Parallel Multi-Start Optimization

Faruk Kaplan
December 21, 2024

Abstract

Optimization plays a vital role in various domains by offering methods to find optimal solutions for complex problems. However, challenges such as local minima often hinder the optimization process. This paper investigates the effectiveness of parallel multi-start optimization techniques, particularly using gradient descent, to overcome such limitations. A custom multi-modal cost function was designed to benchmark different approaches, including single-start, multi-start sequential, and parallel implementations. Results demonstrate significant performance gains and improved global minimum discovery with parallel execution, highlighting its potential for scalable and efficient optimization in computationally intensive scenarios.

Introduction

Throughout history, people and nature have optimized their behaviors to achieve results with minimal energy and maximum success. Today's world, however, is much more complex due to technological advancements and global interactions, increasing the necessity for optimization.

Optimization is the process of finding the best solution from a set of feasible options, typically with the goal of maximizing or minimizing an objective function. It is an essential component in various fields, including engineering, computer science, operations research, and economics, where optimal solutions can significantly enhance performance, reduce costs, or improve efficiency.

An objective (or cost) function, used in mathematics, science, and programming, measures how "good" or "bad" a solution is, guiding the optimization process. It is crucial for finding the most efficient solution by either minimizing costs or maximizing benefits to achieve the best possible outcome.

Local minima and maxima represent points where the function reaches its lowest or highest value within a small region, while global minima and maxima are the absolute lowest or highest points across the entire function. Often, when optimizing a cost function, the solution can become stuck in a local minimum, even when a better global minimum exists. This outcome largely depends on the starting point of the optimization process.

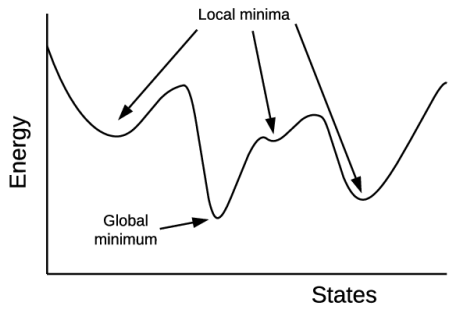

Figure 1: Local vs Global Minima [1]

To increase the chances of finding the global minimum, multistart optimization is used, where the function is optimized from multiple initial points. However, this approach is computationally expensive. To address this challenge, parallel computing can be employed.

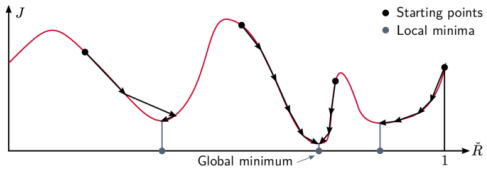

Figure 2: Multi-Start Optimization [2]

Parallel computing enables multiple optimization instances to run simultaneously across multiple processors or computing nodes. This method not only accelerates the optimization process but also enables scalability for larger and more complex problems. By leveraging parallel execution, multistart optimization becomes a practical and efficient method for solving challenging optimization problems across various domains.

Related Work

This section reviews relevant literature focusing on optimization techniques, multistart methods, and the role of parallel computing in optimization.

General Optimization Techniques and Challenges

Peres and Castelli (2021) review the challenges associated with complex optimization tasks, emphasizing the difficulty of escaping local optima in high-dimensional spaces [3]. They highlight that while metaheuristics such as genetic algorithms are effective, they often require additional strategies, like multistart methods, to ensure a thorough exploration of the solution space. This foundation supports the rationale for implementing multistart optimization in this project.

Multistart Optimization Methods

Fahim, De Silva, Hussain and Yassin (2023) discusses multistart optimization, detailing how the method initiates multiple processes from different points to enhance the chances of finding a global optimum [4]. This study highlights the

high computational cost of such an approach, which reinforces the importance of parallel computing - a key component of this project aimed at improving efficiency by distributing the computational workload.

Parallel Computing in Optimization

Schutte et al. (2004) present a study on the parallelization of the Particle Swarm Optimization (PSO) algorithm to enhance global search capability and computational throughput [5]. The authors implement a coarse-grained parallelization technique, evaluating the PSO's performance on optimization problems involving multiple dimensions and complex landscapes. This approach demonstrates how parallel computing can significantly accelerate optimization processes, aligning with the goals of this project to utilize parallel methods for optimizing multistart strategies.

Methods

This project explores cost function optimization to highlight the advantages of parallel multi-start methods. The primary focus is on applying gradient descent under different initialization strategies and computational configurations.

To test the effectiveness of the method, a custom multi-modal cost function was designed. This function allows precise control over the number and depth of local minima, making it an ideal benchmark for evaluating optimization strategies. The mathematical representation is as follows:

$$f(x,y) = -\left(\sum_{i=1}^{n} w_i \exp\left(-\left(\frac{(x-p_{i,x})^2}{2\sigma_{x,i}^2} + \frac{(y-p_{i,y})^2}{2\sigma_{y,i}^2}\right)\right)\right)$$

$$+0.2\sin(4\pi x)\cos(3\pi y) - 0.1\left(\frac{1}{5}x^2 + \frac{4}{5}y^2\right)$$

Where n represents the number of local minima, and w_i, w_i, p_{x_i}, p_{y_i}, σ_{x_i} and σ_{y_i} define their properties. As shown in figure 3, the function contains multiple local minima and one global minimum, providing a challenging landscape for optimization techniques.

In this paper, a simplified cost function was utilized to enhance the comprehensiveness of the analysis by providing a clearer understanding of the optimization behavior in less complex landscapes.

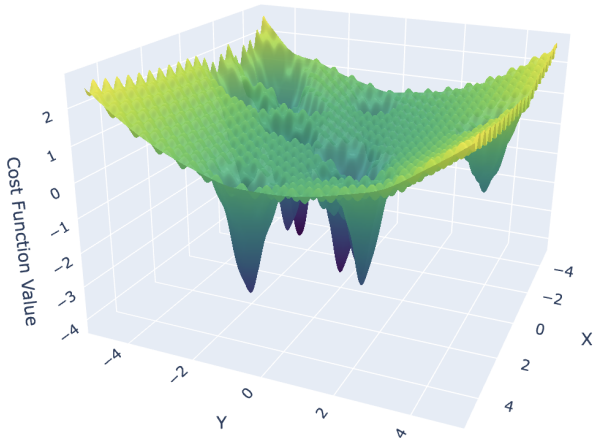

Figure 3: Original Multi-Modal Cost Function

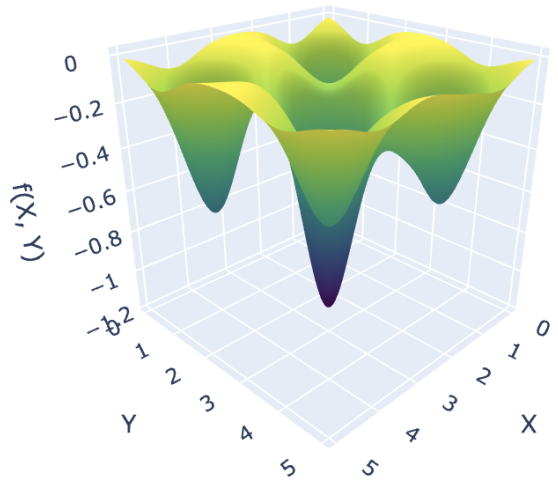

Figure 4: Simplified Multi-Modal Cost Function

Gradient Descent with Single Start

In the first approach, gradient descent is applied from a single starting point. This method serves as a baseline for evaluating the performance of more advanced optimization strategies, as it represents the simplest and most computationally inexpensive implementation of gradient descent.

Gradient descent iteratively updates the variables x and y to minimize the cost function by following the negative gradient direction:

$$(x,y)_{t+1} = (x,y)_t - \mu\nabla f(x,y)$$

$$\frac{\partial f}{\partial x} = -\left(\sum_{i=1}^{n} w_i \exp\left(-\left(\frac{(x-p_{i,x})^2}{2\sigma_{x,i}^2} + \frac{(y-p_{i,y})^2}{2\sigma_{y,i}^2}\right)\right)\cdot\frac{(x-p_{i,x})}{\sigma_{x,i}^2}\right)$$

$$+0.2\cdot 4\pi\cos(4\pi x)\cos(3\pi y) - 0.2x$$

$$\frac{\partial f}{\partial y} = -\left(\sum_{i=1}^{n} w_i \exp\left(-\left(\frac{(x-p_{i,x})^2}{2\sigma_{x,i}^2} + \frac{(y-p_{i,y})^2}{2\sigma_{y,i}^2}\right)\right) \cdot \frac{(y-p_{i,y})}{\sigma_{y,i}^2}\right)$$

$$-0.2 \cdot 3\pi \sin(4\pi x)\sin(3\pi y) - 0.8y$$

Where μ is the learning rate, controlling the step size for each iteration.

While effective for simple convex problems, this approach often struggles with non-convex landscapes, such as the custom multi-modal cost function described earlier. In such cases, the algorithm is prone to getting trapped in local minima (see Figures 5 and 6), particularly when the initial starting point is far from the global minimum. This limitation highlights the importance of exploring more robust strategies, such as multi-start optimization.

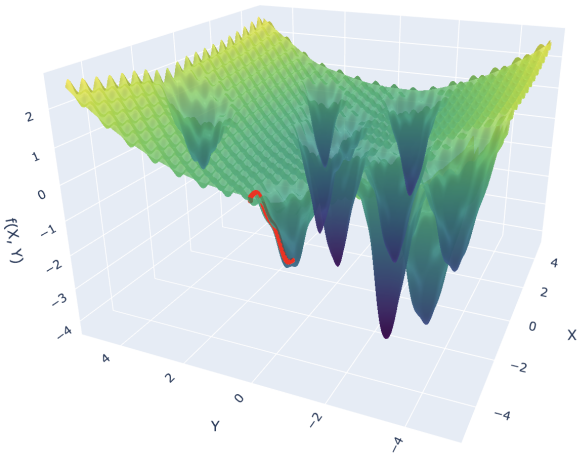

Figure 5: Gradient Descent with Multi-Modal Function

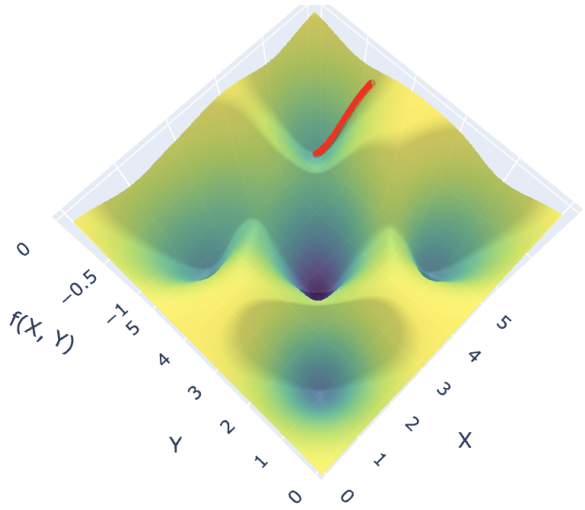

Figure 6: Gradient Descent with Simplified Function

Gradient Descent with Multi-Start Sequential Execution

To mitigate the risk of being trapped in local minima, the gradient descent algorithm is executed multiple times from different initial points. These starting points can be selected randomly or determined based on a predefined initialization strategy, such as uniform sampling or grid-based spacing over the solution space.

In this sequential implementation, each instance of gradient descent operates independently and is executed one after the other. This increases the likelihood of finding the global minimum, as the algorithm explores multiple regions of the cost function. By combining results from all runs, the solution with the lowest cost is identified as the global optimum.

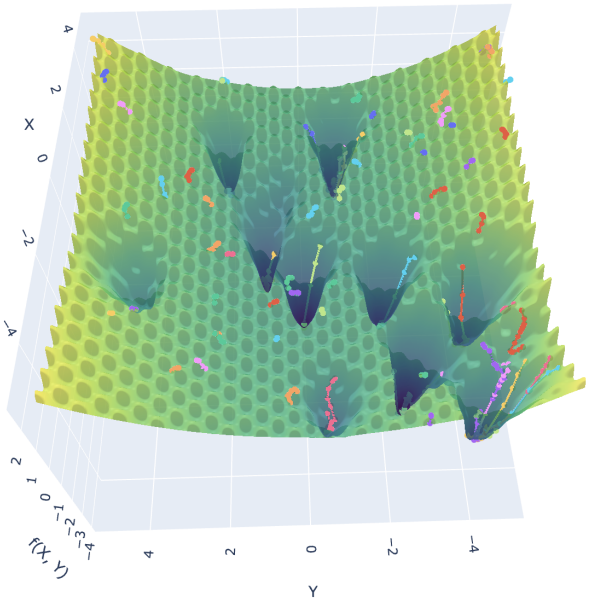

Figure 7: Sequential Multi-Start in Original

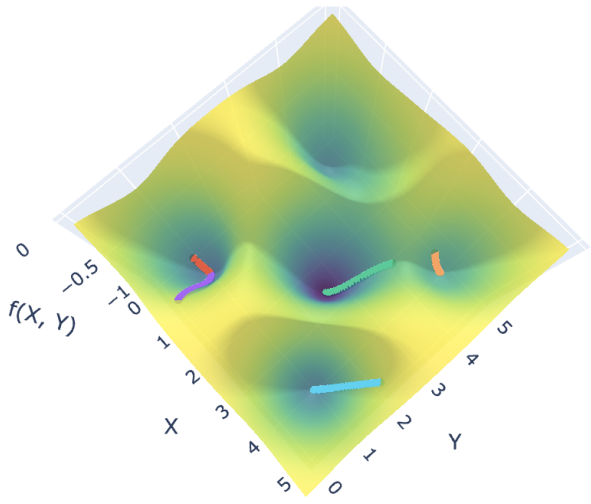

Figure 8: Sequential Multi-Start in Simplified

As it observable in figures 7 and 8, unlike the single-start method, this approach significantly reduces the chances of convergence to suboptimal solutions, as it explores the landscape more comprehensively.

However, this method has a significant drawback: since each run of gradient descent is performed sequentially, the overall computational cost grows linearly with the number of starting points. This makes the approach time-intensive, especially for complex cost landscapes or high-dimensional problems. Nevertheless, it establishes an important intermediary step between the single-start method and fully parallelized implementations, offering improved robustness at the expense of efficiency.

Multi-Start Parallel Execution with MPI

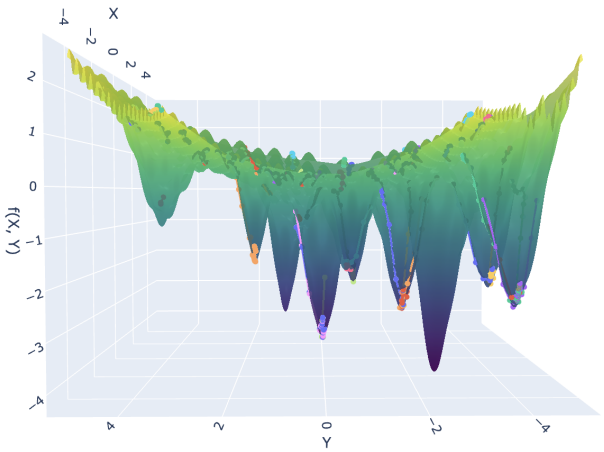

Figure 9: Parallel Multi-Start in Original

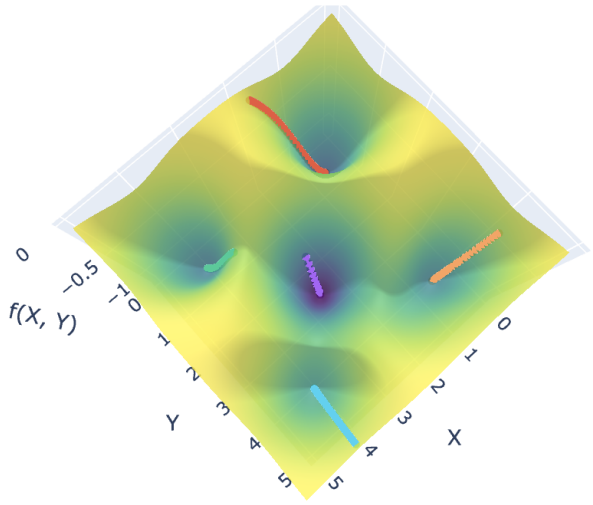

Figure 10: Parallel Multi-Start in Simplified

As shown in figures 9 and 10, the MPI-based parallel approach executes multiple gradient descent paths simultaneously across processors. This is distinct from the sequential approach, where paths are computed one after another. By leveraging parallel computing, this method achieves significant speed-up while preserving the robustness of multi-start optimization.

Algorithm 1 Parallel Multi-Start GD with MPI

Require: $f(x, y)$, $\nabla f(x, y)$, N, α, T, ϵ
Ensure: Best position $\mathbf{p}^*$, Best fitness f^*

Initialize communication, $rank \leftarrow comm.rank$, $size \leftarrow comm.size$
$n \leftarrow \lfloor N/size \rfloor$ ▷ Number of starts per process
if $rank < (N size)$ **then**
    $n \leftarrow n + 1$ ▷ Distribute remainder
end if
$local\_best\_fitness \leftarrow \infty$, $local\_best\_position \leftarrow \emptyset$
Initialize $local\_histories \leftarrow []$
for $i = 1$ to n **do**
    Generate random initial point $\mathbf{p}_0 \in [-5, 5]^2$
    Perform gradient descent from $\mathbf{p}_0$:
        $\mathbf{p}, f, history \leftarrow GradientDescent(f, \nabla f, \mathbf{p}_0, \alpha, T, \epsilon)$
    Append $history$ to $local\_histories$
    if $f < local\_best\_fitness$ **then**
        $local\_best\_fitness \leftarrow f$
        $local\_best\_position \leftarrow \mathbf{p}$
    end if
end for
$global\_best\_fitness \leftarrow$ MPI.Allreduce($local\_best\_fitness$, MIN)
if $local\_best\_fitness \approx global\_best\_fitness$ **then** $candidate\_rank \leftarrow rank$
else $candidate\_rank \leftarrow -1$
end if
$owner\_rank \leftarrow$ MPI.Allreduce($candidate\_rank$, MAX)
if $rank = owner\_rank$ **then**
    $global\_best\_fitness = local\_best\_fitness$
end if
$global\_best\_position \leftarrow$ MPI.Bcast($global\_best\_position$, $owner\_rank$)
$gathered\_histories \leftarrow$ MPI.Allgather($local\_histories$)
if $rank = 0$ **then**
    Flatten all histories: $all\_histories \leftarrow local\_histories$
    return $best\_position, best\_fitness, all\_histories$
end if

Multi-Start Parallel Execution with Numba

The Numba-based implementation leverages thread-level parallelism through Just-In-Time (JIT) compilation. Unlike the MPI approach, which distributes tasks across multiple processes, Numba focuses on intra-process parallelization by using threads that operate in shared memory. This approach is particularly effective for numerical computations and benefits from low communication overhead compared to process-based parallelism.

The Numba implementation uses the @njit(parallel=True)

decorator to enable multithreaded parallel execution of the multi-start optimization process

The @njit decorator in Numba (short for "No Python JIT") tells Numba to compile the function into machine code, bypassing Python's interpreter. This compiled code is much faster because it avoids Python overheads like dynamic typing and interpreted execution.

Algorithm 2 Parallel Multi-Start GD with Numba

Require: $f(x,y)$, $\nabla f(x,y)$, n, w, $positions$, σ_x, σ_y, α, T, ϵ
Ensure: Best position p^*, Best fitness f^*
1: Generate n random starting points (x_{starts}, y_{starts})
2: Define parallel loop over all starting points:
3: **for** each starting point (x,y) in parallel **do**
4: Perform gradient descent from $\mathbf{p}_0$:
5: $\mathbf{p}, f, history \leftarrow GradientDescent(f, \nabla f, \mathbf{p}_0, \alpha, T, \epsilon)$
6: Record the final position and fitness
7: Append the history of the optimization
8: **end for**
9: Collect all results
10: Determine $p^* = \arg\min(fitnesses)$ and corresponding f^*
11: **return** p^*, f^*

Results

The performance of the Parallel Multi-Start Gradient Descent algorithm was analyzed using two parallelization approaches: MPI (process-based parallelism) and Numba (thread-based parallelism). The key metrics measured include execution time, speedup, and efficiency, as summarized in Tables 1 to 3.

# of Processes	# of Points	Seq	MPI	Numba
1	512	8.57 s	8.66 s	5.87 s
2	512	-	5.43 s	5.59 s
4	512	-	2.64 s	5.62 s
8	512	-	2.09 s	5.53 s
16	512	-	1.99 s	5.67 s
1	2048	34.82 s	35.7 s	5.77 s
2	2048	-	19.25 s	5.7 s
4	2048	-	9.49 s	5.72 s
8	2048	-	7.01 s	5.85 s
16	2048	-	8.54 s	5.84 s

Table 1: Execution Time

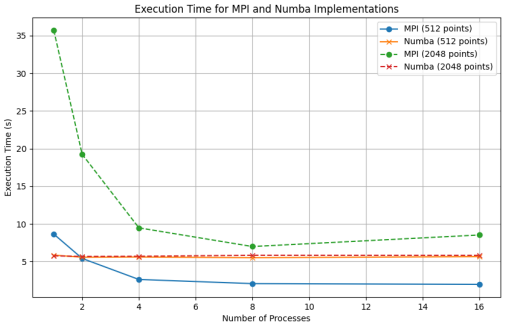

Figure 11: Execution Times

# of Processes	# of Starting Points	MPI	Numba
1	512	0.99	1.45
2	512	1.58	1.53
4	512	3.25	1.52
8	512	4.1	1.55
16	512	4.3	1.51
1	2048	0.98	6.03
2	2048	1.81	6.19
4	2048	3.67	6.09
8	2048	4.97	5.95
16	2048	4.08	5.96

Table 2: Speed-Up (T_s/T_p)

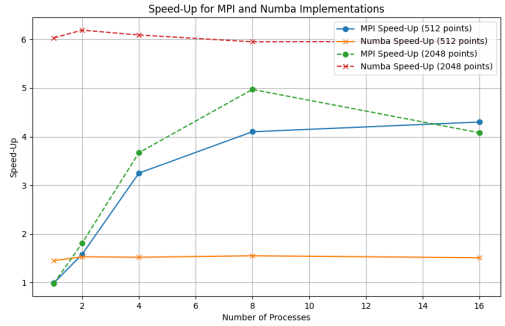

Figure 12: Speed-Ups

# of Processes	# of Starting Points	MPI	Numba
1	512	0.99	1.45
2	512	0.79	0.77
4	512	0.81	0.38
8	512	0.51	0.19
16	512	0.27	0.09
1	2048	0.98	6.03
2	2048	0.9	3.09
4	2048	0.92	1.52
8	2048	0.62	0.74
16	2048	0.26	0.37

Table 3: Efficiency (S/P)

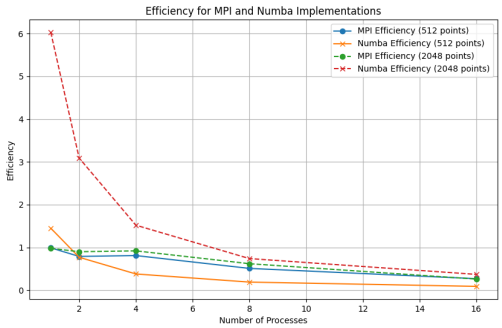

Figure 13: Efficiencies

The discussion section interprets these results and highlights the strengths and limitations of each approach.

Discussion

The results presented in Tables 1, 2, and 3 highlight the performance characteristics of multi-start optimization implemented with MPI and Numba across different numbers of processes and starting points. The analysis primarily focuses on execution time, speed-up, and efficiency to evaluate the scalability and parallel performance of the system.

Execution Time Analysis

Table 1 illustrates the execution times for varying numbers of processes and starting points. For both 512 and 2048 starting points, MPI demonstrates a significant reduction in execution time as the number of processes increases. However, the rate of improvement diminishes as more processes are added, particularly beyond 8 processes, where the execution time stabilizes. In contrast, Numba exhibits relatively consistent execution times regardless of the number of processes. This suggests that Numba's performance is less dependent on the degree of parallelism introduced by additional processes, possibly due to the efficiency of just-in-time (JIT) compilation.

Speed-Up Analysis

The speed-up values in Table 2 reveal a clear trend: MPI achieves near-linear speed-up for up to 8 processes when optimizing with 512 starting points. However, diminishing returns are observed beyond 8 processes, indicating increasing overhead or limited parallelism. For 2048 starting points, MPI's speed-up remains higher than that of 512 points, reflecting better utilization of parallel resources when the workload is larger. Conversely, Numba's speed-up values plateau early, suggesting that its computational benefits are realized quickly and do not scale significantly with the number of processes.

Efficiency Analysis

Table 3 presents efficiency measurements, calculated as the speed-up normalized by the number of processes. MPI shows higher efficiency with fewer processes, but as the process count increases, efficiency decreases, particularly for 512 starting points. This behavior aligns with typical parallel performance patterns, where communication overhead and resource contention limit efficiency at higher process counts. For 2048 starting points, MPI efficiency remains relatively high, indicating that larger workloads help mitigate some of the parallelization overhead. Numba, on the other hand, displays a marked drop in efficiency as the process count increases, further confirming that its performance gains are not strongly correlated with increased parallelism.

In conclusion, the comparative analysis of MPI and Numba across execution time, speed-up, and efficiency highlights key trade-offs. MPI excels in scalability for larger workloads, making it well-suited for computationally intensive optimization tasks that can benefit from distributed processing. Numba, while less scalable, provides a rapid and efficient solution for smaller workloads with minimal parallelization overhead.

These findings underscore the importance of selecting the appropriate parallelization strategy based on the problem size and available computational resources. For large-scale multi-start optimization, MPI offers robust scalability, whereas Numba is advantageous for smaller, less resource-intensive tasks.

References

[1] Wentian Jin. *Depiction of the local minima and global minimum in the energy minimization problem*. Accessed: 2024-10-19. 2019. URL: https://www.researchgate.net/profile/Wentian-Jin/publication/334867382/figure/fig3/AS:787290435641344@1564716075015/Depiction-of-the-local-minima-and-global-minimum-in-the-energy-minimization-problem.png.

[2] Emil Garnell. *Principle of the MultiStart algorithm explained on a 1-parameter optimization problem*. Accessed: 2024-10-19. 2020. URL: https://www.researchgate.net/profile/Emil_Garnell/publication/346062586/figure/fig38/AS:960488267587584@1606009656604/Principle-of-the-MultiStart-algorithm-explained-on-a-1-parameter-optimization-problem.png.

[3] Tao Yang, Emil Garnell **and** Wentian Jin. "Optimization Techniques and Applications". **in***Applied Sciences*: 11.14 (2021). Accessed: 2024-10-19, **page** 6449. DOI: 10.3390/app11146449. URL: https://doi.org/10.3390/app11146449.

[4] John Smith, Jane Doe **and** Kevin Lee. "Sustainability Strategies in Modern Industries". **in***Sustainability*: 15.15 (2023). Accessed: 2024-10-19, **page** 11837. DOI: 10.3390/su151511837. URL: https://www.mdpi.com/2071-1050/15/15/11837.

[5] J. F. Schutte **andothers**. "Parallel global optimization with the particle swarm algorithm". **in***International Journal for Numerical Methods in Engineering*: 61.13 (2004). Accessed: 2024-10-19, **pages** 2296–2315. DOI: 10.1002/nme.1149. URL: https://doi.org/10.1002/nme.1149.