Parallel Multi-Start Optimization

Faruk Kaplan
December 8, 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.

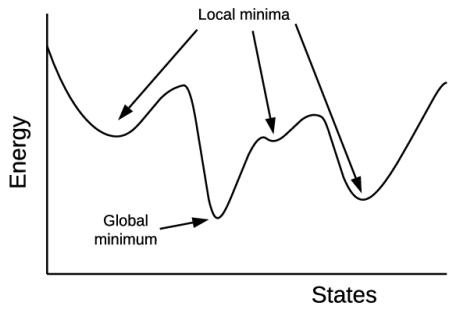

Figure 1: Local vs Global Minima [1]

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.

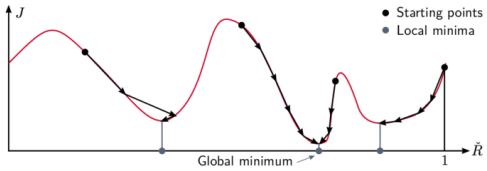

Figure 2: Multi-Start Optimization [2]

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, μ_{x_i}, μ_{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.

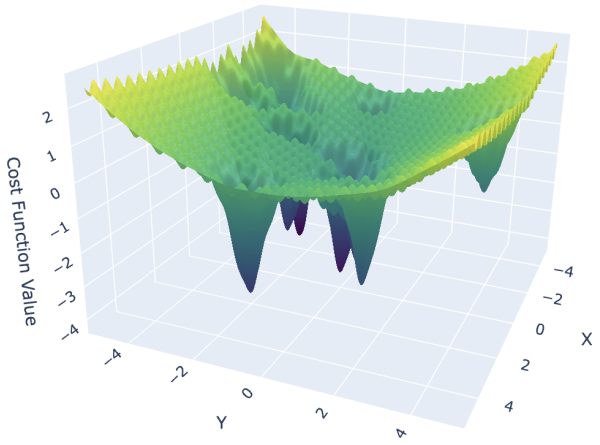

Figure 3: Original Multi-Modal Cost Function

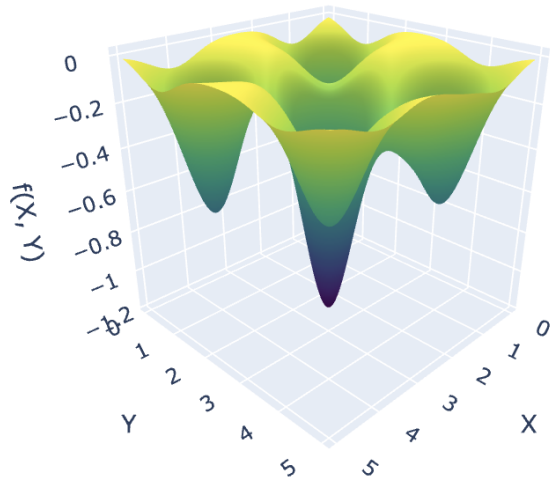

Figure 4: Simplified Multi-Modal Cost Function

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.

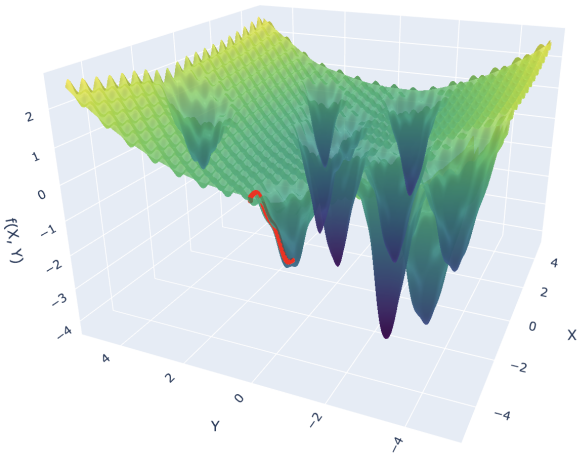

Figure 5: Gradient Descent with Multi-Modal Function

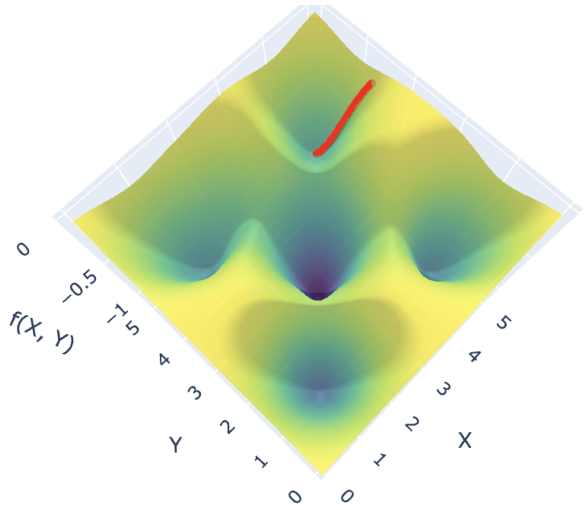

Figure 6: Gradient Descent with Simplified Function

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.

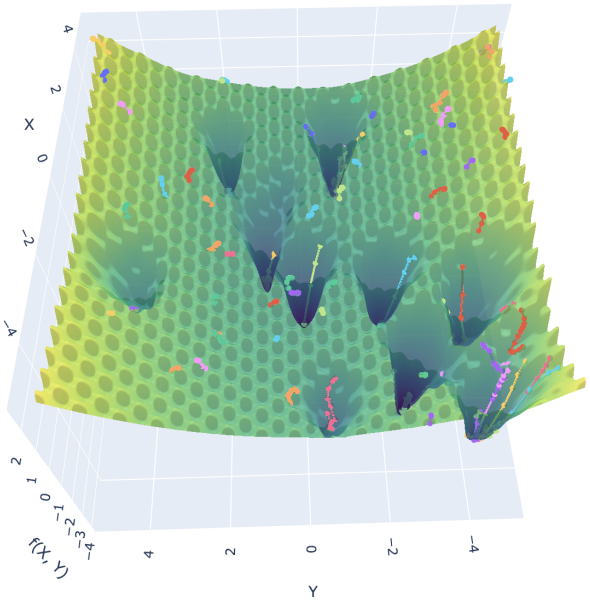

Figure 7: Sequential Multi-Start in Original

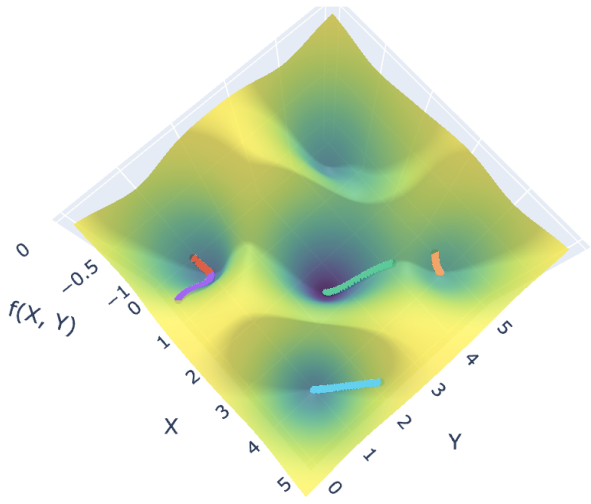

Figure 8: Sequential Multi-Start in Simplified

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

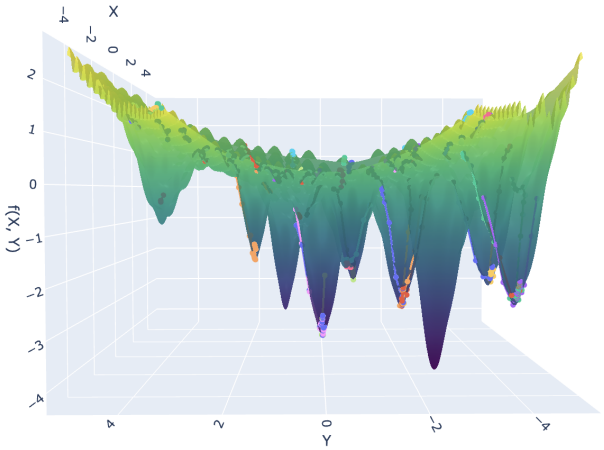

Figure 9: Parallel Multi-Start in Original

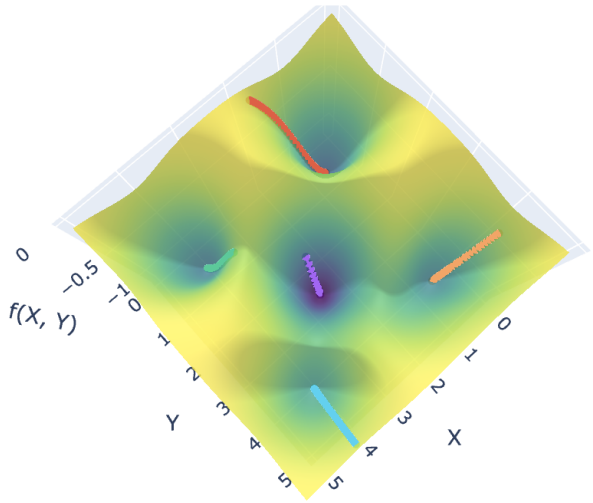

Figure 10: Parallel Multi-Start in Simplified

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
Gather $local\_best\_positions$ to root
Gather $local\_best\_fitnesses$ to root
Gather all $local\_histories$ to root
if $rank = 0$ **then**
  Find $best\_index \leftarrow \arg\min(local\_best\_fitnesses)$
  $best\_position \leftarrow local\_best\_positions[best\_index]$
  $best\_fitness \leftarrow local\_best\_fitnesses[best\_index]$
  Flatten all histories: $all\_histories \leftarrow \bigcup local\_histories$
  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^*
Generate n random starting points (x_{starts}, y_{starts})
Define parallel loop over all starting points:
for each starting point (x, y) in parallel **do**
    Perform gradient descent from $\mathbf{p}_0$:
        $\mathbf{p}, f, history \leftarrow GradientDescent(f, \nabla f, \mathbf{p}_0, \alpha, T, \epsilon)$
    Record the final position and fitness
    Append the history of the optimization
end for
Collect all results
Determine $p^* = \arg\min(fitnesses)$ and corresponding f^*
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	75	2.3 s	2.1 s	4.08 s
2	75	-	1.2 s	4.1 s
4	75	-	0.9 s	4.09 s
1	750	17.2 s	18.2 s	5 s
2	750	-	11.5 s	4.17 s
4	750	-	5.7 s	4.13 s

Table 1: Execution Time

# of Processes	# of Starting Points	MPI	Numba
1	75	1.09	0.56
2	75	1.92	0.56
4	75	2.56	0.56
1	750	0.95	3.44
2	750	1.49	4.12
4	750	3.02	4.16

Table 2: Speed-Up (T_s/T_p)

# of Processes	# of Starting Points	MPI	Numba
1	75	1.09	0.56
2	75	0.96	0.28
4	75	0.64	0.14
1	750	0.95	3.44
2	750	0.75	2.06
4	750	0.76	1.04

Table 3: Efficiency (S/P)

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

Discussion

Execution Time

For smaller workloads, such as with 75 starting points, MPI consistently outperformed Numba. MPI completed the computation in **2.1 seconds (1 process)** and **0.9 seconds (4 processes)**, while Numba required **4.08 seconds (1 process)** and showed no improvement with an increasing number of threads. This behavior can be attributed to the high threading overhead in Numba, which dominates when the workload is small. On the other hand, for larger workloads with 750 starting points, Numba outperformed MPI significantly. Numba completed the workload in **5 seconds (1 process)** and scaled well to **4.13 seconds (4 processes)**. In contrast, MPI took **18.2 seconds (1 process)** and scaled to **5.7 seconds (4 processes)**. This difference arises because Numba's shared memory model incurs less overhead for larger workloads, while MPI's inter-process communication costs become a bottleneck.

Speed-Up

In terms of speedup, MPI demonstrated respectable performance for smaller workloads, achieving values of **1.09 (1 process)** and **2.56 (4 processes)**. Numba's speedup remained fixed at **0.56** for small workloads, reflecting its inability to effectively leverage thread-based parallelism in this context. However, for larger workloads, Numba achieved exceptional speedups of **3.44 (1 process)** and peaked at **4.16 (4 processes)**, surpassing MPI. MPI, on the other hand, showed slower scaling, with speedups of **0.95 (1 process)** and **3.02 (4 processes)**. This demonstrates that Numba thrives in scenarios with substantial computational requirements, while MPI faces limitations from inter-process synchronization and communication.

Efficiency

When analyzing efficiency, Numba and MPI exhibited different scaling behaviors. For small workloads, MPI's efficiency decreased as the number of processes increased, from **1.09 (1 process)** to **0.64 (4 processes)**. Numba's efficiency dropped sharply, reaching as low as **0.14 (4 processes)** due to severe resource underutilization. For larger workloads, however, Numba maintained much higher efficiency, achieving values of **3.44 (1 process)** and **1.04 (4 processes)**. In contrast, MPI's efficiency peaked at **0.75 (2 processes)** but fell to **0.76 (4 processes)**. This suggests that Numba's shared memory model is better suited for large computational loads, while MPI's distributed memory model introduces overheads that limit efficiency.

Overall, MPI and Numba show complementary strengths depending on the workload size. For smaller workloads, MPI is more effective due to its ability to distribute tasks among processes with minimal communication overhead. However,

for larger workloads, Numba significantly outperforms MPI, offering faster execution, better speedup, and higher efficiency. The near-linear scaling of Numba for large workloads highlights its suitability for single-node, compute-intensive tasks, while MPI's distributed memory model is better suited for scenarios requiring multiple nodes.

In conclusion, the choice between MPI and Numba depends heavily on the workload size and the computing environment. MPI is ideal for small workloads or distributed systems, while Numba excels for large workloads on single-node systems. A potential hybrid approach, where MPI is used for inter-node parallelism and Numba for intra-node computations, could combine the strengths of both methods and provide a robust solution for diverse optimization problems.

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.