0.428807 and the execution time was 51.973 seconds.

However, there were cases where increasing the number of solutions was more effective than increasing the number of cycles. Let us take the Q5VZP5 protein as an example. With 800 cycles and 128 solutions, the hypervolume was 64.79%, the minimum distance was 0.470478 and the execution time was 3.398 seconds. Doubling the number of cycles to 1600 while maintaining 128 solutions resulted in a hypervolume of 65.68%, a minimum distance of 0.464115 and an execution time of 6.801 seconds. On the other hand, when we doubled the number of solutions to 256 while maintaining 800

cycles, the hypervolume was 66.44%, the minimum distance was 0.451541 and the execution time was 6.645 seconds. It seems that when the number of solutions is too small compared to the number of cycles, increasing the number of solutions leads to better improvements in hypervolume and minimum distance than increasing the number of cycles. Therefore, while increasing the number of cycles generally improves the hypervolume and minimum distance, when the number of solutions is insufficient compared to the number of cycles, increasing the number of solutions is beneficial. This pattern was not only limited to the Q5VZP5 protein but was also observed with other proteins in our experiments.

Table 5. Experimental results for the protein Q5VZP5 on different solutions with similar execution times to AP-MOABC.

Solutions	Cycles	Hypervolume	Min. distance	Time
128	20730	68.81%	0.449367	88.023
256	10600	71.80%	0.406582	87.922
512	5350	72.54%	0.404609	87.691
1024	2670	72.00%	0.406844	87.043
2048	1350	71.21%	0.412504	87.513

To determine the optimal number of solutions among 128, 256, 512, 1024 and 2048, we conducted tests with each protein, aiming for similar execution times as the AP-MOABC method [28]. Table 5 presents the experimental results for protein Q5VZP5 with indicated numbers of solutions. Notably, when using 512 solutions, we achieved the best results in terms of hypervolume (72.54%) and minimum distance (0.404609). The corresponding number of cycles for this configuration was 5350 with an execution time of 87.691 seconds. Remarkably, the use of 512 solutions consistently provided the best results for both hypervolume and minimum distance across other proteins as well.

Table 6. Experimental results of our method (512 solutions) and AP-MOABC (128 solutions and 100 cycles).

	Our method				AP-MOABC		
Protein	Cycles	Hypervolume	Min. distance	Time	Hypervolume	Min. distance	Time
Q5VZP5	5350	72.54%	0.404609	87.691	59.27%	0.489408	87.937
A4Y1B6	5950	59.83%	0.488580	81.343	52.71%	0.542613	81.924
B3LS90	5000	61.03%	0.459070	100.304	55.59%	0.512751	100.989
B4TWR7	4900	53.88%	0.530695	87.820	49.79%	0.563227	88.211
Q91X51	4550	56.30%	0.529353	91.853	52.02%	0.574168	92.550
Q89BP2	3900	53.51%	0.548313	84.911	50.09%	0.565618	85.567
Average	4942	59.52%	0.493436	88.987	53.25%	0.541298	89.530

Table 6 shows the experimental results for each protein when the number of solutions was 512 and the execution times were similar those of AP-MOABC. For the Q5VZP5 protein, our method achieved a hypervolume that was 13.27% larger and a minimum distance value that was 0.084799 smaller compared to AP-MOABC. Similarly, for the A4Y1B6 protein, our method resulted in a hypervolume that was 7.12% larger and a minimum distance that was 0.054133 smaller compared to AP-MOABC. Additionally, our method outperformed AP-MOABC the B3LS90, B4TWR7, Q91X51 and Q89BP2 proteins as well. Therefore, when execution times were similar to AP-MOABC, our

method consistently delivered better results compared to AP-MOABC and utilizing 512 solutions was the the optimal choice among 128, 256, 512, 1024 and 2048.

5. Related works

Several tools have been developed for designing CDSs to increase protein production, such as COOL (codon optimization online) [32], D-Tailor (DNA-Tailor) [33] and OPTIMIZER [34]. While they take into account the CAI (codon adaptation index) of the host organism during the CDS design, they do not consider the similarity between CDSs. However, Terai et al. [14] proposed a method based on NSGA-II and NSGA-II-CPU, that considers both the CAI and CDS similarity during the design process. In addition, Gonzalez-Sanchez et al. proposed an ABC (artificial bee colony) algorithm-based method, MOABC, with the same objective functions as Terai et al. In their subsequent research [28], they accelerated the method using the parallel programming API called OpenMP [26], AP-MOABC. The main difference between NSGA-II-CPU and Gonzalez-Sanchez et al.'s method lies in how they handle solutions in the generation. Although the latest work of Gonzalez-Sanchez et al. improves the quality of solutions in terms of hypervolume and minimum distance, which are used to scalarize a multi-objective optimization problem, it requires approximately twice the number of computations of the multi-objective functions (and the mutations) within a generation.

Furthermore, there are other recent and relevant studies such as MOBOA (multi-objective butterfly optimization algorithm) [35] and MaOMPE (many-objective mutation-based protein encoding) [36] for designing CDSs to increase protein production. However, they use objective functions that are different from ones used in our method and they are also not parallelized. MaOMPE was based on NSGA-III [37] that is extended from NSGA-II to handle problems with a large number of objective functions. Although MOBOA is similar to NSGA-II, it differs in the introduciton of special structures called BestList that stores good six solutions which can be used multiple times to generate new N solutions and TabooList that stores solutions which cannot be included in the BestList.

5.1. Method based on MOABC

MOABC is multi-objective optimization algorithm based on the swarm intelligence of ABC (artificial bee colony). Gonzalez-Sanchez et al. [25] proposed a method based on MOABC to design a set of CDSs to encode proteins. Similar to NSGA-II-CPU, this method initializes N solutions and does not perform the crossover operation. The generation process of MOABC consists of three steps based on ABC: (1) employed bees step, (2) onlooker bees step and (3) scout bees step. In the employed bees step, N new solutions are generated from the N original solutions using mutation operators, respectively. Then, the Pareto comparison is performed between an original solution and a corresponding mutated solution. If the mutated solution dominates its corresponding original solution, the original solution is replaced with the mutated solution. Prior to the onlooker bees step, non-dominated sorting and crowding distance sorting are applied to the N solutions from the employed bees step and the selection probability of each solution is calculated. Lower ranks and the higher crowding distances lead to higher selection probabilities. These N solutions are then used as the input population for the onlooker bees step. In the onlooker bees step, N additional solutions are generated according to the selection probabilities of the N input solutions, resulting in $2N$ solutions. Solution with relatively high selection probabilities have a greater chance of being selected multiple times to generate new solutions.

However, if a newly generated solution does not dominate its original solution, the original solution is retained as a new solution. Thus, there may be multiple instances of the same solutions. Finally, these $2N$ solutions serve as the input population for the scout bees step. In the scout bees step, the solutions that failed to generate good solutions predefined *limit* times are replaced with new solutions, which are randomly generated and then mutated in proportion to the number of current cycles. After the scout bees step, non-dominated sorting and crowding distance sorting are performed on the set of $2N$ solutions. Then, N solutions are selected and used as the input population for the subsequent generation. Therefore, in terms of calculating the multi-objective functions and performing non-dominated sorting and crowding distance sorting, the MOABC-based method requires $2\times$ and $1.5\times$ more computations per cycle, respectively, compared to our method and NSGA-II-CPU.

5.2. Method based on MOBOA

In the method based on MOBOA, GC3 is introduced to indicate the amount of guanine or cytosine at the third position of codons and it is associated with gene stability. As gene stability increases, protein expression level also increase. Instead of LRCS, the method based on MOBOA optimizes CAI, HD and GC3 objective functions based on the intelligent foraging behavior of butterflies for designing a set of CDSs. This method is similar to our method and NSGA-II-CPU in that it has a cycle of generating N new solutions from N original solutions, performing non-dominated sorting and crowding distance sorting and the selected N solutions are used as N original solutions in the next cycle. However, this method accumulates the solutions with a rank value of 0 in each cycle in a file. Furthermore, this method mutates either one of the solutions in the BestList or one solution among N original solutions for generating one new solution. The BestList contains six good solutions selected from the N original solutions, excluding solutions in the TabooList. When a new solution is generated mutating the solution in the BestList, if the generated solution dominates its original solution, the original solution is moved from the BestList to the TabooList. Therefore, the TabooList is accumulated with solutions that are not included in the BestList. The BestList is filled with the next best solution from the N original solutions. Because checking the TabooList is needed to select the solution for the BestList, this incurs additional cost compared with our method.

5.3. Method based on MaOMPE

MaOMPE optimizes CAI, HD, GC3 and SL (stem length) objective functions based on NSGA-III for designing CDSs. The SL is the longest length within a CDS that can potentially induce hairpin loop formation that decreases protein expressin levels. If this objective function value is high, the probability of hairpin loop occurrence increases. This method is very similar to the NSGA-II approach. One difference is that in the non-dominated sorting and crowding distance sorting, the crowding distance sorting for selecting the better solution among solutions with the same rank is replaced with reference point-based sorting. The reference points are initialized using the Das and Dennis [38] method.

6. Conclusions

In this paper, we proposed an approach to accelerate the process of designing a set of CDSs for achieving high protein expression levels on NVIDIA GPUs. Our implementation optimized the method based on NSGA-II proposed by Terai et al. by efficiently performing the multi-objective functions and

mutation operations on GPUs. We optimized the three objective functions to ensure efficient execution on GPUs as they are the main bottleneck in designing of a set of CDSs with satisfactory protein expression levels. Experimental results demonstrated that our approach achieved significant performance improvement compared to others. We further analyzed the impacts of varying the numbers of cycles and solutions for designing multiple genes encoding the same protein. The open-source software developed in this study is available at: https://github.com/CAU-HPCL/CUDA-protein.

7. Future work

Multiple multi-objective optimization algorithms have been used to solve problems across different domains. For example, Dulebenets [39] addressed the truck scheduling problem for reducing the operational cost of a cross-docking terminal (CDT) using the adaptive polyploid memetic algorithm (APMA). Pasha et al. [40] proposed a hybrid multi-objective evolutionary algorithm (HMOEA) for the vehicle routing problem with a factory-in-a-box, which is crucial in urgent scenarios such as COVID-19 for the production and distribution of goods. Gholizadeh et al. [41] implemented a modified scenario-based GA (msb-GA) to optimize preventive maintenance schedules in a waste-to-energy production system. Dulebenets *et al.* [42] proposed a self-adaptive evolutionary algorithm (SAEA) to solve the berth scheduling problem. Zhao and Zhang [43] designed the online-learning-based reference vector evolutionary many-objective algorithm (RVMEA/OL) to optimize problems with many-objective functions and this approach is similar to the traditional evolutionary many-objective optimization algorithms (EMaOAs), but it incorporates a dynamic mutation strategy based on feedback about the optimization level of each sub-problem during the optimization process.

In future works, we plan to extend the application of our approach to include other objective functions commonly used in various multi-objective optimization problems in bioinformatics. We will achieve this by parallelizing them for efficient execution on GPUs. Additionally, we aim to make potential improvements by dynamically adjusting mutation probabilities instead of a static mutation probability. This adjustment will help alleviate the impact of other objective functions when one objective function is revised. Furthermore, we will revise mutation operators that tend to intentionally improve an objective function simultaneously with others. Finally, our objective is to develop an open-source framework on GPUs that can facilitate high-performance biological experiments in computational biology.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgments

This research was supported in part by the Chung-Ang University Research Grants in 2021 and in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2022R1G1A1013586).

Conflict of interest

The authors declare there is no conflicts of interest.

References

1. S. Fields, O. Song, A novel genetic system to detect protein–protein interactions, *Nature*, **340** (1989), 245–246. https://doi.org/10.1038/340245a0
2. S. Varambally, S. M. Dhanasekaran, M. Zhou, T. R. Barrette, C. Kumar-Sinha, M. G. Sanda, et al., The polycomb group protein ezh2 is involved in progression of prostate cancer, *Nature*, **419** (2002), 624–629. https://doi.org/10.1038/nature01075
3. G. Blander, L. Guarente, The sir2 family of protein deacetylases, *Annu. Rev. Biochem.*, **73** (2004), 417–435. https://doi.org/10.1146/annurev.biochem.73.011303.073651
4. S. P. Kaur, V. Gupta, Covid-19 vaccine: A comprehensive status report, *Virus Res.*, **288** (2020), 198114. https://doi.org/10.1016/j.virusres.2020.198114
5. M. Ahmad, M. Hirz, H. Pichler, H. Schwab, Protein expression in pichia pastoris: Recent achievements and perspectives for heterologous protein production, *Appl. Microbiol. Biotechnol.*, **98** (2014), 5301–5317. https://doi.org/10.1007/s00253-014-5732-5
6. D. Fouque, K. Kalantar-Zadeh, J. Kopple, N. Cano, P. Chauveau, L. Cuppari, et al., A proposed nomenclature and diagnostic criteria for protein–energy wasting in acute and chronic kidney disease, *Kidney Int.*, **73** (2008), 391–398. https://doi.org/10.1038/sj.ki.5002585
7. A. D. Bandaranayake, S. C. Almo, Recent advances in mammalian protein production, *FEBS Lett.*, **588** (2014), 253–260.
8. J. Dehghani, A. Movafeghi, E. Mathieu-Rivet, N. Mati-Baouche, S. Calbo, P. Lerouge, et al., Microalgae as an efficient vehicle for the production and targeted delivery of therapeutic glycoproteins against sars-cov-2 variants, *Marine Drugs*, **20** (2022), 657. https://doi.org/10.3390/md20110657
9. S. C. Spohner, H. Müller, H. Quitmann, P. Czermak, Expression of enzymes for the usage in food and feed industry with pichia pastoris, *J. Biotechnol.*, **202** (2015), 118–134. https://doi.org/10.1016/j.jbiotec.2015.01.027
10. A. Haldimann, B. L. Wanner, Conditional-replication, integration, excision, and retrieval plasmid-host systems for gene structure-function studies of bacteria, *J. Bacteriol.*, **183** (2001), 6384–6393.
11. P. Gu, F. Yang, T. Su, Q. Wang, Q. Liang, Q. Qi, A rapid and reliable strategy for chromosomal integration of gene (s) with multiple copies, *Sci. Rep.*, **5** (2015), 1–9. https://doi.org/10.1038/srep09684
12. C. A. Scorer, J. J. Clare, W. R. McCombie, M. A. Romanos, K. Sreekrishna, Rapid selection using g418 of high copy number transformants of pichia pastoris for high–level foreign gene expression, *Nat. Biotechnol.*, **12** (1994), 181–184. https://doi.org/10.1038/nbt0294-181
13. K. E. Tyo, P. K. Ajikumar, G. Stephanopoulos, Stabilized gene duplication enables long-term selection-free heterologous pathway expression, *Nat. Biotechnol.*, **27** (2009), 760–765. https://doi.org/10.1038/nbt.1555

14. G. Terai, S. Kamegai, A. Taneda, K. Asai, Evolutionary design of multiple genes encoding the same protein, *Bioinformatics*, **33** (2017), 1613–1620. https://doi.org/10.1093/bioinformatics/btx030

15. A. Vassileva, D. A. Chugh, S. Swaminathan, N. Khanna, Expression of hepatitis b surface antigen in the methylotrophic yeast pichia pastoris using the gap promoter, *J. Biotechnol.*, **88** (2001), 21–35. https://doi.org/10.1016/S0168-1656(01)00254-1

16. R. Aw, K. M. Polizzi, Can too many copies spoil the broth?, *Microbial cell factories*, **12** (2013), 1–9. https://doi.org/10.1186/1475-2859-12-128

17. J. M. Buerstedde, N. Lowndes, D. G. Schatz, Induction of homologous recombination between sequence repeats by the activation induced cytidine deaminase (aid) protein, *Elife*, **3** (2014), e03110. https://doi.org/10.7554/eLife.03110

18. J. Jurka, P. Klonowski, V. Dagman, P. Pelton, Censor–a program for identification and elimination of repetitive elements from dna sequences, *Comput. Chem.*, **20** (1996), 119–121. https://doi.org/10.1016/S0097-8485(96)80013-1

19. J. Athey, A. Alexaki, E. Osipova, A. Rostovtsev, L. V. Santana-Quintero, U. Katneni, et al., A new and updated resource for codon usage tables, *BMC Bioinf.*, **18** (2017), 1–10. https://doi.org/10.1186/s12859-017-1793-7

20. J. M. Comeron, M. Aguadé, An evaluation of measures of synonymous codon usage bias, *J. Mol. Evol.*, **47** (1998), 268–274. https://doi.org/10.1007/PL00006384

21. M. Gouy, C. Gautier, Codon usage in bacteria: correlation with gene expressivity, *Nucleic Acids Res.*, **10** (1982), 7055–7074. https://doi.org/10.1093/nar/10.22.7055

22. T. Ikemura, Correlation between the abundance of escherichia coli transfer rnas and the occurrence of the respective codons in its protein genes: A proposal for a synonymous codon choice that is optimal for the E. coli translational system, *J. Mol. Biol.*, **151** (1981), 389–409. https://doi.org/10.1016/0022-2836(81)90003-6

23. P. M. Sharp, W. H. Li, The codon adaptation index-a measure of directional synonymous codon usage bias, and its potential applications, *Nucleic Acids Res.*, **15** (1987), 1281–1295. https://doi.org/10.1093/nar/15.3.1281

24. K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, *IEEE Trans. Evol. Comput.*, **6** (2002), 182–197. https://doi.org/10.1109/4235.996017

25. B. Gonzalez-Sanchez, M. A. Vega-Rodríguez, S. Santander-Jiménez, J. M. Granado-Criado, Multi-objective artificial bee colony for designing multiple genes encoding the same protein, *Appl. Soft Comput.*, **74** (2019), 90–98. https://doi.org/10.1016/j.asoc.2018.10.023

26. L. Dagum, R. Menon, Openmp: An industry standard api for shared-memory programming, *IEEE Comput. Sci. Eng.*, **5** (1998), 46–55. https://doi.org/10.1109/99.660313

27. Y. Zhou, Y. Tan, Gpu-based parallel multi-objective particle swarm optimization, *Int. J. Artif. Intell.*, **7** (2011), 125–141.

28. B. Gonzalez-Sanchez, M. A. Vega-Rodríguez, S. Santander-Jiménez, Parallel multi-objective optimization approaches for protein encoding, *J. Supercomput.*, 5118–5148. https://doi.org/10.1007/s11227-021-04073-z

29. F. C. Holstege, E. G. Jennings, J. J. Wyrick, T. I. Lee, C. J. Hengartner, M. R. Green, et al., Dissecting the regulatory circuitry of a eukaryotic genome, *Cell*, **95** (1998), 717–728. https://doi.org/10.1016/S0092-8674(00)81641-4

30. Z. Jia, M. Maggioni, B. Staiger, D. P. Scarpazza, Dissecting the nvidia volta gpu architecture via microbenchmarking, preprint, arXiv:1804.06826.

31. T. U. Consortium, UniProt: The universal protein knowledgebase in 2023, *Nucleic Acids Res.*, **51** (2023), D523–D531. https://doi.org/10.1093/nar/gkac1052

32. J. X. Chin, B. K. S. Chung, D. Y. Lee, Codon optimization online (cool): A web-based multi-objective optimization platform for synthetic gene design, *Bioinformatics*, **30** (2014), 2210–2212. https://doi.org/10.1093/bioinformatics/btu192

33. J. C. Guimaraes, M. Rocha, A. P. Arkin, G. Cambray, D-tailor: Automated analysis and design of DNA sequences, *Bioinformatics*, **30** (2014), 1087–1094. https://doi.org/10.1093/bioinformatics/btt742

34. P. Puigbo, E. Guzmán, A. Romeu, S. Garcia-Vallve, Optimizer: A web server for optimizing the codon usage of DNA sequences, *Nucleic Acids Res.*, **35** (2007), W126–W131. https://doi.org/10.1093/nar/gkm219

35. B. Gonzalez-Sanchez, M. A. Vega-Rodríguez, S. Santander-Jiménez, A multi-objective butterfly optimization algorithm for protein encoding, *Appl. Soft Comput.*, **139** (2023), 110269. https://doi.org/10.1016/j.asoc.2023.110269

36. M. V. Díaz-Galián, M. A. Vega-Rodríguez, Many-objective approach based on problem-aware mutation operators for protein encoding, *Inf. Sci.*, **613** (2022), 376–400.

37. K. Deb, H. Jain, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints, *IEEE Trans. Evol. Comput.*, **18** (2013), 577–601. 10.1109/TEVC.2013.2281535

38. I. Das, J. E. Dennis, Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems, *SIAM J. Optim.*, **8** (1998), 631–657.

39. M. A. Dulebenets, An adaptive polyploid memetic algorithm for scheduling trucks at a cross-docking terminal, *Inf. Sci.*, **565** (2021), 390–421. https://doi.org/10.1016/j.ins.2021.02.039

40. J. Pasha, A. L. Nwodu, A. M. Fathollahi-Fard, G. Tian, Z. Li, H. Wang, et al., Exact and metaheuristic algorithms for the vehicle routing problem with a factory-in-a-box in multi-objective settings, *Adv. Eng. Inf.*, **52** (2022), 101623. https://doi.org/10.1016/j.aei.2022.101623

41. H. Gholizadeh, H. Fazlollahtabar, A. M. Fathollahi-Fard, M. A. Dulebenets, Preventive maintenance for the flexible flowshop scheduling under uncertainty: A waste-to-energy system, *Environ. Sci. Pollut. Res.*, 1–20. https://doi.org/10.1007/s11356-021-16234-x

42. M. A. Dulebenets, M. Kavoosi, O. Abioye, J. Pasha, A self-adaptive evolutionary algorithm for the berth scheduling problem: Towards efficient parameter control, *Algorithms*, **11** (2018), 100. https://doi.org/10.3390/a11070100

43. H. Zhao, C. Zhang, An online-learning-based evolutionary many-objective algorithm, *Inf. Sci.*, **509** (2020), 1–21. https://doi.org/10.1016/j.ins.2019.08.069

AIMS Press