Parallel Moth-Flame Optimization Algorithm

Matteo Minardi (238789), Mattia Nardon (233707)
High-Performance Computing for Data Science, University of Trento, Trento, 2024

***Abstract*—This report investigates the Moth Flame Optimization (MFO) algorithm's effectiveness in optimizing a predetermined benchmark function. The study explores strategies to enhance its performance through parallelization using MPI (Message Passing Interface) and OpenMP (Open Multi-Processing).**
Initially outlining MFO's significance in optimization tasks, the report focuses on its possible increases in performance with the different configurations that have been tried. The study first implements MPI, distributing computational tasks among multiple processes to expedite optimization, and, subsequently, OpenMP, a shared-memory multi-threading technique.
Evaluation against the selected benchmark function compares the serial, MPI-only and two MPI-OpenMP hybrid approaches. Metrics such as execution time and scalability are analyzed.
In summary, this study presents a concise assessment of MFO's workflow, importance, and performance improvements using MPI and OpenMP. The evaluation against the selected benchmark function demonstrates the potential of parallel computing to enhance MFO's scalability and efficiency in solving complex optimization challenges.

I. Introduction

THIS report investigates the workflow, utility, and potential enhancements of the Moth Flame Optimization (MFO) algorithm.
The MFO algorithm emulates moths' attraction to light sources (flames) to solve optimization problems. It iteratively updates potential solutions (represented as moths) based on the brightness (fitness) of nearby flames. This nature-inspired approach balances exploration and exploitation in search spaces and can handle diverse optimization problems. Its simplicity and adaptability make it a compelling choice for finding optimal or near-optimal solutions across various search spaces.
The study proceeds to implement parallelization using MPI, an established library for distributed computing. By distributing the computational workload across multiple processes, we aim to exploit the power of parallel computing to improve the optimization process. The report shows the modifications made to the original MFO algorithm to allow for MPI and discusses the results obtained from this parallel implementation.
Furthermore, the study is extended by exploring the integration of OpenMP, an ulterior parallelization technique, into the MFO algorithm. By leveraging shared-memory multiprocessing offered by OpenMP, we hope to enhance the algorithm's performance further. The report outlines the adaptations required for incorporating OpenMP into the MFO algorithm and provides an analysis of the outcomes achieved through two hybrid approaches.
For each configuration: serial, MPI-only and the two MPI-OpenMP hybridizations, the report presents and compares the obtained results. The analysis shows metrics such as execution time and scalability. Our deductions are provided for the observed outcomes of the efficiency and the effectiveness of the different parallelization strategies employed.

II. State of the Art Analysis

Currently, there's minimal ongoing research directly on the Moth Flame Optimization algorithm. Most studies focus on enhancing its performance by combining it with other optimization methods, such as the *Butterfly Optimization Algorithm*, creating the *h-MFOBOA* [1]. However, a small fraction of research aims to refine the algorithm itself, but more on the quality side of the solutions rather than about the performance. For example, one proposed improvement involves adding stochastic migration for the least successful moths early in execution, intending to guide them toward better solutions within the search space, called *Migration-Based Moth-Flame Optimization Algorithm* [2]. Another one is called *MFO-SFR* [3], that uses finding-and-replacing strategies to improve the fitness of the worse moths.
Our own implementations do not include these specific features as they are mostly used in cases where the number of flames is decreased as the iterations go by. We did introduce this possibility in our code, but for the benchmarks we kept the number of flames fixed throughout the whole executions, so they were not going to be as effective and were not going to affect the execution times significantly, either.

III. Inspirations

Since there is no official implementation of the MFO algorithm, and we could not find an available one written in C, we designed the code ourselves.
The workflow was defined taking into account different sources, all different from each other in some minute details, with the main one being [5].
The benchmarking functions on the other hand were already available and were therefore taken from the following GitHub repository [4], adjusting them slightly to fit our design.

IV. Parameters

The parameters specified in the code play crucial roles in the optimization towards effective convergence and solution discovery within the defined problem space. The two with the most impact on the execution time are: *NUM_DIMENSIONS*, that determines the dimensionality of the search space, influencing the complexity and size of the solution space, and *POPULATION_SIZE*, which defines the number of moths or potential solutions within this space, directly impacting the algorithm's exploration and exploitation capabilities.
Giving an explanation for the other parameters:

- *FIX_N_FLAMES* determines if the number of flames stays constant.
- *LOWER_BOUND* and *UPPER_BOUND* set domain boundaries.
- *MAX_ITERATIONS* limits the algorithm's runtime.
- *BETA_INIT* and *ALPHA* guide moth movements.
- *EARLY_STOPPING* and *EARLY_STOPPING_PATIENCE* manage premature termination conditions.

These parameters collectively direct the algorithm's behavior, optimizing solution search and convergence within the defined problem space. Not all of them were activated when performing the benchmarks, for example the number of flames was always fixed but the early stopping was not exploited.

V. Sequential solution

Each step in the algorithm, from initialization to finalization, serves a crucial role in enabling the optimization process, ensuring efficient movement within the search space, and guiding the algorithm towards discovering optimal or near-optimal solutions.
Let's break down the sequential code into segments and provide a cohesive explanation of each part:

A. Initialization

The initial segment sets up necessary variables and arrays to manage the optimization process. It initializes the random number generator to ensure different random sequences across multiple runs of the program. This randomness is essential in algorithms like MFO for diversifying the search space exploration.
Variables like start, finish, and arrays (population, flames, fitness, flames_fitness) facilitate time measurement and storage of moth positions, fitness, and related data.

B. Population Initialization

Random initial positions are assigned to moths within the specified search space. The initialization of moths is crucial as it influences the starting point for the optimization process. Each moth immediately undergoes fitness evaluation. Accurate fitness evaluation is pivotal for determining solution quality and guiding the whole optimization process.

C. Optimization Loop

The loop updates moth positions based on information from the flames, aiding moths to move towards better solutions. This step iteratively refines the solutions towards optimal or near-optimal configurations.
After updating positions, fitness for the new positions is calculated. Flames are updated if superior solutions are discovered. This step keeps track of the best solutions found so far. Identifying and updating the best solution among flames is essential as it helps monitor the overall progress and determines the final optimal or near-optimal solution.
The early stopping check prevents unnecessary iterations once a stopping criterion, like convergence, is met. It minimizes computational resources and prevents overfitting to the data. For the computation of the final results, this check was disabled.
Adapting the number of flames, if required, allows flexibility in the population size, potentially enhancing exploration and exploitation of the search space. Again, for the final computations, this option was disabled and the number of kept flames was fixed through the whole execution.
Finally, for each iteration, adjusting the beta value for moth movement ensures controlled and effective search space exploration by regulating the influence of the best solutions on other moths.

D. Finalization and Output

Computing execution time measures the algorithm's efficiency, facilitating comparisons and performance evaluation. Presenting the best fitness value and elapsed time provides insights into the optimization outcome and aids in analyzing the algorithm's effectiveness, although this wasn't the focus of our experiments, so no results about this aspect are provided. Finally, properly finalizing the environment is necessary for a clean termination of the processes.

E. Pseudo-code

Note that it still uses MPI, but that is just because it's needed to compute the time using the *MPI_Wtime* function.
The code can be found at 2 after the conclusion.

VI. Parallel solution

We performed the transformation of a sequential solution into a parallel one by leveraging the Message Passing Interface (MPI) paradigm, aiming to distribute the workload across multiple processes. The motivation behind this transformation is primarily to enhance the algorithm's performance by executing tasks concurrently, reducing computation time and potentially achieving faster results.
While parallel algorithms can significantly accelerate computations, it may not always guarantee improvements. Challenges such as communication overhead, load balancing, and synchronization among processes can appear, potentially negating the expected performance gains. Therefore, this chapter critically emphasizes how this adaptation affects the overall efficiency and scalability of the solution.
Let's break down the updated parallel code into segments and provide an explanation of each part:

A. MPI Initialization

Initializes the MPI environment, allowing communication between different processes. This initialization is crucial to start parallel execution.

B. Workload Distribution

Obtain the rank and total number of processes in the MPI communicator (*MPI_COMM_WORLD*). These identifiers help in workload division and synchronization. Dividing the

population among processes ensures that each process handles a portion of the total workload to execute the algorithm in parallel. Load balancing is critical for efficient parallel computing, preventing process under-utilize or overload.

C. Local Population Initialization

Allocating local arrays (*local_population*, *local_flames*, etc.) allows each process to store its subset of the population and intermediate results separately. This segregation prevents data conflicts and facilitates efficient data manipulation within each process.
Initializing the local population, calculating fitness values, and preparing arrays for flames and fitness help start the MFO algorithm independently on each process. This step sets the groundwork for parallel execution.

D. Optimization Loop

The main iterative loop runs the MFO algorithm independently within each process. This parallel execution is crucial for speeding up the optimization process as multiple processes work simultaneously on their local data.
Each process updates its local population and flames based on the MFO algorithm rules. This step involves performing computations on a subset of the population, enhancing parallelism and reducing computational time.

E. Adaptive Parameters and Communication

Modifying parameters like *n_flames* based on specific conditions within each process is essential for adapting the algorithm's behavior. These adaptations ensure exploration and exploitation within each sub search space.
Broadcasting the updated beta value among processes allows synchronization by sharing the latest information, since that variable is only dependent on the number of the iteration and therefore just needs to be computed once and can be shared immediately after. Synchronization is crucial to maintain consistency in algorithm parameters across all processes.

F. Synchronization

Inserting barriers at each iteration (*MPI_Barrier*) ensures that all processes reach the same point before moving to the next iteration. Synchronization is vital to facilitate collective actions.
While this operation isn't essential in our code due to the threads' disjoint subpopulations, preventing any interference among them, it is also unnecessary as it follows a broadcast operation that implies an implicit barrier just before it. However, we retained it for clarity's sake and because its presence doesn't affect the performance.

G. Global Reduction

Conducting global reduction (*MPI_Allreduce*) gathers information from all processes to find the minimum fitness value. This collective operation aids in determining the best solution or fitness value among all processes and their respective sub populations.

H. Finalization and Output

This step is kept the same as in the sequential case, showing the execution time and the best fitness found across all sub populations.

I. Pseudo-code

For this solution, inspiration was taken from the slides of the course.
This pseudocode represents the structure and flow of the MPI-based parallel MFO algorithm, outlining the key steps involved in distributing the workload among MPI processes, performing parallel optimization, handling adaptive changes, and synchronizations of the processes.
The code can be found at 3 after the conclusion.

VII. Hybrid Strategy

We decided to use MPI for initial distribution of work among different processes, especially when dealing with large populations, then, within each MPI process, use OpenMP to exploit thread-level parallelism for tasks that involve independent data or computations.
Another idea was to implement Dynamic Load Balancing so that it could redistribute workload among processes if some processes finish their tasks earlier than others, and manage thread allocation based on varying workloads within a process, but we immediately found the overhead to be too high for our modest application.
We know that overuse of synchronization primitives, such as barriers, or excessive communication between threads/processes can introduce overhead, hence why we decided to opt for a single explicit barrier at the end of each iteration.

VIII. Hybrid solution

This section introduces the integration of the Open Multi-Processing (OpenMP) paradigm into the existing MPI solution, aiming to create a hybrid parallel version. This hybrid approach further subdivides the workload, distributing computation among threads within each of the MPI processes. The strategy of dividing the workload equally among MPI processes and employing OpenMP within each process ensures effective utilization of computational resources, reducing idle times and enhancing overall performance.
The integration of OpenMP into an MPI-based parallel solution seeks to harness the advantages of both paradigms. OpenMP operates on shared-memory architectures, allowing threads to access shared data efficiently. By combining these paradigms, the goal is to enhance performance by efficiently utilizing both distributed and shared memory architectures.
However, the effectiveness of this hybrid approach can vary based on several factors. Utilizing multiple levels of parallelism introduces complexities in synchronization, load balancing, and data management. While the hybrid model can potentially improve performance by reducing communication overhead and exploiting shared memory, improper implementation might lead to increased overhead and complexities,

resulting in diminished performance gains. This is why we present not only the standard first Hybrid solution, but also the results of its fastest configuration with the least amount of overhead.
Let's break down the updated hybrid code into segments and provide a cohesive explanation of each part:

A. MPI Initialization

By initializing MPI, the program was already prepared for communication between different processes. Again, dividing the population into chunks ensures an equal distribution among MPI processes, maintaining load balance and minimizing times.

B. Workload Distribution and Local Population Initialization

Just as in the previous case, allocating memory for local variables based on chunk size ensures each process has its dedicated space for computation.
Now, leveraging OpenMP's parallelism, the initialization of local populations is distributed across multiple threads. This approach enhances efficiency by concurrently initializing multiple elements within the multiple processes.

C. Optimization Loop

OpenMP ensures that each process concurrently updates its local population, utilizing the available CPU cores efficiently. Individual threads compute fitness values and compare them locally to determine the best fitness values for each local population. OpenMP enables these comparisons to happen simultaneously, hoping to reduce the overall execution time.

D. Critical Section

The critical section is crucial due to the concurrent access to a shared variable that holds essential information, namely the *min_flame_fitness* variable, across multiple threads. Since multiple threads access and update that variable concurrently, updating shared variables simultaneously without proper synchronization might lead to race conditions, causing inconsistent or erroneous results.

E. Synchronization and Global Reduction

The *MPI_Barrier* was kept even though one could derive the same conclusions said for the previous solution.
MPI's collective communication operation (*MPI_Allreduce*) aggregates the best fitness values across all MPI processes to determine the global minimum fitness. This synchronization ensures the correct global result across all processes. This step is identical as the one in the previous solution.

F. Finalization and Output

Again, this step performs the same operations described in the previous solutions, printing the execution time and the best fitness found across all sub populations.

G. Pseudo-code

Below is the pseudocode for the MPI-OpenMP hybrid implementation, detailing the main steps involved in the parallelization strategy.
The code can be found at 4 after the conclusion.

IX. Alternative Hybrid Strategy

We noticed that even though the properties showed by the first hybrid solution were great, it was evident that further enhancements could be made to beat the execution time of the previous solution. With this objective in mind, a decision was made to explore an alternative implementation.
The focus shifted towards refining the system by minimizing the overhead associated with thread creations. To achieve this, a strategy was adopted wherein threads are generated at the start and promptly tasked with the subdivision of the problem's dimensionality into smaller, more manageable units.
This approach aimed to optimize the process by efficiently allocating resources from the start, thereby fostering an environment conducive to surpassing the previous parallel solution in terms of efficiency and performance.

X. Alternative Hybrid Solution

We will describe the intricacies of the new implementation through the upcoming subsections, delineating its pivotal role in surpassing the priors solution.

A. MPI Initialization

Again, the rank and size determine how to distribute the workload among processes, essential for parallel computing. The start and end indices are calculated to define the range of elements assigned to each process.

B. Workload Distribution and Local Population Initialization

Local arrays (*local_population*, *local_flames*, *local_fitness*) hold the local populations, flame arrays, and fitness values for each process, enabling parallel computation within each process.
The thread management parameters optimize the workload distribution among OpenMP threads, enhancing intra-process parallelism.

C. OpenMP Parallel Region and Optimization Loop

This time we immediately create a parallel region for OpenMP. The loops within this region initialize the local population and perform the MFO iterations in parallel among threads. This parallel region exploits multi-threading within each process, enabling simultaneous execution of population initialization, fitness evaluation, and iteration updates, enhancing computational speed.
Correctness is kept exploiting critical sections and atomic operations.

D. Global Reduction

Throughout the optimization process, individual threads within each process independently identify the most optimal solution within their respective domain.
Subsequently, the MPI Allreduce operation efficiently compares and synchronizes the local best solutions obtained from each process. This synchronization enables the alignment of the best fitness values across all processes so that the master process knows the global minimum fitness.

E. Finalization and Output

Once more, this phase executes identical operations outlined in prior solutions, displaying both the execution time and the most optimal fitness discovered among all sub-populations.

F. Pseudo-code

Updated pseudocode for the alternative MPI-OpenMP hybrid solution, implementing the previously described key-points.
The code can be found at 5 after the conclusion.

XI. Operations Suitable for Parallelization

Evaluating fitness for individual moths/flames can be parallelized efficiently. Each moth's fitness calculation is typically independent, allowing concurrent execution across multiple threads or processes. Utilizing OpenMP's parallel for loops or MPI's process distribution would facilitate parallel fitness evaluations.
Updating moth/flame positions based on certain rules can be parallelized effectively. OpenMP's parallel constructs or MPI's process distribution can facilitate simultaneous updates across multiple moths or processes.
Determining the global best solution involves finding the minimum fitness value across all moths/flames. While parallelization is possible, it requires synchronization. MPI can be used for communication and synchronization among processes to determine the global best solution efficiently.

XII. Operations Less Incline to Parallelization

The different iterations of the algorithm are not parallelizable because this is a process that works through refinement over multiple sequential iterations, and therefore they must be performed one after the other.
If the number of flames is set to be not fixed, sorting the local flames or indexes might be complex to parallelize efficiently within the sub populations due to dependencies on memory access patterns. Parallel sorting across threads might incur substantial communication overhead due to data exchanges and synchronization requirements.

XIII. Data Structures Impact on Parallelization

Arrays often offer better data locality, facilitating parallelization due to contiguous memory access. On the other hand, dependencies might pose synchronization challenges when multiple threads/processes access shared data.
This is why using structures optimized for parallel processing, such as disjoint data for independent operations like in our case, can reduce synchronization and communication overheads, although we noticed that adapting algorithms to parallel-friendly structures might require significant redesign and could impact the overall algorithm complexity.

XIV. Data Dependencies

Some calculations may depend on the results of others, creating potential bottlenecks or requiring synchronization. Our biggest problem for the parallelization were loop-carried dependencies between iterations and DataFlow dependencies each time the best solution within each sub population needed to be updated. In fact, in that last case there happen to be all kinds of DataFlow dependencies that we saw in the lectures: flow-dependence, anti-dependence, and output-dependence, which have been solved with the use of the critical section.

Algorithm 1 Example of a Critical Section

```
#pragma omp critical
{
if local_fitness[i] < local_best_fitness then
    local_best_fitness = local_fitness[i]
end if
}
```

XV. Conclusion

The report's analysis of four different approaches: serial/sequential, MPI-only, and two hybrid versions, highlighted crucial factors impacting their execution times.
Among the problem's variables, the dimensionality of the problem influenced the most the execution time. Following closely was the population size, which also notably affected the algorithms' performance.
Within the observed trends, maintaining consistent dimensionality showed a clear linear relationship between execution time and population size. As the population size increased, so did the execution time, displaying a proportional increment.
The number of processes employed exhibited an inverse linear relationship with execution time. With more processes utilized, the execution time decreased proportionally.
While the initial hybrid implementation aligned with these trends, it was slower than the MPI-only solution. Consequently, we designed an improved hybrid solution, showing superior performance over all previous alternative, especially as the population size increased. Even though (with our parameters) it doesn't work with smaller populations, this revised hybrid solution consistently outperformed other approaches, proving itself as the most efficient choice across various parameters, outshining both the serial, MPI-only, and the initial hybrid approaches.

A. Speedup Analysis

Speedup: ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the

parallel algorithm to solve the same problem on p processors (Ts / Tp).
Fixing our analysis on the scenario with $POPULATION\_SIZE = 200$, here is the speedup compared to the serial algorithm for each of the our configurations:

- (MPI-only): 2 processes $220.47/99.51 = 2.22$
- (MPI-only): 4 processes $220.47/51.28 = 4.30$
- (MPI-only): 8 processes $220.47/25.43 = 8.69$
- (first Hybrid): 2 processes $220.47/206.88 = 1.06$
- (first Hybrid): 4 processes $220.47/117.69 = 1.87$
- (first Hybrid): 8 processes $220.47/61.97 = 3.56$
- (fastest first Hybrid): 8 processes $220.47/27.01 = 8.16$
- (fastest alternative Hybrid): 2 processes $220.47/28.08 = 7.85$
- (fastest alternative Hybrid): 4 processes $220.47/26.85 = 8.21$
- (fastest alternative Hybrid): 8 processes $220.47/17.19 = 12.82$

So we can confidently say that the achieved results are often very promising, especially for the 8 processes MPI-only solution and the fastest alternative Hybdrid one.

B. Efficiency Analysis

Efficiency: ratio of Speedup to the number of processors, it measures the fraction of time for which a processor is usefully utilized (S / p) In the case of linear speedup, $E = 1$.
Computing the efficiency for the previous speedups we obtain:

- (MPI-only): 2 processes $2.22/2 = 1.11$
- (MPI-only): 4 processes $4.30/4 = 1.08$
- (MPI-only): 8 processes $8.69/8 = 1.09$
- (first Hybrid): 2 processes $1.06/2 = 0.53$
- (first Hybrid): 4 processes $1.87/4 = 0.47$
- (first Hybrid): 8 processes $3.56/8 = 0.45$
- (fastest first Hybrid): 8 processes $8.16/8 = 1.02$
- (fastest alternative Hybrid): 2 processes $7.85/2 = 3.93$
- (fastest alternative Hybrid): 4 processes $8.21/4 = 2.05$
- (fastest alternative Hybrid): 8 processes $12.82/8 = 1.60$

Again, the results of the first Hybrid proposal are the only non-satisfying ones, with all the others being even better than the linear scenario. This proved the importance of good overhead management.

C. Strong Scalability Analysis

Strong scalability measures how effectively a parallel algorithm maintains or improves its performance when the problem size stays constant, but the number of processors or computing resources increases.
Looking at our results, we can see that often the execution time halves each time the number of processes and cores is doubled, proving good strong scalability. The only approach with less good strong scalability seems to be the Fastest Alternative Hybrid solution, where the increase in number of processes and cores doesn't reflect the same good achievements, although still being the fastest.

D. Weak Scalability Analysis

Weak scalability, on the other hand, assesses a system's ability to maintain or enhance efficiency as both the problem size and the number of processing units grow proportionally, so measures if the workload per processor remains constant as the system scales up.
Analyzing our execution times, we can say that there is also good weak scalability. It is possible to see that almost any time the population size, the number of processes and the number of cores increase, the execution time is able to remain almost the same, with variations of very little amounts of seconds.

XVI. Results

Showing the results of our three approaches in different configurations.
In order to not overload too much the section, all the results that will be shown are performed with a fixed set of parameters, and only the changing ones will be highlighted.
The fixed parameters include: dimensionality ($NUM\_DIMENSIONS = 200000$), constant amount of flames ($FIX\_N\_FLAMES = 1$), number of maximum iterations ($MAX\_ITERATIONS = 50$), starting beta value ($BETA\_INIT = 1$), alpha ($ALPHA = 0.2$) and the fitness function ($Ackley$).

A. Fitness function

The *Ackley* function is a benchmark function. Its popularity stems from several characteristics:

- Contains a combination of multiple local minima and a single global minimum
- It is smooth and continuous, allowing for stochastic optimization techniques
- It is computationally inexpensive to evaluate, useful when it needs to be evaluated multiple times

B. Quantitative Results

POPULATION_SIZE	Domain	Time (s)
10	-5,+5	10.18
10	-20,+20	11.43
25	-5,+5	27.52
25	-20,+20	28.26
50	-5,+5	55.39
50	-20,+20	55.34
100	-5,+5	108.42
100	-20,+20	109.20
200	-5,+5	222.59
200	-20,+20	220.47

TABLE I: Results of the *Serial* ($select = 1$, $ncpus = 1$, #processes$= 1$) application on varying the population size of the algorithm ($POPULATION\_SIZE$). Notice how the domain does not influence the execution time significantly.

POPULATION_SIZE	ncpus	#processes	Time (s)
10	2	2	4.92
10	4	4	3.34
10	8	8	2.91
25	2	2	11.35
25	4	4	7.07
25	8	8	4.92
50	2	2	24.34
50	4	4	12.10
50	8	8	7.89
100	2	2	49.77
100	4	4	25.76
100	8	8	13.60
200	2	2	99.51
200	4	4	51.28
200	8	8	25.43
400	8	8	65.98
800	8	8	129.36

TABLE II: Results of the *MPI-only* ($select = 1$, domain: $[-20, +20]$) application on varying the population size of the algorithm ($POPULATION\_SIZE$)

POPULATION_SIZE	ncpus	#processes	Time (s)
10	2	2	12.58
10	4	4	14.04
10	8	8	10.31
25	2	2	24.94
25	4	4	17.34
25	8	8	10.26
50	2	2	51.29
50	4	4	30.32
50	8	8	17.11
100	2	2	99.65
100	4	4	57.47
100	8	8	34.21
200	2	2	206.88
200	4	4	117.69
200	8	8	61.97

TABLE III: Results of the first *Hybrid* ($select = 1$, domain: $[-20, +20]$) application on varying the population size of the algorithm ($POPULATION\_SIZE$)

XVII. Pseudo-codes

References

[1] Saroj Kumar Sahoo, Apu Kumar Saha. 2022. 'A Hybrid Moth Flame Optimization Algorithm for Global Optimization'. ResearchGate. https://www.researchgate.net/publication/361697805_A_Hybrid_Moth_Flame_Optimization_Algorithm_for_Global_Optimization

[2] Mohammad Nadimi-Shahraki. 2021. 'Migration-Based Moth-Flame Optimization Algorithm'. MDPI. https://www.mdpi.com/2227-9717/9/12/2276

[3] Mohammad Nadimi-Shahraki. 2023. 'MFO-SFR: An Enhanced Moth-Flame Optimization Algorithm Using an Effective Stagnation Finding and Replacing Strategy'. MDPI. https://www.mdpi.com/2227-7390/11/4/862

[4] Evolutionary Systems and Biomedical Engineering Lab. 2018. 'Benchmark Functions Github Repository'. GitHub. https://github.com/LaSEEB/openpso/blob/master/src/functions/basic/functions.c

[5] Saroj Kumar Sahoo, Apu Kumar Saha. 2022. 'Moth Flame Optimization: Theory, Modifications, Hybridizations, and Applications'. CIMNE. https://link.springer.com/article/10.1007/s11831-022-09801-z

POPULATION_SIZE	Time (s)
10	3.22
25	3.43
50	6.58
100	13.78
200	27.01
400	37.34
800	77.44

TABLE IV: Results of the fastest configuration of the first *Hybrid* ($select = 1$, $ncpus = 8$, #processes$= 8$, domain: $[-20, +20]$) application on varying the population size of the algorithm ($POPULATION\_SIZE$)

POPULATION_SIZE	ncpus	#processes	Time (s)
100	2	2	24.18
100	4	4	19.01
100	8	8	17.02
200	2	2	28.08
200	4	4	26.85
200	8	8	17.19
400	2	2	33.14
400	4	4	28.46
400	8	8	21.76
800	2	2	43.98
800	4	4	33.99
800	8	8	22.12

TABLE V: Results of the fastest configuration of the alternative *Hybrid* ($select = 1$, domain: $[-20, +20]$) application on varying the population size of the algorithm ($POPULATION\_SIZE$). Notice how there are no timings for population sizes less than 100, that is because the parameters chosen do not allow a small number of moths.

Algorithm 2 Sequential MFO Algorithm

```
Initialization of parameters
srand(time(NULL))
MPI_Init(NULL, NULL)
start = MPI_Wtime()
Initialize population, flames, fitness, flames_fitness, n_flames, early_stopping_counter, best_solution, best_fitness
for iteration ← 1 to MAX_ITERATIONS do
    Update population based on flames
    Boundary check for population
    Evaluate fitness of updated population
    Update flames if fitness improved
    Find the best flame and its fitness
    if early_stopping and early_stopping_counter ≥ EARLY_STOPPING_PATIENCE then
        break
    end if
    if not fix_n_flames and POPULATION_SIZE > 1 then
        Sort flames and keep best ones
    end if
    Update beta for next iteration
end for
finish = MPI_Wtime()
Print best fitness and time elapsed
MPI_Finalize()
```

Algorithm 3 MPI-only Parallel MFO Algorithm

```
Initialize variables
Initialize MPI environment
Obtain rank (rank) and total processes (num_processes)
if rank == 0 then
    start ← MPI_Wtime()
end if
Compute chunk_size, extra based on POPULATION_SIZE and num_processes
Divide population among processes
Initialize local data for each process
for iteration ← 0 to MAX_ITERATIONS do
    Update local population based on flames
    Boundary check for population
    Evaluate fitness of updated population
    Find the local best flame and its fitness
    if iteration > 0 then
        Handle early stopping
    end if
    Print iteration's best fitness for each process
    Handle adaptive changes for n_flames
    if rank == 0 then
        Compute and broadcast beta to all processes
    else
        Receive beta from process 0
    end if
    Synchronize processes using MPI_Barrier
end for
Perform global reduction to find minimum fitness using MPI
if rank == 0 then
    finish ← MPI_Wtime()
    Print global best fitness and time elapsed
end if
MPI_Finalize()
```

Algorithm 4 MPI-OpenMP Hybrid MFO Algorithm

```
Initialize variables
Initialize MPI environment
Obtain rank (rank) and total processes (num_processes)
if rank == 0 then
    start ← MPI_Wtime()
end if
Compute chunk_size, extra based on POPULATION_SIZE and num_processes
Divide population among processes
Initialize local data for each process
Parallel initialization of local population using OpenMP
for iteration ← 0 to MAX_ITERATIONS do
    Parallel population update and fitness calculation using OpenMP
    Determine minimum flame fitness using OpenMP:
        using OpenMP to access critical section
        Determine local best fitness
    if iteration > 0 then
        Handle early stopping
    end if
    Print iteration's best fitness for each process
    Handle adaptive changes for n_flames
    if rank == 0 then
        Compute and broadcast beta to all processes
    else
        Receive beta from process 0
    end if
    Synchronize processes using MPI_Barrier
end for
Perform global reduction to find minimum fitness using MPI
if rank == 0 then
    finish ← MPI_Wtime()
    Print global best fitness and time elapsed
end if
MPI_Finalize()
```

Algorithm 5 Alternative MPI-OpenMP Hybrid MFO Algorithm

Initialize variables
Initialize MPI environment
Obtain rank ($rank$) and total processes ($num\_processes$)
if $rank == 0$ **then**
    $start \leftarrow MPI\_Wtime()$
end if
Compute $chunk\_size$ based on $POPULATION\_SIZE$ and $num\_processes$
Divide population among processes
Allocate local data for each process
Define number of $total\_threads$ for each process
Compute $thread\_chunk\_size$
Start of the OpenMP parallel section
Define thread management variables
for each OpenMP thread in parallel section do
    Initialize thread-local population and calculate fitness
    Copy initial population to flames
    Critical section
    if thread-local fitness is better than process-local best fitness **then**
        Update thread-local best fitness
    end if
    Initialize thread-local beta
    for $iteration \leftarrow 0$ **to** $MAX\_ITERATIONS$ **do**
        Update population using MFO equations in parallel
        Bound population within limits
        if current moth fitness improved **then**
            Update local moth fitness and flame
        end if
        Calculate minimum flame fitness across threads
        Critical section
        if minimum flame fitness is better than process-local best fitness **then**
            Update process-local best fitness
        end if
        if iteration > 0 **then**
            Handle early stopping
        end if
        Print iteration details
        Update thread-local beta for next iteration
    end for
end for

Perform global reduction to find minimum fitness
if $rank == 0$ **then**
    $finish \leftarrow MPI\_Wtime()$
    Print global best fitness and time elapsed
end if
MPI_Finalize()