




Parallel multi-objective optimization approaches for protein encoding

Belen Gonzalez-Sanchez¹ · Miguel A. Vega-Rodríguez¹  · Sergio Santander-Jiménez¹

Accepted: 4 September 2021 / Published online: 17 September 2021
© The Author(s) 2021

Abstract

One of the main challenges in synthetic biology lies in maximizing the expression levels of a protein by encoding it with multiple copies of the same gene. This task is often conducted under conflicting evaluation criteria, which motivates the formulation of protein encoding as a multi-objective optimization problem. Recent research reported significant results when adapting the artificial bee colony algorithm to address this problem. However, the length of proteins and the number of copies have a noticeable impact in the computational costs required to attain satisfying solutions. This work is aimed at proposing parallel bioinspired designs to tackle protein encoding in multiprocessor systems, considering different thread orchestration schemes to accelerate the optimization process while preserving the quality of results. Comparisons of solution quality with other approaches under three multi-objective quality metrics show that the proposed parallel method reaches significant quality in the encoded proteins. In addition, experimentation on six real-world proteins gives account of the benefits of applying asynchronous shared-memory schemes, attaining efficiencies of 92.11% in the most difficult stages of the algorithm and mean speed-ups of 33.28x on a 64-core server-grade system.

Keywords Parallel multi-objective optimization approach · Protein encoding · Synchronous and asynchronous parallelism · Design of multiple genes

✉ Miguel A. Vega-Rodríguez
mavega@unex.es

Belen Gonzalez-Sanchez
belengs@unex.es

Sergio Santander-Jiménez
sesaji@unex.es

¹ Escuela Politécnica, University of Extremadura, Campus Universitario s/n, 10003 Cáceres, Spain

1 Introduction

Maximizing the expression levels of a protein is one of the most critical tasks in synthetic biology. A commonly adopted approach to deal with this problem involves the integration of multiple genes encoding the same protein into a host organism. This strategy makes use of alternative protein-coding sequences (also known as CDS), which contain nucleotide triplets (codons) codifying every particular amino acid of the target protein. By integrating multiple genes into the host, the expression levels tend to increase proportionally to the number of integrated copies [25]. This behaviour is not verified in all the cases, as shown in [8], but appears in many cases. Therefore, this technique represents one of the main approaches to maximize the expression levels of a protein [6].

Encoding proteins through the integration of multiple genes is a challenging research topic from both biological and computational perspectives. In order to reduce cost and time burdens, many research works [6, 19, 24] address the problem by integrating the gene copies very close to each other within the host genome. This approach has a drawback in the fact that they can induce homologous recombination, a critical issue that implies the loss of some of the integrated copies when identical or very similar sub-sequences are used [2]. As an illustration, given six concatenated genes ($g_1, g_2, g_3, g_4, g_5, g_6$), an homologous recombination between g_1 and g_4 will motivate the loss of g_2 and g_3 , thus reducing the number of integrated copies to (g_1, g_4, g_5, g_6) and, consequently, the expected expression level.

As a result, each one of the CDSs involved in the protein encoding task must be as different as possible in order to avoid homologous recombination between CDSs or sub-sequences within the same CDS. Furthermore, the length of the repeated sub-sequences must be minimized. Several works in the literature discussed the minimum length that tends to induce homologous recombination, yet this value is strongly related to the characteristics of the host organism. For example, an experimental study in *Saccharomyces cerevisiae* established in 30 bp (base pairs) the sub-sequence length that increases the likelihood of homologous recombination [13]. On the other side, another research [10] reported that identical sub-sequences with 70 bp in length induce homologous recombination in *Bacillus subtilis*. Finally, [20] verified that this recombination occurs in sequences with 23 bp in the case of *Escherichia coli*. Despite these differences, all the studies agree on the same condition: the likelihood of inducing homologous recombination can be minimized by reducing the length of identical sub-sequences as much as possible. Therefore, the protein encoding task must be conducted according to these two initial optimization criteria: maximize the differences among CDSs and minimize the length of identical sub-sequences.

Designing different CDSs that encode the same protein is possible due to the fact that each amino acid can be encoded by using multiple synonymous codons. Each amino acid has between one to six different codifications, but some of them are better adapted to the target host organism than others. This implies that the codons with better adaptation properties will likely promote a higher expression level of the protein [1]. Therefore, the selection of accurate codon synonyms is

important to satisfactorily tackle the problem. This optimization criterion is commonly designated as codon adaptation index (CAI) and tries to define CDSs with the most adapted synonymous codons. Consequently, protein encoding can be tackled as a multi-objective optimization problem that deals with the codification of CDSs as different as possible among them but by using the best-adapted codons, being conflicting objectives. More precisely, these optimization goals can be formulated by using three objective functions: (1) CAI, (2) hamming distance between CDSs, and (3) length of repeated or common substrings.

Related works on protein encoding optimization are mostly focused on defining procedures based on single-objective formulations [23, 26, 27]. In this context, the use of CAI is common, within separated CDSs, as the objective employed to find solutions that result in higher expression levels. Some key biological methods commonly applied under this kind of formulations are COOL [3], OPTIMIZER [18], and D-Taylor [7]. However, the need to consider multiple optimization criteria simultaneously gave rise to the proposal of novel methods that apply multi-objective optimization strategies. Examples of multi-objective metaheuristics that have been applied in this context are the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [4], as in [22], and the multi-objective artificial bee colony algorithm (MOABC) devised in [5].

From a computational perspective, multi-objective protein encoding introduces new layers of complexity to the optimization process, as a result of the consideration of three quality criteria. On top of this, the lengths of the considered proteins and the number of copies to be encoded also have a significant impact in execution times. In fact, [22] reported an increase in runtime of order $O(t \log t)$, where t is the total length of the CDSs, which effectively imposes limitations in the characteristics of the CDSs that can be synthetically designed. Therefore, this problem demands the application of parallel computing techniques to achieve satisfying time-to-solution properties [21]. Multiple research works have pointed out the benefits of applying high-performance computing to bioinformatics problems, as in the case of epistasis detection [12], RNA sequence alignment [11], cellular model tuning [17], and others [15]. However, to the best of our knowledge, no previous works have properly addressed the parallelization of the multi-objective protein encoding problem.

This work is aimed at proposing parallel solutions, based on our MOABC approach, to efficiently tackle multi-objective protein encoding optimizations. The devised methods are oriented towards execution in shared-memory multiprocessor systems, which are widely spread and commonly adopted in bioinformatics research, using the OpenMP standard [16]. Taking into account the characteristics of the baseline MOABC code, two different orchestration schemes are defined to allow the cooperation of threads under synchronous and asynchronous execution models. Using the information provided by their time profiles, the proposed designs are thoroughly analysed under different parallel and multi-objective quality metrics, in order to identify the most efficient and effective strategies to tackle this problem. Comparisons with alternative methods are also presented to examine potential impacts in solution quality and validate the relevance of the proposal.

The main contributions of this work can be summarized as:

CDS_1	UCU	CUU	GUA	CCU	UAC	CGA
CDS_2	UCC	CUA	GUG	CCA	UAU	AGA
CDS_3	UCA	CUG	GUU	CCC	UAC	AGG
Amino acids sequence	S	L	V	P	Y	R

Fig. 1 An example of a solution with 3 CDSs for encoding a simplified protein with 6 amino acids. Each amino acid is coded with a codon (different synonymous codons exist for each amino acid), which is a triplet of three nucleotides (A, C, G or U). This example also illustrates the computation of *MLRCS*. CDS_1 contains *UACC* as a repeated substring within the same CDS (in red), but the *MLRCS* is found between the pair CDS_2 and CDS_3 , which have *UCCCUA* (in blue) as a common substring

- Identification of parallelism opportunities when using MOABC to conduct protein encoding tasks, examining which stages of the algorithm are suitable for parallelization in shared-memory multicore platforms.
- Proposal of parallel bioinspired designs for MOABC following two different parallelization approaches, based on the orchestration of OpenMP execution threads under synchronous and asynchronous schemes.
- Analysis and discussion of parallel performance (speedups and efficiencies) in a 64-core hardware infrastructure to identify the opportunities brought by the devised parallel approaches, using for experimentation purposes six problem instances belonging to real-world proteins.
- Analysis of the quality of the results reported by the proposal under three different multi-objective quality metrics, compared with other tools from the literature to examine the impact of parallelism in the outcome of the algorithm.

The rest of this paper is organized as follows: the next section provides insight into the formulation of the tackled problem and the objective functions under consideration. Section 3 details the alternative designs devised to parallelize the MOABC algorithm. Then, Sect. 4 summarizes the datasets and metrics used for evaluation purposes, while Sect. 5 discusses the experimental results obtained by the proposed approaches and the comparisons with other tools. Finally, Sect. 6 includes conclusions and defines future research directions.

2 Problem definition

In the problem addressed in this work, a solution is defined by a set of m CDSs encoding a single protein with m copies. Each CDS provides a set of codons that translate into the amino acids of the target protein, being all the CDSs of equal length. A solution is therefore codified by m strings of characters, where each character represents one of the four main RNA bases: A, C, G, U (Adenine, Cytosine, Guanine, and Uracil respectively). Figure 1 presents an example for a simplified protein.

With the aim of guiding the search towards optimal solutions, we have considered three objective functions to determine the fitness of each solution. The first one determines if the best-adapted codons have been used in each CDS, since these codons should be used preferably. The second and third objective functions try to avoid homologous recombination issues. For this purpose, the second objective examines the differences between pairs of CDSs using the Hamming distance, while the third objective evaluates repeated or common substrings found in the CDSs. The following subsections describe in detail each objective.

2.1 Codon adaptation index (CAI)

This objective function assesses the CDSs in a solution in accordance with their potential adaptation to the host organism. These adaptation values are calculated for each CDS based on the synonymous codons used for encoding the protein. Due to the fact that some synonymous codons are better adapted than others, the idea behind the CAI criterion is to use codons with a higher usage frequency. The calculation of CAI for a single CDS can be expressed as shown in Eq. 1:

$$CAI(CDS_i) = \sqrt[N]{\prod_{n=1}^N W(\text{codon}_{i,n})}, \quad (1)$$

where i refers to the i -th CDS in the evaluated solution, N the number of codons in a CDS, n the position of the codon within the sequence, and W the adaptation weight assigned to the n -th codon from the i -th CDS. This weight is set as the usage frequency of the selected synonymous codon over the usage frequency of the most common synonymous codon. In order to calculate these weights, we employed as a reference the codon usage frequencies from [22].

Since each solution comprises multiple CDSs (as defined by the expert), the CAI objective function must evaluate the minimum Codon Adaptation Index ($mCAI$) value found among the CDSs. Given a solution with I CDSs, the $mCAI$ score can be computed as expressed in Eq. 2:

$$mCAI = \min_{1 \leq i \leq I} CAI(CDS_i). \quad (2)$$

The main goal is to maximize the $mCAI$ value within a solution so that all CDSs try to reach high values. It is worth remarking that the use of the average CAI is not a representative measurement of adaptation in this context, since poor adaptation values can be hidden among all the other CAI values.

2.2 Hamming distance between CDSs (HD)

The second objective function is aimed at examining the similarities between pairs of CDSs. For this purpose, the Hamming distance (HD) is adopted to calculate the differences between nucleotides, at the same positions, from each pair of CDSs. HD can be calculated as shown in Eq. 3:

$$HD(CDS_i, CDS_j) = \sum_{1 \leq k \leq L} \sigma(CDS_{i,k}, CDS_{j,k}), \quad (3)$$

where CDS_i and CDS_j are the i -th and j -th CDSs in the evaluated solution, L the sequence length (which is equal for all CDSs), k the currently processed position in the sequence, and σ a function that measures if the compared nucleotides are equal ($\sigma = 0$) or not ($\sigma = 1$).

In order to generate CDSs as different as possible, the HD is calculated for all the paired combinations of CDSs within a solution. In this way, the objective function will give preference to the solution that maximizes the minimum value of HD (mHD) found among the comprised I CDSs, as expressed in Eq. 4:

$$mHD = \min_{1 \leq i < j \leq I} \frac{HD(CDS_i, CDS_j)}{L}. \quad (4)$$

Similarly to the CAI function, the use of average HD value is not a representative fitness measurement because it can hide pairs of CDSs with poor HD values. Therefore, the optimization goal is to maximize the mHD score.

2.3 Length of repeated or common substrings (LRCS)

The third objective function is focused on detecting repeated substrings that appear within the same CDS or common substrings that appear between pairs of CDSs. Given a CDS with a nucleotide substring S of length l , a common substring is found when another CDS in the solution contains exactly the same nucleotide substring S . In the case of finding a common substring between two CDSs, the starting positions p and q of the common substring in each CDS can be equal or different. Otherwise, if there is a repeated substring within a single CDS, the starting positions p and q will be different. The first step is to identify the length of repeated or common substrings (LRCS), as shown in Eq. 5:

$$LRCS(CDS_i, CDS_j) = \underset{1 \leq p, q, l \leq L}{\text{length}(S_{i,p,l})} \quad \text{when } (S_{i,p,l} = S_{j,q,l}). \quad (5)$$

In Eq. 5, if the two substrings $S_{i,p,l}$ and $S_{j,q,l}$ are found between the pair CDS_i and CDS_j , being $i \neq j$, the substring starting positions p and q can be any within each CDS. On the other side, if CDS_i and CDS_j are the same ($i = j$), the starting positions have to be different ($p \neq q$).

The final goal is to find the maximum length of repeated or common substring ($MLRCS$), among all the I CDSs, as expressed in Eq. 6:

$$MLRCS = \max_{1 \leq i \leq j \leq l} \frac{LRCS(CDS_i, CDS_j)}{L}. \quad (6)$$

Remember that every CDS has a length of L nucleotides. Figure 1 depicts the identification of $MLRCS$ in an example with three CDSs.

Under this objective function, the optimization is aimed at finding the solution that minimizes the $MLRCS$ score.

3 Parallel strategies for multi-objective protein encoding

This section is devoted to outline first the serial metaheuristic employed in this research. Afterwards, it proceeds with the description of the parallel designs herein devised to accelerate protein encoding in multiprocessor systems.

3.1 Baseline metaheuristic: MOABC

Finding highly adapted CDSs that encode the same protein, while avoiding homologous recombination, is a complex optimization task. Previous research showed the relevance of applying metaheuristics in this context. In particular, MOABC is a highly promising proposal that attained significant results for this problem in previous work [5]. Moreover, MOABC is based on ABC (artificial bee colony) metaheuristic, which has been used in many different problems since its development, achieving very good results [9]. Therefore, MOABC will be the baseline algorithm targeted in this work.

MOABC is a population-based metaheuristic based on swarm intelligence that undertakes complex optimization tasks by mimicking the behaviour of bee colonies. This algorithm is built upon an exploitation-exploration algorithmic scheme, involving three main search strategies, to evolve the population:

1. *Employed bees step* In this stage, new candidate solutions are generated by processing the neighbourhood of the solutions handled by the algorithm in the current generation.
2. *Onlooker bees step* This procedure carries out the exploitation of the fittest solutions found in the previous step, assigning a higher selection probability to the most promising solutions.
3. *Scout bees step* This stage is aimed at processing unexplored regions of the search space, by means of replacing stagnated solutions that have not been successfully improved in a *limit* number of generations.

Algorithm 1 MOABC pseudo-code.

Require: *colony_size* (number of solutions in the population), *max_cycles* (maximum number of generations), *limit* (stagnation limit), and P_m (mutation probability)

Ensure: *nondominated_file* (file with the non-dominated solutions)

- 1: *nondominated_file* $\leftarrow \emptyset$
- 2: **for** $i = 1$ to *colony_size* **do**
- 3: *population*[i] \leftarrow Initialize Solution
- 4: *population*[i].*counter* $\leftarrow 0$ /* Initialize its trial counter */
- 5: **end for**
- 6: **for** $cycle = 1$ to *max_cycles* **do**
- 7: /* 1) Employed bees step */
- 8: **for** $i = 1$ to *colony_size* **do**
- 9: *new_solution* \leftarrow Employed Bee Search (*population*[i], P_m)
- 10: *new_solution* \leftarrow Calculate Objective Functions (*new_solution*)
- 11: *population*[i] \leftarrow Pareto Comparison (*new_solution*, *population*[i])
- 12: **if** *population*[i] has not been improved **then**
- 13: *population*[i].*counter* \leftarrow *population*[i].*counter* + 1
- 14: **end if**
- 15: **end for**
- 16: *population* \leftarrow Sort by Rank-Crowding (*population*, *colony_size*)
- 17: *population* \leftarrow Calculate Selection Probabilities (*population*)
- 18: /* 2) Onlooker bees step */
- 19: **for** $i = colony_size + 1$ to $2 * colony_size$ **do**
- 20: *sel_solution* \leftarrow Select Solution (*population*)
- 21: *new_solution* \leftarrow Onlooker Bee Search (*sel_solution*, P_m)
- 22: *new_solution* \leftarrow Calculate Objective Functions (*new_solution*)
- 23: *population*[i] \leftarrow Pareto Comparison (*new_solution*, *sel_solution*)
- 24: **if** *sel_solution* has not been improved **then**
- 25: *population*[i].*counter* \leftarrow *population*[i].*counter* + 1
- 26: **end if**
- 27: **end for**
- 28: /* 3) Scout bees step */
- 29: **for** $i = 1$ to $2 * colony_size$ **do**
- 30: **if** *population*[i].*counter* > *limit* **then**
- 31: *population*[i] \leftarrow Scout Bee Search ($cycle$, P_m)
- 32: *population*[i] \leftarrow Calculate Objective Functions (*population*[i])
- 33: *population*[i].*counter* $\leftarrow 0$
- 34: **end if**
- 35: **end for**
- 36: *population* \leftarrow Sort by Rank-Crowding (*population*, $2 * colony_size$)
- 37: *nondominated_file* \leftarrow Update Non-dominated Solutions File (*nondominated_file*)
- 38: **end for**
- 39: **Return** *nondominated_file*

Algorithm 1 illustrates the main stages of MOABC. This metaheuristic requires the following input parameters: (1) the number of solutions in the population managed by the algorithm (*colony_size*), (2) the stop criterion, given by a maximum number of cycles or generations (*max_cycles*), (3) the maximum number of generations that a stagnated solution can remain in the population (*limit*), and (4) the mutation probability considered in the generation of new candidate solutions (P_m). As the outcome of a multi-objective metaheuristic is not a single solution but a set of Pareto solutions, MOABC additionally manages a file, designated as *nondominated_file*, to store the non-dominated solutions found throughout the execution of the algorithm.

In order to adapt MOABC to protein encoding, the codification of a solution will be given by a set of CDSs, that is, a set of equal-length character sequences. Considering this solution encoding, the algorithm initializes the population (lines 2–5 in Algorithm 1) by generating *colony_size* – 1 solutions randomly. The remaining initial solution is generated by selecting for each amino acid the codon with the highest adaptation, so its *mCAI* value is equal to 1. This strategy is employed to boost the optimization of solutions with high CAI values. Furthermore, for every solution, the trial counter (line 4) is also initialized. This counter is used to count the number of times that the corresponding solution has not been successfully improved.

Until the stop criterion is satisfied, the main loop of MOABC sequentially applies the three main search strategies to evolve the population. The first one, the employed bees step (lines 7–15), operates over the population and generates *colony_size* new neighbour solutions by applying mutations with a probability of P_m . More specifically, four mutation variants are implemented:

1. For each CDS, each codon is randomly replaced by a different synonymous codon.
2. For the CDS with the minimum CAI value, each codon is replaced by a synonymous codon with greater adaptation weight.
3. For the pair of CDSs with the minimum HD, each codon is randomly replaced by a synonymous codon.
4. The codons included in the longest length repeated or common substring (LRCS) are randomly replaced with synonymous codons.

Once applied the mutations, the new solutions are compared with the original ones by using Pareto dominance, in order to store the most satisfying ones. Pareto ranks and crowding distances are then calculated for the solutions kept after the employed bees step (line 16), sorting them according to their multi-objective quality (convergence and diversity) [4]. The results of this sorting procedure are used to calculate the selection probabilities (line 17) required in the onlooker bees step (lines 18–27). In this step, roulette-wheel selection is performed to choose the solutions to be exploited, generating *colony_size* additional solutions through the mutation operators previously defined. Lastly, the scout bees step (lines 28–35) checks the population for exhausted, stagnated solutions, that is, solutions that have not been successfully improved in a *limit* number of generations. The

identified exhausted solutions are replaced with randomly generated solutions, which are mutated n times (where n is proportional to the current generation cycle).

At the end of a generation, the $2 \times colony_size$ solutions in the population are sorted according to Pareto ranks and crowding distances (line 36). The best $colony_size$ solutions will be assigned to the employed bees in order to continue their tasks in the next generation. Furthermore, the non-dominated file is updated at each generation with the non-dominated solutions found by the algorithm (line 37), being reported as output when the stop criterion is satisfied.

As discussed in [5], the strengths of MOABC's optimization engine allow this algorithm to achieve good results in the task of designing proteins encoded with multiple CDSs. However, the main weakness of this approach lies in the high execution times required to deal with large protein instances. With the aim of addressing this problem, we propose in this research work different parallel designs to improve the execution time for the multi-objective protein encoding in shared-memory architectures with MOABC.

3.2 Synchronous parallel MOABC (SP-MOABC)

The first approach examined in this work follows a synchronous parallelization scheme. The main goal of this scheme lies in distributing the workload of the search loops that compose a generation of the metaheuristic, thus preserving the original algorithmic design. The search loops are suitable for parallelization in multicore machines, since the processing of a new candidate solution x is independent from the processing associated to any other new solution y .

Algorithm 2 shows the OpenMP implementation of the synchronous parallel design proposed for MOABC (explained in detail in the previous subsection, Algorithm 1), designated as SP-MOABC. The first step is the initialization of the file that will store the non-dominated solutions (line 1 in Algorithm 2). A new input parameter (the rest of parameters are the same as in Algorithm 1), *num_threads*, is added to define the number of execution threads that are considered for parallel processing, which is set by using the `omp_set_num_threads` routine. The parallel team comprising these execution threads will be initialized at the beginning of the algorithm via `#pragma omp parallel` (lines 2–4 in Algorithm 2). In this moment (line 3), some variables are declared as `private` in order to avoid data problems.

Algorithm 2 SP-MOABC: Synchronous Parallel MOABC pseudo-code.

Require: *colony_size* (number of solutions in the population), *max_cycles* (maximum number of generations), *limit* (stagnation limit), P_m (mutation probability), and *num_threads* (number of execution threads)

Ensure: *nondominated_file* (file with the non-dominated solutions)

```

1: nondominated_file  $\leftarrow \emptyset$ 
2: omp_set_num_threads(num_threads)
3: #pragma omp parallel private (new_solution, sel_solution, cycle)
4: {
5: #pragma omp for
6: for  $i = 1$  to colony_size do
7:   population[ $i$ ]  $\leftarrow$  Initialize Solution and Trial Counter
8: end for
9: for  $cycle = 1$  to max_cycles do
10:  #pragma omp for
11:  for  $i = 1$  to colony_size do
12:    new_solution  $\leftarrow$  Employed Bee Search (population[ $i$ ],  $P_m$ )
13:    population[ $i$ ]  $\leftarrow$  Compare and Update (new_solution, population[ $i$ ])
14:  end for
15:  #pragma omp single
16:  population  $\leftarrow$  Sort by Rank-Crowding (population, colony_size)
17:  population  $\leftarrow$  Calculate Selection Probabilities (population)
18:  #pragma omp for
19:  for  $i = colony\_size + 1$  to  $2 * colony\_size$  do
20:    sel_solution  $\leftarrow$  Select Solution (population)
21:    new_solution  $\leftarrow$  Onlooker Bee Search (sel_solution,  $P_m$ )
22:    population[ $i$ ]  $\leftarrow$  Compare and Update (new_solution, sel_solution)
23:  end for
24:  #pragma omp for
25:  for  $i = 1$  to  $2 * colony\_size$  do
26:    if population[ $i$ ].counter > limit then
27:      population[ $i$ ]  $\leftarrow$  Perform Scout Bee Processing (cycle,  $P_m$ )
28:    end if
29:  end for
30:  #pragma omp single
31:  population  $\leftarrow$  Sort by Rank-Crowding (population,  $2 * colony\_size$ )
32:  nondominated_file  $\leftarrow$  Update Non-dominated Solutions File
    (nondominated_file)
33: end for
34: }
35: Return nondominated_file

```

The main target of parallelization lies in the loops that implement the employed, onlooker, and scout bees steps, since their computations do not require any sharing of information among solutions. The initialization of the population also verifies this condition and consequently can be also conducted in parallel. Parallel loops are therefore defined for each step by using `#pragma omp for` worksharing directives to distribute independent chunks of iterations among the available threads, thus accelerating the process of defining new candidate solutions at each step of the algorithm (lines 5–8 for the initialization, lines 10–14 for the employed bees step, lines 18–23 for the onlooker bees step, and lines 24–29 for the scout bees step). The operations that perform the management, update, and sorting of globally shared structures (e.g. the non-dominated file) will be handled by a single execution thread defined through `#pragma omp single` directives (lines 15–17 and 30–32).

The synchronous behaviour of SP-MOABC is attained by using the implicit barriers located at the end of the `#pragma omp for` loops, which is a simple strategy to guarantee the correctness and accuracy shown by the original serial algorithm. The idea herein pursued consists in completing the evolution of the population in a generation (that is, an iteration of the main loop) before proceeding with the parallel calculations corresponding to the next generation.

3.3 Asynchronous parallel MOABC (AP-MOABC)

The second approach herein devised, AP-MOABC, applies an asynchronous execution model to parallelize MOABC. The main contribution of this approach lies in introducing bioinspired parallelization strategies to minimize synchronization and waiting times by mimicking the parallel behaviour of honey bees. In nature, a bee does not wait for the rest of the colony to continue its tasks, that is, they work in an asynchronous mode. In practical terms, this behaviour can be translated to the way execution threads undertake parallel tasks, avoiding implicit barriers at the end of the parallel loops to improve the exploitation of hardware resources in shared-memory parallel platforms.

AP-MOABC follows a master-worker parallelization model, where the execution threads can play two possible roles:

- Master thread: the master is responsible for updating and managing the population immediately as new candidate solutions arrive from the workers, performing rank+crowding sorting and the calculation of selection probabilities upon detec-

tion of new solutions. It also controls the end of the execution by verifying if the stop criterion is satisfied.

- Worker threads: while the master thread does not send a signal of termination, all the other available threads perform worker tasks involving the generation of new candidate solutions according to employed, onlooker, and scout bees strategies. Each thread is responsible for processing, in an asynchronous way, a chunk of solutions in the population and for communicating the obtained results to the master thread.

The main structure of AP-MOABC is shown in Algorithm 3, which applies OpenMP directives to specify and configure the parallel regions. It can be observed several changes with regard to previous pseudo-codes. First, the stop criterion is not given by a maximum number of cycles/generations, since the asynchronous model implicitly removes the idea of generation from the algorithmic design. That is, the population evolves immediately as new solutions are generated by a thread, without waiting for other threads. An equivalent stop criterion, given by a maximum number of evaluations (*max_eval*), must be therefore implemented in AP-MOABC. The master thread will control the stop criterion condition, updating the number of consumed evaluations when new solutions are communicated from the workers. These communications represent the second main change, as AP-MOABC includes a set of FIFO (First In, First Out) queues, designated as *solQueues*, to conduct worker-master interactions (line 2 in Algorithm 3). Each worker thread *i* has associated a queue *solQueues*[*i*], where the solutions generated by the worker are progressively stored (write operation at the tail of the queue). The master will iteratively check these queues to identify the arrival of new solutions (read operation at the head of the queue) and, upon detection, proceed with the update of the population. Therefore, *num_thread* - 1 queues are required to implement these interactions. This approach allows the workers to execute in an asynchronous way, independently from each other, without sharing data among them.

Algorithm 3 AP-MOABC: Asynchronous Parallel MOABC pseudo-code.

Require: *colony_size* (number of solutions in the population), *max_eval* (maximum number of evaluations), *limit* (stagnation limit), P_m (mutation probability), and *num_threads* (number of execution threads)

Ensure: *nondominated_file* (file with the non-dominated solutions)

```

1: nondominated_file  $\leftarrow \emptyset$ 
2: solQueues  $\leftarrow$  Initialize Solution Queues (num_threads, colony_size)
3: #omp_set_num_threads(num_threads)
4: #pragma omp parallel private(thread_id)
5: {
6: #pragma omp for
7: for  $i = 1$  to colony_size do
8:   population[ $i$ ]  $\leftarrow$  Initialize Solution and Trial Counter
9: end for
10: #pragma omp single
11:   population  $\leftarrow$  Sort by Rank-Crowding (population, colony_size)
12:   population  $\leftarrow$  Calculate Selection Probabilities (population)
13: /* Part of Initialize Solutions */
14: #pragma omp for
15: for  $i = 1$  to colony_size do
16:   population_swap[ $i$ ]  $\leftarrow$  population[ $i$ ]
17: end for
18: /* Initializations done. Performing asynchronous parallel execution */
19: thread_id  $\leftarrow$  omp_get_thread_num()
20: if thread_id == num_threads - 1 then
21:   Perform Master Tasks (population, population_swap, solQueues,
     colony_size, max_eval, num_threads)
22: else
23:   Perform Worker Tasks (population, solQueues[thread_id], colony_size,
     limit,  $P_m$ , nondominated_file, num_threads, thread_id)
24: end if
25: }
26: nondominated_file  $\leftarrow$  Update Non-dominated Solutions File
   (nondominated_file)
27: Return nondominated_file

```

At the initialization stage, the population in AP-MOABC is duplicated by means of the structure $population_{swap}$. The introduction of this structure is aimed at ensuring the integrity and consistency of the population data throughout the asynchronous execution. More specifically, the *population* structure represents the consistent copy of the population that is visible for the worker threads at a particular stage of the optimization process. On the other side, $population_{swap}$ is a copy exclusively handled by the master thread to update the data of the population when new solutions

are detected. It is over $population_{\text{swap}}$ where the master applies the procedures for sorting and calculating selection probabilities. Since $population_{\text{swap}}$ will then contain the updated population, the $population$ and $population_{\text{swap}}$ pointers must be swapped to make the current status of the population available to the workers.

Once both copies of the population have been initialized and sorted (with an initial calculation of selection probabilities, lines 6–17), the assignment of master-worker roles is performed to begin the asynchronous execution. The last thread in the parallel team, with thread identifier $thread_id = num_threads - 1$, will serve as the master instantiating the *perform master tasks* procedure (line 21), while the remaining threads will operate as workers through the *perform worker tasks* procedure (line 23). The obtaining of thread identifiers is performed by using the OpenMP `omp_get_thread_num` routine.

Algorithm 4 AP-MOABC: Master task pseudo-code.

```

1:  $eval \leftarrow 0$ 
2: while  $eval < max\_eval$  do
3:    $updatePopulation \leftarrow \text{false}$ 
4:   for  $i = 0$  to  $num\_threads - 2$  do
5:     while  $solQueues[i].empty() == \text{false} \ \&\& \ eval < max\_eval$  do
6:        $solution, pos \leftarrow solQueues[i].pop()$ 
7:        $population_{\text{swap}}[pos] \leftarrow solution$ 
8:        $updatePopulation \leftarrow \text{true}$ 
9:        $eval \leftarrow \text{Update Evaluations}(eval, solution)$ 
10:    end while
11:  end for
12:  if  $updatePopulation$  then
13:     $population_{\text{swap}} \leftarrow \text{Sort by Rank-Crowding}(population_{\text{swap}}, 2 * colony\_size)$ 
14:     $population_{\text{swap}} \leftarrow \text{Calculate Selection Probabilities}(population_{\text{swap}})$ 
15:     $population, population_{\text{swap}} \leftarrow \text{Swap Populations}(population_{\text{swap}}, population)$ 
16:  end if
17: end while
18:  $stop\_worker \leftarrow \text{true}$ 
19: Send Stop Signal to the Workers ( $stop\_worker, num\_threads - 1$ )

```

Algorithm 4 illustrates the master thread tasks. As previously introduced, the master thread is aimed at keeping updated the status of the optimization process, accordingly updating the population with the solutions available in the queues of the worker threads. While the stop criterion is not satisfied, the master iteratively checks each $solQueues[i]$ in order to detect the arrival of new solutions from the workers (lines 3–11 in Algorithm 4). When a new solution is available, it is extracted from the queue and stored in the master-only copy of the population,

$population_{swap}$. In this step, the master also updates the variable $eval$ used to keep track of the number of evaluations performed by the algorithm. Once all the queues have been checked, the master proceeds with the calculation of Pareto ranks and crowding distances over $population_{swap}$, sorting them and calculating afterwards selection probabilities (lines 13 and 14). As at this stage the $population$ structure is obsolete, the master exchanges the $population$ and $population_{swap}$ structure pointers in order to make available the updated population and selection probabilities to the worker threads (line 15). These steps are repeated until the stop criterion is verified (that is, $eval = max_eval$), finishing the master execution by sending stop signals to the workers (lines 18 and 19).

Algorithm 5 AP-MOABC: Worker task pseudo-code.

```

1:  $start, end \leftarrow$  Calculate Chunk ( $thread\_id, num\_threads, colony\_size$ )
2:  $pos \leftarrow start$ 
3: while  $stop\_worker == false$  do
4:   if  $thread\_id < num\_threads/2$  then
5:      $solution \leftarrow$  Perform Employed Bee Processing ( $pos, population, P_m$ )
6:   else
7:      $solution \leftarrow$  Perform Onlooker Bee Processing ( $pos, population, P_m$ )
8:   end if
9:   if  $solution.counter > limit$  then
10:     $solution \leftarrow$  Perform Scout Bee Processing ( $P_m$ )
11:   end if
12:    $solQueues[thread\_id].push(solution, pos)$ 
13:    $pos \leftarrow pos + 1$ 
14:   if  $pos == end$  then
15:      $pos \leftarrow start$ 
16:   end if
17: end while
18: while  $solQueues[id\_thread].empty() == false$  do
19:    $solution, pos \leftarrow solQueues[id\_thread].pop()$ 
20:   Update Non-dominated Solutions File ( $nondominated\_file, solution$ )
21: end while

```

The worker thread stages of AP-MOABC are shown in Algorithm 5. These threads will iteratively perform the generation of new candidate solutions until receiving the stop signal from the master thread. At the beginning, different chunks of the population are distributed among the worker threads (line 1 in Algorithm 5), so that each worker will operate over the assigned chunk of solutions. If the thread identifier lies within the first half of the number of worker threads, the worker will generate new candidate solutions by conducting employed bee searches (*Perform Employed Bee Processing*, line 5) over the currently processed position of the assigned chunk. Otherwise, the worker will operate following onlooker bee strategies, using the selection probabilities of the

consistent population structure to choose the most promising solutions (*Perform Onlooker Bee Processing*, line 7). The exploration tasks related to the scout bee searches are implemented in both types of workers, regardless of their basic role (employed or onlooker). That is, when a solution has been processed, its trial counter is checked in order to verify if the solution has surpassed the maximum trial limit. If so, the solution is replaced (*Perform Scout Bee Processing*, line 10). The candidate solution resulting from these searches is then introduced into the queue (line 12), proceeding afterwards with the processing of the next position of the chunk (lines 13–16). Figure 2 introduces a graphical comparison between the thread orchestration schemes in SP-MOABC and AP-MOABC.

4 Evaluation methodology

This section provides insight into the evaluation methodology followed to assess the designed parallel approaches for protein encoding. The characteristics of the employed datasets and the performance measurements used to evaluate parallel and multi-objective results are herein described.

4.1 Protein datasets

In this study, we use six representative real-world proteins in FASTA format from the UniProt¹ (Universal Protein Resource) database, which is a database very used in this field. A protein instance is defined according to two attributes: length (number of amino acids, AA) and number of CDSs. All the instances herein considered show a balance between these attributes since both of them contribute to the complexity of the problem. Instances with less CDSs include larger proteins, while instances with shorter proteins comprise a high number of CDSs. In this way, very different scenarios will be evaluated. These proteins are from different organisms (human, mouse, yeast, salmonella, etc.). Table 1 details the protein instances used in this work.

4.2 Parallel performance metrics

In order to evaluate the performance of each design and analyse the benefits of parallel processing in this context, two widely used parallel metrics are adopted: speedup and efficiency. First, the speedup metric (S_c) measures the improvement observed in execution time when a parallel algorithm is employed, in comparison with the serial version of that algorithm. More specifically, it calculates how many times the parallel execution (using c cores) is faster than the serial execution, as expressed in Eq. 7:

¹ <https://www.uniprot.org/>.

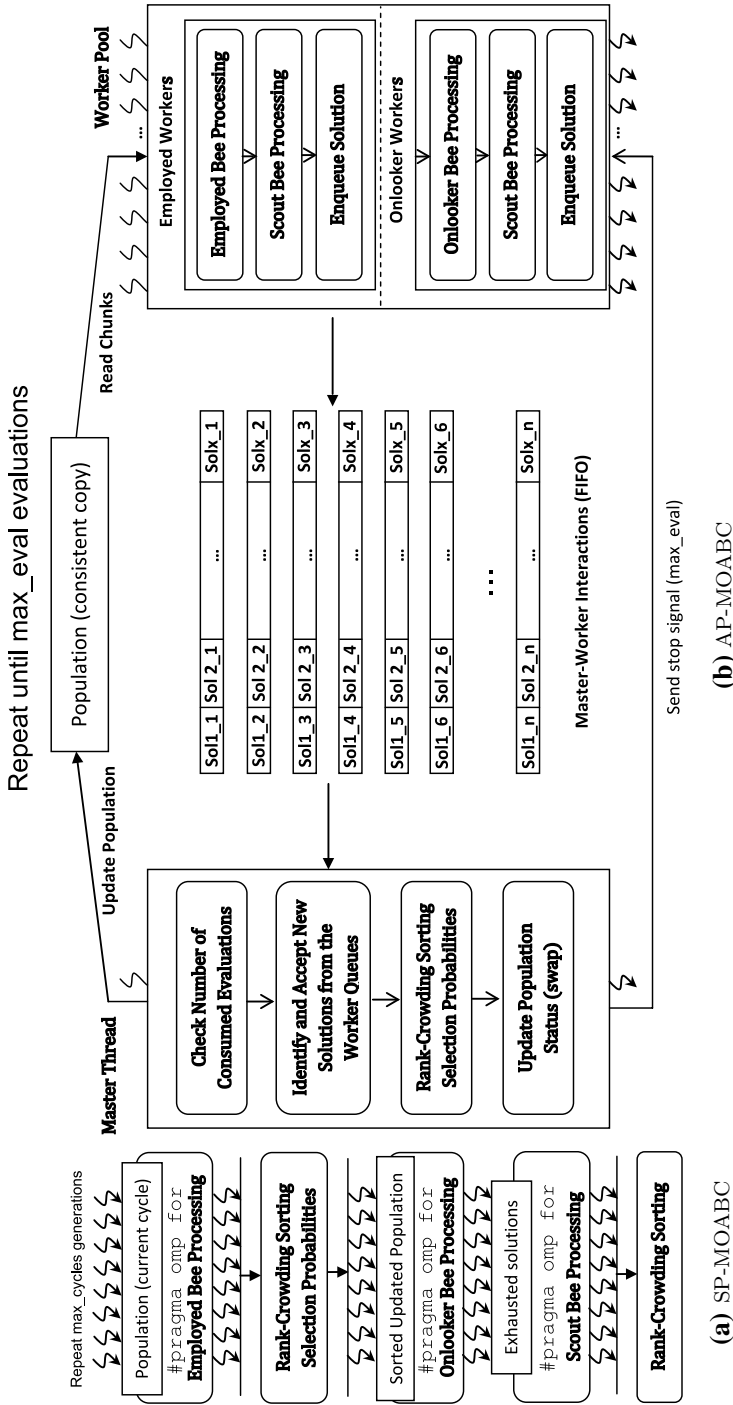


Fig. 2 Representation of parallel MOABC approaches for protein encoding

Table 1 Protein instances used in the experiments

Code	Name	CDSs	Length (AA)	CDSs*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

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

Objective	Nadir value	Ideal value
mCAI	0	1
mHD	0	0.40
MLRCS	1	0

$$S_c = \frac{ET_1}{ET_c}, \quad (7)$$

where ET_1 is the execution time of the serial algorithm and ET_c the execution time of the parallel version using c cores.

Second, the efficiency metric (E_c) examines how the designed parallel solution effectively takes advantage of the available parallel resources. It can be calculated in terms of the observed speedup over the number of cores (c) used for execution purposes, as indicated in Eq. 8:

$$E_c = \frac{S_c}{c} \times 100. \quad (8)$$

4.3 Solution quality metrics

In addition to the parallel metrics previously mentioned, insights into the quality of solutions will be given to examine if the application of parallel computing techniques has an impact over the output of the algorithm. The goal of the proposed parallel designs is to improve the execution time without changing the good results attained by the original serial algorithm (in terms of quality of solution). In order to determine this, three multi-objective quality metrics are considered: hypervolume, set coverage, and minimum distance to the ideal point.

The hypervolume (HV) indicator is one of the most commonly adopted metrics to assess the quality of the results in multi-objective optimization problems. This unary quality indicator calculates the volume of the objective space, delimited by the nadir and ideal points established in Table 2, which is covered by

the set of non-dominated solutions found by the evaluated algorithm. Equation 9 shows how hypervolume is calculated:

$$HV(\mathcal{A}, r) = Leb\left(\bigcup_{i=1}^{|\mathcal{A}|} h(a_i, r)\right), \quad (9)$$

where Leb denotes the Lebesgue measure, $|\mathcal{A}|$ the size of the set of non-dominated solutions \mathcal{A} , and $h(a_i, r)$ the volume covered by each solution $a_i \in \mathcal{A}$ taking r as the reference points.

The second multi-objective metric herein considered is the set coverage (SC). This binary quality indicator is based on the weak Pareto dominance concept. In the original Pareto dominance, we say that a solution b_j is dominated by another solution a_i iff a_i is not worse than b_j in all the considered objectives and is better at least in one of them. In the case of weak Pareto dominance, we say that a_i weakly dominates (\geq) b_j iff a_i is simply not worse than b_j in each objective. The set coverage therefore compares two solution sets \mathcal{A} and \mathcal{B} and calculates the percentage of solutions from \mathcal{B} that are weakly dominated by at least one solution from \mathcal{A} . The set coverage of \mathcal{A} over \mathcal{B} can be expressed as shown in Eq. 10:

$$SC(\mathcal{A}, \mathcal{B}) = \frac{|\{b_j \in \mathcal{B}; \exists a_i \in \mathcal{A} : a_i \geq b_j\}|}{|\mathcal{B}|}, \quad (10)$$

where $|\mathcal{B}|$ indicates the size of the solution set \mathcal{B} .

The third metric calculates the minimum distance to the ideal point for each set of solutions. The ideal point represents the most optimistic point in the objective space, a theoretically unreachable solution that optimizes all the considered objective functions simultaneously. However, it is not possible to reach it due to the conflicts among objectives. Even so, solutions as close as possible to the ideal values are the goal. Despite the outcome of a multi-objective optimization algorithm being a set of solutions, sometimes the expert can be interested in a single solution: the closest one to the ideal point. Under this metric, the best proposed approach will be the one containing the solution that minimizes the Euclidean distance to the ideal point.

5 Experimental evaluation and results

This section discusses the experimental results obtained by the proposed parallel approaches for multi-objective protein encoding. In a first step, the experimental settings and time profiling of MOABC are presented. Afterwards, the parallel results reported by the synchronous and asynchronous designs are comparatively evaluated. Finally, the assessment of solution quality with regard to state-of-the-art approaches is undertaken.

Table 3 List of parameter settings and configured values

Parameter	Description	Value
<i>colony_size</i>	Number of solutions in the population	128
<i>max_cycles</i>	Number of cycles/generations (SP-MOABC)	100
<i>max_eval</i>	Number of evaluations (AP-MOABC)	12800
<i>limit</i>	Attempts to improve a solution	10
P_m	Mutation probability	5%

5.1 Experimental settings

The hardware platform used to perform the experiments in this work is a multi-processor computing node composed of four 16-core AMD Opteron Abu Dhabi 6376 processors (a total of 64 physical cores, 2.3GHz) with 96GB DDR3 RAM, running Ubuntu 18.04LTS. The MOABC algorithm has been implemented in C/C++ language, using OpenMP for multithreading processing purposes. The C/C++ compiler version was GCC 8.4.0 with OpenMP 4.5. The experimental campaign involved 11 independent runs of the tested methods per experiment, with the aim of ensuring statistical reliability in the performance analysis.

Regarding the configuration of parameters of the MOABC algorithm, we took as a reference the parametric studies performed in previous research [5]. Table 3 shows the parameter settings that allow MOABC to maximize its optimization capabilities in the tackled problem and to achieve a better mapping to the characteristics of the targeted hardware platform. It is also worth remarking that the algorithmic design of AP-MOABC required a change in the stop criterion (from *max_cycles* to *max_eval*). The *max_eval* value was set to 12800 in order to make it equivalent to the original stop criterion (that is, $max_eval = colony_size \times max_cycles$).

5.2 MOABC time profiling

In order to better understand the benefits of applying parallel approaches, we first focus on analysing the time profile of the baseline MOABC, detailing the execution times of the serial version. Table 4 presents the median times in seconds together with the quartile deviation ($median_{\pm quartile\ deviation}$) observed for each stage in MOABC, as well as the total execution time per analysed protein instance. This table identifies each step of the algorithm according to the following notation:

- *initialize_solutions* refers to the initialization of solutions in the population (lines 1–5 in Algorithm 1).
- *employed_bee_processing* identifies the employed bee search step (lines 7–15 in Algorithm 1).
- *rank-crowding_sorting_1* refers to the sorting of employed bee solutions, according to Pareto ranks and crowding distances (line 16 in Algorithm 1).

Table 4 Execution time profile (in seconds, $median_{\pm quartile_deviation}$) for each stage of the serial version of MOABC

MOABC Stage	Q5VZP5	A4Y1B6	B3LS90
initialize_solutions	15.52 \pm 0.07	13.59 \pm 0.30	18.11 \pm 0.26
employed_bee_processing	1279.55 \pm 27.84	1147.53 \pm 7.87	1443.95 \pm 8.79
rank-crowding_sorting_1	47.12 \pm 1.63	64.11 \pm 0.78	66.97 \pm 0.48
selection_probabilities	0.002 \pm 0.00	0.002 \pm 0.00	0.002 \pm 0.00
onlooker_bee_processing	1281.74 \pm 26.70	1134.25 \pm 7.89	1426.35 \pm 4.94
scout_bee_processing	111.85 \pm 8.52	112.86 \pm 3.44	144.13 \pm 12.51
rank-crowding_sorting_2	190.52 \pm 9.61	253.82 \pm 3.23	261.09 \pm 2.68
update_nondominated_file	0.224 \pm 0.01	0.273 \pm 0.01	0.304 \pm 0.01
Total	2926.53	2726.44	3360.91
MOABC Stage	B4TWR7	Q91X51	Q89BP2
initialize_solutions	16.26 \pm 1.42	16.50 \pm 0.35	10.30 \pm 1.36
employed_bee_processing	1233.54 \pm 43.73	1288.94 \pm 6.73	1168.34 \pm 41.58
rank-crowding_sorting_1	64.41 \pm 2.41	69.24 \pm 0.80	61.15 \pm 2.98
selection_probabilities	0.002 \pm 0.00	0.002 \pm 0.00	0.002 \pm 0.00
onlooker_bee_processing	1246.42 \pm 45.50	1295.96 \pm 7.78	1168.24 \pm 45.16
scout_bee_processing	122.78 \pm 6.73	143.60 \pm 7.10	195.79 \pm 17.78
rank-crowding_sorting_2	251.97 \pm 9.17	265.49 \pm 2.77	243.54 \pm 1.17
update_nondominated_file	0.284 \pm 0.02	0.315 \pm 0.01	0.319 \pm 0.01
Total	2935.67	3080.05	2847.68

The ‘Total’ rows show the overall time per protein instance

- *selection_probabilities* corresponds to the calculation of selection probabilities prior to the onlooker bee step (line 17 in Algorithm 1).
- *onlooker_bee_processing* identifies the onlooker bee search step (lines 18–27 in Algorithm 1).
- *scout_bee_processing* refers to the scout bee search step (lines 28–35 in Algorithm 1).
- *rank-crowding_sorting_2* identifies the sorting of the overall solutions (employed + onlooker), according to Pareto ranks and crowding distances, for the next generation (line 36 in Algorithm 1).
- *update_nondominated_file* denotes the update of the file containing non-dominated solutions (line 37 in Algorithm 1).

Table 5 summarizes the percentage of the total time spent on each stage, averaged for all the protein instances under evaluation. Focusing on the stages involving global data management (with data dependencies), it can be observed that, on the one hand, *selection_probabilities* and *update_nondominated_file* do not have a significant impact in the execution time, showing time percentages ≤ 0.01 . On the other hand, the sorting operations *rank-crowding_sorting_1*

Table 5 Average time percentages from the baseline serial MOABC

MOABC stages	Average time (%)	Standard deviation (%)
<i>initialize_solutions</i>	0.50	0.001
<i>employed_bee_processing</i>	42.28	0.009
rank-crowding_sorting_1	2.09	0.002
selection_probabilities	0.00	0.000
<i>onlooker_bee_processing</i>	42.23	0.009
<i>scout_bee_processing</i>	4.67	0.010
rank-crowding_sorting_2	8.22	0.009
update_nondominated_file	0.01	0.000

The italic rows highlight the stages considered for parallelization purposes

and *rank-crowding_sorting_2* contribute with more noticeable time percentages. Nevertheless, the three key stages with inherent parallel processing opportunities, which are *employed_bee_processing*, *onlooker_bee_processing*, and *scout_bee_processing*, represent all together a percentage of almost 90% of the total execution time. This percentage illustrates the intrinsically parallel nature of MOABC and the potential benefits of applying parallel versions in the context of multi-objective protein encoding.

5.3 Parallel performance evaluation: SP-MOABC versus AP-MOABC

We now analyse the parallel performance results achieved by the synchronous and asynchronous strategies devised to parallelize MOABC (Sect. 3). Taking into account the specifications of the hardware platform, experimentation was performed considering configurations of 4, 8, 16, 32, and 64 execution threads. In this sense, it is worth remarking that the asynchronous approach in AP-MOABC requires at least three threads to work: a master thread, an employed worker thread, and an onlooker worker thread. In the case of the synchronous model SP-MOABC, an important issue to take into account is the scheduling policy applied to distribute iterations in the `#pragma omp for` loops among the available threads. By default, OpenMP applies a static scheduling that can imply load imbalance issues in this context. In order to ensure the execution of parallel loops according to the most efficient scheduling policy, we examined the different strategies available in the OpenMP standard and verified that the `guided` policy led to the most satisfying results. Consequently, the experiments with SP-MOABC employed this scheduling policy, progressively decreasing the chunk size, to attain a guided distribution of iterations while reducing the impact of the thread management overhead.

Figure 3 introduces the comparison between AP-MOABC and SP-MOABC, illustrating the evolution of speedups for each parallel model when the number of threads is increased (scalability study). The presented results are averaged

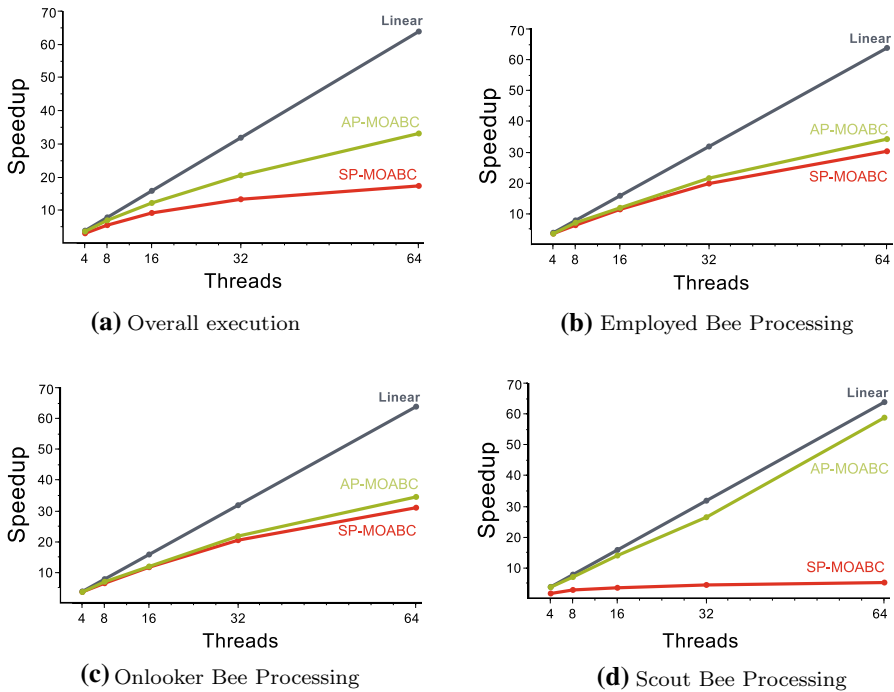


Fig. 3 Evaluation of AP-MOABC and SP-MOABC: evolution of speedups for the whole execution and for each search step separately. The presented results are averaged considering all the protein instances

considering all the protein instances and repetitions per experiment. Along with the overall speedups of the application, this figure also details the speedups observed in the three main search stages of the algorithm, namely the employed, onlooker, and scout bee processing. A more detailed comparison is presented in Table 6, following the parallel performance metrics introduced in Sect. 4.2 and considering execution scenarios from 4 to 64 threads. The upper side of Table 6 provides insight into the average speedups achieved in the experimentation, while the bottom side details the evaluation of efficiency values.

In overall terms, it can be observed that AP-MOABC attains better parallel performance than SP-MOABC in all the execution scenarios herein considered (from 4 to 64 cores). In fact, the differences between the two approaches under evaluation become more noticeable for larger system sizes. For 32 cores, AP-MOABC reports a speedup of 20.7x with regard to the baseline serial MOABC, which implies an efficiency around 65%. On the other side, SP-MOABC leads to a speedup of 13.5x and consequently an efficiency below the 50% threshold. Furthermore, the differences become more noticeable for 64 cores, where AP-MOABC is able to attain a speedup of 33.3x, in comparison with the 17.5x reported by SP-MOABC.

The analysis of each parallel stage of the algorithm gives account of the reasons that justify the performance gains achieved when asynchronous parallel designs are adopted in this problem. In the simplest step, the initialization of solutions

Table 6 Parallel evaluation of SP-MOABC and AP-MOABC, detailing the results observed in each stage as well as the overall results

	Average speedup	S_4	S_8	S_{16}	S_{32}	S_{64}
SP-MOABC	initialize_solutions	3.40	6.52	10.19	8.57	2.92
	employed_bee_processing	3.72	6.45	11.58	19.97	30.45
	onlooker_bee_processing	3.86	6.67	11.87	20.65	31.19
	scout_bee_processing	1.87	3.02	3.72	4.64	5.42
	Overall	3.21	5.67	9.34	13.46	17.49
AP-MOABC	initialize_solutions	3.64	6.80	11.01	12.36	5.06
	employed_bee_processing	3.69	7.24	12.08	21.72	34.40
	onlooker_bee_processing	3.82	7.14	12.10	21.98	34.69
	scout_bee_processing	3.92	7.21	14.19	26.63	58.95
	Overall	3.77	7.12	12.35	20.67	33.28
	<i>Average efficiency</i>	$E_4(\%)$	$E_8(\%)$	$E_{16}(\%)$	$E_{32}(\%)$	$E_{64}(\%)$
SP-MOABC	initialize_solutions	85.05	81.55	63.71	26.79	4.56
	employed_bee_processing	99.07	80.59	72.37	62.40	47.57
	onlooker_bee_processing	96.49	83.34	74.22	64.53	48.73
	scout_bee_processing	46.71	37.80	23.24	14.49	8.46
	Overall	80.25	70.88	58.38	42.06	27.33
AP-MOABC	initialize_solutions	90.96	86.27	68.81	38.63	7.91
	employed_bee_processing	92.18	90.52	75.53	67.88	53.75
	onlooker_bee_processing	95.61	89.19	75.60	68.69	54.20
	scout_bee_processing	97.94	90.13	88.72	83.21	92.11
	Overall	94.25	89.00	77.19	64.59	52.00

The columns S_c and E_c refer to the speedups and efficiencies observed when using c cores. The presented results are averaged considering all the protein instances

initialize_solutions, both approaches have a similar behaviour in terms of scalability. This is mostly motivated by the low demanding workload of the initialization loop, which contributes only a 0.5% of the execution time of the metaheuristic (as introduced in Table 5). The results for the *employed_bee_processing* and *onlooker_bee_processing*, shown in Fig. 3b and c, denote that these stages benefit more from the use of asynchronous strategies when a larger number of cores is employed, as the execution threads in AP-MOABC do not have any synchronization dependence imposed by the idea of generation in evolutionary algorithms. Finally, the *scout_bee_processing* step represents the stage with the most noticeable differences in performance between AP-MOABC and SP-MOABC. On the one hand, SP-MOABC barely reaches speedups of only 1.9x (4 cores), 3.0x (8 cores), 3.7x (16 cores), 4.6x (32 cores), and 5.4x (64 cores). On the other hand, the adoption of asynchronous strategies represents a satisfying solution to parallelize this step, with efficiencies of 92% when all the available parallel resources are used.

The scout bee step in MOABC is characterized by the fact that the number of solutions to be processed strongly depends on the degree of stagnation of the

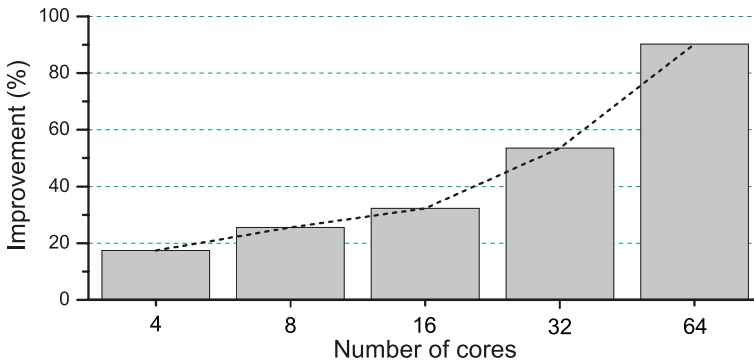


Fig. 4 Performance gains (%) obtained by AP-MOABC over SP-MOABC. It can be observed how AP-MOABC leads to significant improvements on system sizes involving a larger number of cores

population. That is, the number of parallel tasks to be processed is governed by the number of exhausted solutions detected by the *limit* condition. As a result, the number of scout bee searches to be conducted varies and can range from 0 (no exhausted solutions detected) to $2 * colony_size$ (the whole population is exhausted). When the number of scout searches to be performed is less than the number of execution threads, a synchronous approach leads to a poor utilization of hardware resources due to the presence of idle threads waiting for the termination of the parallel loop. In the worst case scenario (only one stagnated solution), the scout bee search contributes to the serial percentage of the algorithm, thus negatively affecting the achievable parallelism in accordance with Amdahl's law. By adopting the asynchronous approach in AP-MOABC, the threads are able to continue their execution when another thread is processing a scout bee search, consequently minimizing the impact in parallel performance introduced by the variable nature of this step.

In order to better depict the benefits of AP-MOABC with regard to SP-MOABC, Fig. 4 graphically represents the performance improvements attained in our experimentation for 4, 8, 16, 32, and 64 cores. It can be observed how the improvements progressively increase from 4 to 16 cores (17.5% to 32.2%). When larger system sizes are considered, the performance gains verified by AP-MOABC become even more noticeable (53.6% for 32 cores), reaching peak gains when all the resources in the parallel platform are employed (90.3% for 64 cores). Therefore, it can be concluded that the design of asynchronous strategies represents a satisfying solution to parallelize multi-objective protein encoding tasks on multicore, multiprocessor architectures.

5.4 Comparisons with other authors' proposals

After examining parallel performance, the next step in this study involves the assessment of the quality of solutions for AP-MOABC, in order to determine if the adoption of asynchronous strategies is able to preserve the search capabilities of the serial baseline MOABC [5]. For this purpose, we compare the results obtained by AP-MOABC with other methods from the literature, namely COOL [3] and Terai's

Table 7 Hypervolume results
(median_{±quartile_deviation})

Protein	AP-MOABC	COOL[3]	Terai's method[22]
Q5VZP5	59.27% _{±0.004%}	0.21% _{±0.000%}	59.92% _{±0.180%}
A4Y1B6	52.71% _{±0.001%}	0.31% _{±0.000%}	52.53% _{±0.060%}
B3LS90	55.59% _{±0.001%}	0.28% _{±0.000%}	54.62% _{±0.150%}
B4TWR7	49.79% _{±0.001%}	0.36% _{±0.000%}	48.91% _{±0.210%}
Q91X51	52.02% _{±0.001%}	0.18% _{±0.000%}	50.47% _{±0.230%}
Q89BP2	50.09% _{±0.001%}	0.17% _{±0.000%}	48.61% _{±0.220%}
Average	53.25%	0.25%	52.51%

The best values are in bold for each instance. The last row shows the average of the six protein instances

Table 8 Set Coverage results

Protein	SC(COOL[3], AP-MOABC)	SC(AP-MOABC, COOL[3])	SC(Terai[22], AP-MOABC)	SC(AP-MOABC, Terai[22])
Q5VZP5	0.00%	100.00%	64.04%	95.00%
A4Y1B6	0.00%	100.00%	62.75%	84.00%
B3LS90	0.00%	100.00%	43.92%	88.00%
B4TWR7	0.00%	100.00%	62.69%	74.00%
Q91X51	0.00%	100.00%	56.90%	93.00%
Q89BP2	0.00%	100.00%	49.37%	82.00%
Average	0.00%	100.00%	56.61%	86.00%

The best values are in bold, and the last row represents the average of the six protein instances

approach [22]. COOL is a biological tool that undertakes protein encoding optimizations by putting priority on the CAI objective function, while Terai's approach is a multi-objective evolutionary algorithm inspired by NSGA-II that optimizes the same three objective functions considered in this work.

These state-of-the-art methods are not focused on the parallelization of protein encoding. Therefore, we will employ these tools as a reference to evaluate the optimization capabilities (solution quality) of the proposed AP-MOABC. Nevertheless, it is worth remarking that, in the worst case scenario, our proposal is able to finish protein encoding tasks with execution time around 100 seconds, thus representing a significant step further over other methods that imply runtimes from many minutes to more than one hour. To the best of our knowledge, AP-MOABC represents the first attempt to parallelize the protein encoding task, and more specifically, a multi-objective search engine aimed at maximizing protein expression with multiple CDSs.

In order to perform these comparisons, we consider three different quality metrics: hypervolume, set coverage, and the minimum distance to the ideal point (detailed in Sect. 4.3). For the sake of fairness in the comparisons, all the methods were configured using the same parameter settings. Table 7 reports the

Table 9 Minimum distances to the ideal point/solution

Protein	AP-MOABC	COOL[3]	Terai's method[22]
Q5VZP5	0.489408	1.028865	0.503676
A4Y1B6	0.542613	1.050567	0.551986
B3LS90	0.512751	1.081525	0.512885
B4TWR7	0.563227	1.093655	0.574876
Q91X51	0.574168	1.166339	0.589626
Q89BP2	0.565618	1.121477	0.569445
Average	0.541298	1.090405	0.550416

The best values for each instance are in bold, and the last row provides the average for the six instances

median values for the hypervolume indicator together with the quartile deviation ($median_{\pm quartile_deviation}$), while Table 8 shows the set coverage results. For the set coverage, we consider the percentage of solutions from AP-MOABC covered by the other methods, SC(COOL,AP-MOABC) and SC(Terai,AP-MOABC), and the percentages covered by AP-MOABC over the other tools, SC(AP-MOABC,COOL) and SC(AP-MOABC, Terai). For both hypervolume and set coverage, higher scores denote better results. Finally, Table 9 shows the minimum distance from, at least, one solution of the median Pareto front to the ideal or utopian solution. Lower distances imply a better approximation to the ideal solution.

On examining the scores in these tables, it can be observed that AP-MOABC provides more satisfying results than the ones reported by the reference works. This is especially noticeable in the case of the set coverage metric (Table 8), which indicates that our approach is able to dominate or cover a significant percentage of solutions from the other tools (in average, 100% over COOL and 86% over Terai's method), while the other tools do not cover many of the solutions generated by our approach. AP-MOABC also reports the best hypervolume scores (Table 7) in almost all the protein instances, along with better approximations (minimum distances, Table 9) to the ideal solution in all the cases. These successful comparisons confirm the relevance of MOABC, and more specifically, of AP-MOABC in the multi-objective protein encoding problem.

6 Conclusions and future work

This work proposed parallel multi-objective approaches to undertake protein encoding optimization tasks, boosting expression levels through the integration of multiple CDSs. This problem was tackled according to a multi-objective formulation targeted at minimizing the effect of homologous recombination while maximizing codon adaptation levels. The length of the proteins and the number of copies to be encoded represent two key complexity factors, which turn this problem into a computationally demanding task in real-world scenarios. Two parallel designs, based on the MOABC algorithm, were devised to provide an efficient, accurate solver for protein encoding on shared-memory multicore platforms. The

first approach herein proposed, SP-MOABC, follows a synchronous generational scheme to orchestrate execution threads. The second approach, designated as AP-MOABC, adopts asynchronous strategies, combined with a master-worker design, to improve the utilization of hardware resources and minimize idle times in the most difficult stages of the algorithm.

Six real-world protein instances were used to evaluate the performance of the proposed parallel strategies in a 64-core multiprocessor machine. This experimental evaluation was performed attending to two different perspectives: (1) parallel performance (speedup and efficiency) and (2) quality of solutions (hypervolume, set coverage, and minimum distance to the ideal point). The profiling of MOABC revealed the impact of the three main search steps of the baseline algorithm (employed, onlooker, and scout bee searches), accounting for almost 90% of the execution time. Taking into account this information, the evaluation of parallel performance showed the bottleneck introduced by the scout bee step and how AP-MOABC was able to successfully deal with it, obtaining efficiencies of 92% in this difficult stage for 64 cores (along with additional improvements in the rest of steps). As a result, performance gains of 90% over SP-MOABC were attained by AP-MOABC when using the whole hardware infrastructure. Furthermore, the comparison of solution quality with other methods from the literature (the biological tool COOL [3] and the Terai's multi-objective approach [22]) indicated that AP-MOABC was able to achieve very good results from the quality viewpoint, dominating or covering 100% of the COOL solutions and 86% of the Terai solutions, also offering better approximations to the ideal solution in all the cases.

Two main directions of future research work can be established. Due to the parallel potential exhibited by AP-MOABC, the adaptation of this scheme to other hardware platforms beyond shared-memory scenarios will be investigated. In this work, multicore systems were considered because they are widely spread and commonly adopted in bioinformatics research, but as the problem addressed does not have significant memory requirements, other hardware platforms (such as GPUs) will be investigated. Emphasis will be put to the orchestration of heterogeneous resources in distributed-memory environments, studying strategies to balance performance and power consumption. The second direction will deal with the inclusion of other optimality goals for protein encoding. Particularly, the GC content measures the stability of sequences and has significantly contributed to the enhancement of protein expression levels in real-world studies [14]. Furthermore, the number of multi-objective metaheuristic methods applied to solve the protein encoding is still very reduced. For this reason, the development of other new multi-objective metaheuristic methods for this problem and their parallelization is also a topic of future research. This will imply their design for this specific problem, their implementation, their parallelization, their execution, and finally, their comparison.

Acknowledgements This work was partially funded by the MCIU (Ministry of Science, Innovation and Universities, Spain), the AEI (State Research Agency, Spain), and the ERDF (European Regional Development Fund, EU), under the contract PID2019-107299GB-I00/AEI/10.13039/501100011033

(Multi-HPC-Bio project). This research was also partially funded by the Government of Extremadura (Spain) and the ERDF (European Regional Development Fund, EU) under the project IB16002.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Declaration

Conflict of Interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Athey J, Alexaki A, Osipova E, Rostovtsev A, Santana-Quintero LV, Katneni U, Simonyan V, Kimchi-Sarfaty C (2017) A new and updated resource for codon usage tables. *BMC Bioinform* 18(1):1–10 (Article number: 391). <https://doi.org/10.1186/s12859-017-1793-7>
2. Aw R, Polizzi KM (2013) Can too many copies spoil the broth? *Microbial Cell Fact* 12(1):1–9 (Article number: 128). <https://doi.org/10.1186/1475-2859-12-128>
3. Chin JX, Chung BKS, Lee DY (2014) Codon optimization OnLine (COOL): a web-based multi-objective optimization platform for synthetic gene design. *Bioinformatics* 30(15):2210–2212. <https://doi.org/10.1093/bioinformatics/btu192>
4. Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans Evol Comput* 6(2):182–197. <https://doi.org/10.1109/4235.996017>
5. Gonzalez-Sanchez B, Vega-Rodríguez MA, Santander-Jiménez S, Granada-Criado JM (2019) Multi-objective artificial bee colony for designing multiple genes encoding the same protein. *Appl Soft Comput* 74:90–98. <https://doi.org/10.1016/j.asoc.2018.10.023>
6. Gu P, Yang F, Su T, Wang Q, Liang Q, Qi Q (2015) A rapid and reliable strategy for chromosomal integration of gene(s) with multiple copies. *Sci Rep* 5:1–9 (Article number 9684). <https://doi.org/10.1038/srep09684>
7. Guimaraes JC, Rocha M, Arkin AP, Cambray G (2014) D-Tailor: automated analysis and design of DNA sequences. *Bioinformatics* 30(8):1087–1094. <https://doi.org/10.1093/bioinformatics/btu742>
8. Hohenblum H, Gasser B, Maurer M, Borth N, Mattanovich D (2004) Effects of gene dosage, promoters, and substrates on unfolded protein stress of recombinant *Pichia pastoris*. *Biotechnol Bioeng* 85(4):367–375. <https://doi.org/10.1002/bit.10904>
9. Karaboga D, Gorkemli B, Ozturk C, Karaboga N (2014) A comprehensive survey: artificial bee colony (ABC) algorithm and applications. *Artif Intell Rev* 42:21–57. <https://doi.org/10.1007/s10462-012-9328-0>
10. Khasanov FK, Zvingila DJ, Zainullin AA, Prozorov AA, Bashkirov VI (1992) Homologous recombination between plasmid and chromosomal DNA in *Bacillus subtilis* requires approximately 70 bp of homology. *Mol Gen Genet* 234(3):494–497. <https://doi.org/10.1007/BF00538711>
11. Lalwani S, Sharma H (2019) Multi-objective three level parallel PSO algorithm for structural alignment of complex RNA sequences. *Evolut Intell pp* 1–9, <https://doi.org/10.1007/s12065-018-00198-y>
12. Li X (2017) A fast and exhaustive method for heterogeneity and epistasis analysis based on multi-objective optimization. *Bioinformatics* 33(18):2829–2836. <https://doi.org/10.1093/bioinformatics/btx339>

13. Manivasakam P, Weber SC, McElver J, Schiestl RH (1995) Micro-homology mediated PCR targeting in *Saccharomyces cerevisiae*. *Nucleic Acids Res* 23(14):2799–2800. <https://doi.org/10.1093/nar/23.14.2799>
14. Newman ZR, Young JM, Ingolia NT, Barton GM (2016) Differences in codon bias and GC content contribute to the balanced expression of TLR7 and TLR9. *Proceedings of the National Academy of Sciences* 113(10):E1362–E1371. <https://doi.org/10.1073/pnas.1518976113>
15. Ocaña K, Oliveira D (2015) Parallel computing in genomic research: advances and applications. *Adv Appl Bioinform Chem* 8:23–35. <https://doi.org/10.2147/AABC.S64482>
16. van der Pas R, Stotzer E, Terboven C (2017) *Using OpenMP - The Next Step*. The MIT Press, Cambridge
17. Pouranbarani E, Weber dos Santos R, Nygren A (2019) A robust multi-objective optimization framework to capture both cellular and intercellular properties in cardiac cellular model tuning: analyzing different regions of membrane resistance profile in parameter fitting. *PLoS ONE* 14(11):1–19. <https://doi.org/10.1371/journal.pone.0225245>
18. Puigbò P, Guzmán E, Romeu A, Garcia-Vallvé S (2007) OPTIMIZER: a web server for optimizing the codon usage of DNA sequences. *Nucleic Acids Res* 35(suppl 2):W126–W131. <https://doi.org/10.1093/nar/gkm219>
19. Scorer CA, Clare JJ, McCombie WR, Romanos MA, Sreekrishna K (1994) Rapid selection using G418 of high copy number transformants of *Pichia pastoris* for high-level foreign gene expression. *Bio Technol* 12:181–184. <https://doi.org/10.1038/nbt0294-181>
20. Shen P, Huang HV (1986) Homologous recombination in *Escherichia coli*: dependence on substrate length and homology. *Genetics* 112(3):441–457
21. Talbi EG (2015) Parallel Evolutionary Combinatorial Optimization. In: *Springer handbook of computational intelligence*, Springer, pp 1107–1125. https://doi.org/10.1007/978-3-662-43505-2_55
22. Terai G, Kamegai S, Taneda A, Asai K (2017) Evolutionary design of multiple genes encoding the same protein. *Bioinformatics* 33(11):1613–1620. <https://doi.org/10.1093/bioinformatics/btx030>
23. Tran TA, Vo NT, Nguyen HD, Pham BT (2015) A novel method to predict highly expressed genes based on radius clustering and relative synonymous codon usage. *J Comput Biol* 22(12):1086–1096. <https://doi.org/10.1089/cmb.2015.0121>
24. Tyo KEJ, Ajikumar PK, Stephanopoulos G (2009) Stabilized gene duplication enables long-term selection-free heterologous pathway expression. *Nat Biotechnol* 27:760–765. <https://doi.org/10.1038/nbt.1555>
25. Vassileva A, Chugh DA, Swaminathan S, Khanna N (2001) Expression of hepatitis B surface antigen in the methylotrophic yeast *Pichia pastoris* using the GAP promoter. *J Biotechnol* 88(1):21–35. [https://doi.org/10.1016/S0168-1656\(01\)00254-1](https://doi.org/10.1016/S0168-1656(01)00254-1)
26. Webster GR, Teh AYH, Ma JKC (2017) Synthetic gene design: the rationale for codon optimization and implications for molecular pharming in plants. *Biotechnol Bioeng* 114(3):492–502. <https://doi.org/10.1002/bit.26183>
27. Yu CH, Dang Y, Zhou Z, Wu C, Zhao F, Sachs MS, Liu Y (2015) Codon usage influences the local rate of translation elongation to regulate co-translational protein folding. *Mol Cell* 59(5):744–754. <https://doi.org/10.1016/j.molcel.2015.07.018>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.