# Comparison Report: Multi-threading vs Multi-processing

## Introduction

This report compares two parallel computation strategies in Python, multi-threading and multi-processing using an example task, computing the sum of squares of a list of numbers.

### Task Description

The program splits a list of numbers into 4 chunks and calculates the sum of squares for each chunk in parallel. The intermediate results are then aggregated to get the final result.

* **Function:** sum\_of\_squares(numbers, result, index)
* **Goal:** Parallelize this computation using threads or processes.

## Implementation

### Multi-threading Implementation

**Code Highlights:**

* Uses **threading.Thread.**
* Shared list results stores partial results.
* Threads are joined at the end to ensure completion.

**Advantages:**

* Lightweighted lower memory usage.
* Fast context switching between threads.

### Multi- processing Implementation

**Code Highlights:**

* Uses **multiprocessing.Process** and **multiprocessing.Manager().list** to store results.
* Each process runs in its own Python interpreter.
* Processes are joined to synchronize.

**Advantages:**

* **True parallelism:** Utilizes multiple CPU cores.
* Better performance for CPU-bound tasks.

## Performance Comparison

For small input sizes, both multi-threading and multi-processing perform similarly due to the minimal computation involved. For example, with an input size of just 10 elements:

* **Multi-threading:** ~0.001 seconds
* **Multi-processing:** ~0.2 seconds

In this case, multi-threading is noticeably faster due to its lower overhead—creating threads is lightweight compared to processes. However, as the input size grows, multi-processing begins to outperform threading by better utilizing multiple CPU cores.

For instance, with an input size of one million elements:

* **Multi-threading:** ~5.4 seconds
* **Multi-processing:** ~5.0 seconds

Although the difference is subtle here, it becomes more significant with even larger datasets or more complex operations. This shift occurs because Python's Global Interpreter Lock (GIL) limits the effectiveness of threads in CPU-bound tasks, while multiprocessing spawns separate processes that bypass the GIL, enabling true parallel execution.

## Use Cases

|  |  |
| --- | --- |
| **Task** | **Approach** |
| I/O bound | Multi-threading |
| CPU bound | Multi-processing |
| Small data | Both are ideal |
| Large data | Multi-processing |

## Conclusion

Both approaches achieves similar results. The choice between threading and multiprocessing depends on the nature of the task.

* For lightweight, I/O-bound tasks, multi-threading is more efficient in terms of both memory and execution time.
* For heavy computation, such as large-scale data processing or complex mathematical operations, multi-processing is more suitable.
* In general, for CPU-bound tasks, multi-processing provides better performance by utilizing multiple CPU cores.