Multiprocessing Shared Array

Help me understand multiprocessing in Python and the Array class. What does it mean to have a shared array and when is it a good idea? Provide examples

Sure, Gabriel! Let's break this down step by step:

🧠 What Is `multiprocessing` in Python?

The `multiprocessing` module allows you to **run code in parallel** by creating separate processes. Each process has **its own memory space**, which is **different from threads** that share memory.

Why Use It?

Python's Global Interpreter Lock (GIL) limits **threads** from running Python bytecode truly in parallel. But **processes** can run on multiple CPU cores independently — making `multiprocessing` ideal for **CPU-bound tasks** (e.g., data processing, numerical computations).

🔗 Shared Memory and the `Array` Class

Since **each process has its own memory**, **sharing data between them isn't trivial**. Normally, you'd pass data through **pipes or queues**, which means **copying data between processes** — slow for large arrays.

To avoid this, Python provides **shared memory constructs** such as:

- `multiprocessing.Array` (for sharing C-style arrays)
- `multiprocessing.Value` (for single shared values)
- `multiprocessing.shared_memory` (more advanced, introduced in Python 3.8)

📦 `multiprocessing.Array`: What Is It?

`multiprocessing.Array` creates a **fixed-size, mutable array** in shared memory. All processes can read/write to it, making it useful when:

- You need **parallel access** to a dataset (e.g., matrix calculations)
- You want to **avoid copying** large data between processes

🔧 Example 1: Shared Array for Parallel Squaring

```python
import multiprocessing
import ctypes

def square(index, shared_array):
    shared_array[index] = shared_array[index] ** 2

if __name__ == '__main__':
    # Create a shared array of integers with initial values [1, 2, 3, 4]
    shared_array = multiprocessing.Array(ctypes.c_int, [1, 2, 3, 4])

    processes = []
    for i in range(4):
        p = multiprocessing.Process(target=square, args=(i, shared_array))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    print(list(shared_array))  # Output: [1, 4, 9, 16]
```

🧠 What's Going On?

- `ctypes.c_int` defines the array as holding integers.
- All processes access the **same shared memory**.
- This avoids copying and allows **in-place modification**.

🔐 But What About Race Conditions?

If multiple processes **write to the same element**, you may have **race conditions**. You can prevent this by:

- Using `Lock()` to guard access.
- Structuring the computation so each process writes to its **own index**.

⚠️ When Should You Use a Shared Array?

✅ Good Use Cases:

- Numerical computations where each process works on **a distinct chunk** of data.
- Shared buffers (like audio or image frames).
- Avoiding serialization cost (vs. Queue or Pipe).

🚫 Avoid When:

- You need dynamic resizing (Array is fixed-size).
- You need high-level operations (NumPy is better for this).
- Simpler inter-process communication (Queue or Pipe may be easier).

⚙️ Example 2: Shared Array With Lock

```python
def safe_increment(shared_array, lock):
    with lock:
        for i in range(len(shared_array)):
            shared_array[i] += 1

if __name__ == '__main__':
    shared_array = multiprocessing.Array(ctypes.c_int, [0, 0, 0])
    lock = multiprocessing.Lock()

    processes = [multiprocessing.Process(target=safe_increment, args=(shared_array, lock)) for _ in range(3)]

    for p in processes:
        p.start()
    for p in processes:
        p.join()

    print(list(shared_array))  # Output: [3, 3, 3]
```

Here, the `lock` ensures **mutual exclusion**, so no two processes write at the same time.

🧠 Want to Go Deeper?

You can explore:

- **`multiprocessing.shared_memory`** (Python 3.8+): Works directly with NumPy arrays.
- **`concurrent.futures.ProcessPoolExecutor`**: Higher-level API for parallelism.
- Vectorized shared memory using NumPy and **`multiprocessing.Array`**.

Let me know if you’d like to see an example integrating this with NumPy or a functional-style design!

I need to make several large matrix operations. Each process consumes from the same numpy array for their operations, but they don't write on it. For this case, what is your recommendation: using shared_memory or Array?

Great question, Gabriel — and you're thinking about exactly the right trade-offs.

🧪 Scenario:

You have:

- **Several large matrices** (e.g., NumPy arrays).
- Multiple **processes** that:
  - **Read** from a **shared NumPy array**.
  - **Don't modify** the array.

- **Write** results elsewhere (e.g., return values, another structure).

✅ Recommendation: Use `multiprocessing.shared_memory` (Python 3.8+)

Why?

1. **Efficient zero-copy sharing of NumPy arrays**:
    - Unlike **`multiprocessing.Array`**, **`shared_memory`** supports **true NumPy arrays** with slicing, broadcasting, etc.
    - No need to convert to/from flat C-types or use manual locking for read-only access.
2. **Better performance and ergonomics for large data**:
    - More natural if you're working with high-dimensional data (**`matrices`**, **`tensors`**, etc.).
    - Cleaner integration with NumPy.

🆚 Comparison: `Array` vs `shared_memory`

Feature	`multiprocessing.Array`	`multiprocessing.shared_memory`
NumPy support	✅ Indirect (needs **`np.frombuffer`**)	✅✅ Native, full support
Multidimensional arrays	😐 Needs manual reshaping	✅ Fully supported
Data copying	❌ No copy, but harder setup for read-only	❌ True zero-copy
Performance (for large data)	🐢 Slower due to type conversions	🚀 Faster for large read-only arrays
API simplicity	✅ Simpler for basic use	🧠 Slightly more verbose but powerful

---

🔧 Example: Sharing a NumPy Array Using `shared_memory`

```python
from multiprocessing import Process, shared_memory
import numpy as np

def worker(name, shape):
    # Attach to the existing shared memory block by name
    shm = shared_memory.SharedMemory(name=name)
    # Recreate the original array from shared memory
    existing_array = np.ndarray(shape, dtype=np.float64, buffer=shm.buf)

    # Perform read-only operations
    print("Sum:", np.sum(existing_array))

    # Clean up (do not unlink here, only the main process should do that)
    shm.close()

if __name__ == "__main__":
    # Original array
    data = np.random.rand(1000, 1000)
    # Create shared memory block
    shm = shared_memory.SharedMemory(create=True, size=data.nbytes)

    # Create NumPy array backed by shared memory
    shared_array = np.ndarray(data.shape, dtype=data.dtype, buffer=shm.buf)
    shared_array[:] = data[:]  # Copy data into shared memory

    # Start worker processes
    processes = [Process(target=worker, args=(shm.name, data.shape)) for _ in range(4)]
    for p in processes:
        p.start()
    for p in processes:
        p.join()

    # Clean up shared memory
    shm.close()
    shm.unlink()
```

💡 Notes:

- Each worker attaches to the shared memory by **name**.
- Since they **only read**, there's **no need for locks**.
- The main process owns and **cleans up** the shared memory (via **`unlink()`**).

✅ Summary

Use **`multiprocessing.shared_memory`** when:

- You have **large NumPy arrays**.
- You need **zero-copy, read-only access** across processes.
- You want **NumPy-friendly operations** and structure.

Only use **`multiprocessing.Array`** when:

- You're working with small to medium data.
- You don't want the extra setup of shared memory blocks.
- You don’t need full NumPy functionality.

Would you like an example where each worker **returns results** or writes them to another shared structure?

If I have two large arrays, can I copy both using the same SharedMemory object?

Excellent question, Gabriel — the short answer is:

> ✅ **Yes, you *can* store multiple arrays in a single `SharedMemory` block**, but you’ll need to **manually manage the layout and slicing** of that memory buffer.

Let’s walk through what that means, why you might do it, and how.

🧠 Why Share Multiple Arrays in One `SharedMemory`?

- 🔋 **Efficiency**: You reduce overhead from creating multiple shared memory segments.
- 💾 **Memory locality**: Arrays sit contiguously, which can improve cache performance.
- 📦 **Organizational control**: Especially useful if you're passing both arrays together as a unit.

⚠️ The Challenge: Manual Memory Layout

The `SharedMemory` buffer is just a flat chunk of bytes (`shm.buf`). You must:

1. Compute **each array’s size in bytes**.
2. Make sure they are **non-overlapping** inside the buffer.
3. Use slicing to reconstruct each array correctly.

🧮 Example: Two Arrays in One SharedMemory

Let’s say you have:

- `A`: shape `(1000, 1000)`, dtype `float64`
- `B`: shape `(1000,)`, dtype `float64`

Step-by-step code:

python

```python
import numpy as np
from multiprocessing import Process, shared_memory

def worker(name, shape_A, shape_B):
    shm = shared_memory.SharedMemory(name=name)

    # Total buffer
    buf = shm.buf

    # Recover array A
    size_A = np.prod(shape_A)
    A = np.ndarray(shape_A, dtype=np.float64, buffer=buf[:size_A * 8])

    # Recover array B
    B = np.ndarray(shape_B, dtype=np.float64, buffer=buf[size_A * 8:])

    print("Worker: A.sum() =", np.sum(A), ", B.mean() =", np.mean(B))

    shm.close()

if __name__ == "__main__":
    A = np.random.rand(1000, 1000)
    B = np.random.rand(1000)

    size_A = A.size * A.itemsize
    size_B = B.size * B.itemsize
    total_size = size_A + size_B
    shm = shared_memory.SharedMemory(create=True, size=total_size)
    buf = shm.buf

    # Create shared version of A
    shared_A = np.ndarray(A.shape, dtype=A.dtype, buffer=buf[:size_A])
    shared_A[:] = A

    # Create shared version of B
    shared_B = np.ndarray(B.shape, dtype=B.dtype, buffer=buf[size_A:])
    shared_B[:] = B

    # Launch worker
    p = Process(target=worker, args=(shm.name, A.shape, B.shape))
    p.start()
    p.join()

    shm.close()
    shm.unlink()
```

✅ Summary

Task	Yes/No	Details
Store multiple arrays in one `SharedMemory`	✅ Yes	Manually layout and slice buffer
Use standard NumPy syntax to slice	✅ Yes	`np.ndarray(..., buffer=buf[start:end])`
Automatic memory management	❌ No	You must compute byte offsets manually
Works well for read-only use	✅ Yes	Locking not needed

Would you like a version that also **writes results** or synchronizes using a `Lock` or `Barrier`?

Can I use two different objects for simplification?

Yes, Gabriel — you absolutely **can and often *should*** use **two separate `SharedMemory` objects** if your goal is **code clarity and maintainability** rather than raw optimization.

✅ Why Use Separate `SharedMemory` Objects?

🧠 Simpler:

- No need to manually calculate byte offsets or worry about alignment.
- Each array can be treated independently — no slicing errors.

🧹 Cleaner Code:

- You don't mix concerns — each array has a name, a shape, and a `SharedMemory` block.
- Easier to pass around and debug.

🔄 Still Efficient:

- Minor extra memory overhead for two blocks instead of one.
- No performance hit for read-only access, especially for large arrays.

📦 Example: Two Arrays with Separate SharedMemory Objects

```python
from multiprocessing import shared_memory, Process
import numpy as np

def worker(name_a, shape_a, name_b, shape_b):
    shm_a = shared_memory.SharedMemory(name=name_a)
    shm_b = shared_memory.SharedMemory(name=name_b)

    A = np.ndarray(shape_a, dtype=np.float64, buffer=shm_a.buf)
    B = np.ndarray(shape_b, dtype=np.float64, buffer=shm_b.buf)

    print("Worker: A.mean() =", np.mean(A), " | B.sum() =", np.sum(B))

    shm_a.close()
    shm_b.close()

if __name__ == "__main__":
    A = np.random.rand(1000, 1000)
    B = np.random.rand(1000)

    shm_a = shared_memory.SharedMemory(create=True, size=A.nbytes)
    shm_b = shared_memory.SharedMemory(create=True, size=B.nbytes)

    shared_A = np.ndarray(A.shape, dtype=A.dtype, buffer=shm_a.buf)
    shared_B = np.ndarray(B.shape, dtype=B.dtype, buffer=shm_b.buf)

    shared_A[:] = A
    shared_B[:] = B

    p = Process(target=worker, args=(shm_a.name, A.shape, shm_b.name, B.shape))
    p.start()
    p.join()

    shm_a.close()
    shm_b.close()
    shm_a.unlink()
    shm_b.unlink()
```

✅ When to Use Two SharedMemory Objects

Situation	Recommendation
You prioritize clarity	✅ Use separate blocks
Arrays are large but independent	✅ Use separate blocks
Arrays must be packed together	❌ Use a single block (but carefully)
You're managing dozens of arrays	⚠️ Consider structured packing, or shared-memory manager

---

💡 Rule of Thumb:

> Start with **separate shared memory objects** for **simplicity**.
> Optimize to a **single block only** if:

- Memory overhead is critical.
- You're managing **many small arrays**.
- You can handle the increased complexity safely.

Let me know if you want to generalize this setup with a utility class for shared NumPy arrays.

I ran a profiler on my code and found out the following:

cProfile profiler output:

```
bash
    22/18   20.542    0.934   41.253    2.292 {built-in method posix.read}
        3    0.152    0.051    0.152    0.051 {built-in method numpy.array}
       45    0.113    0.003    0.113    0.003 {method 'poll' of 'select.poll' objects}
        1    0.059    0.059    0.210    0.210 simulate.py:11(generates_portfolios_by_idxs)
       45    0.019    0.000    0.019    0.000 selectors.py:272(close)
        8    0.014    0.002    0.014    0.002 {built-in method posix.fork}
        1    0.013    0.013   20.704   20.704 {method 'acquire' of '_multiprocessing.SemLock'
objects}
    20/16    0.006    0.000    0.006    0.000 {method 'acquire' of '_thread.lock' objects}
        1    0.004    0.004   20.690   20.690 pool.py:527(_handle_tasks)
```

Can you help me interpret the results?

Absolutely, Gabriel. Let's walk through the profiler output step by step and break it down.

🧠 How to Read cProfile Output

Each line in the `cProfile` output typically looks like this:

```bash
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
```

Where:

- `ncalls`: Number of calls to the function.
- `tottime`: Time spent in the function **itself** (excluding calls to sub-functions).
- `percall`: `tottime / ncalls`
- `cumtime`: Time spent **in the function and all its sub-functions**.
- `percall`: `cumtime / ncalls`
- `filename:lineno(function)`: Location and name of the function.

🔍 Interpreting Your Results

Here’s a summary of the most relevant lines:

1. 📦 `{built-in method posix.read}`

```pgsql
22/18    20.542    0.934    41.253    2.292 {built-in method posix.read}
```

- **22/18 calls**, ~**20.5s** spent just on `read()`, and **41.2s** including all sub-functions.
- This suggests a lot of **I/O** time — possibly reading large files or inter-process communication (pipes/sockets).
- **This is your main bottleneck**.

---

2. 🔐 `{method 'acquire' of '_multiprocessing.SemLock' objects}`

```pgsql
1    0.013    0.013    20.704    20.704 {method 'acquire' of '_multiprocessing.SemLock' objects}
```

- Called once, took **20.7s** — this is a red flag 🚩.
- It likely means your code is **waiting to acquire a semaphore lock** — maybe one process is blocking others.
- Could indicate:
  - **Process starvation**
  - **Shared memory lock contention**
  - A worker is stuck and **not releasing** its lock.

---

3. 🧪 `pool.py:527(_handle_tasks)`

```
1    0.004    0.004    20.690    20.690 pool.py:527(_handle_tasks)
```

- This is `multiprocessing.Pool`'s internal worker dispatch.
- Its **cumtime (20.7s)** overlaps with the semaphore — it's likely **blocked** waiting for a task to complete.

---

4. 🧮 Your function: `simulate.py:11(generates_portfolios_by_idxs)`

```
1    0.059    0.059    0.210    0.210 simulate.py:11(generates_portfolios_by_idxs)
```

- Only took **0.21s total** — this function isn’t the bottleneck.
- It might be **dispatching work to subprocesses**, which do the actual heavy lifting.

5. 🧬 `numpy.array`

```pgsql
3    0.152    0.051    0.152    0.051 {built-in method numpy.array}
```

- Quick calls to convert data into NumPy arrays. Not a problem here.

🧠 Summary

Observation	Meaning
High time in `posix.read`	Likely due to reading large data or heavy IPC.
Long wait on `_multiprocessing.SemLock.acquire`	Indicates blocking/waiting for a resource — possible deadlock, or long-running process.
`_handle_tasks` blocked	Confirms that a **`Pool`** worker is taking a long time — might be your shared memory operations or a slow child process.
Your actual function is fast	So the slowness isn't in your Python logic but in the **parallel execution layer**.

🛠️ Suggestions

1. **Check if your child processes are accessing large data improperly**:
    - Are they reading large files instead of using shared memory?
    - Is shared memory working as intended?
2. **Profile a single worker separately**:
    - Time how long it takes to do its job **without multiprocessing**.
3. **Try using `multiprocessing.shared_memory` instead of passing large data** through **`Pool`** or **`Queue`** (if you aren't already).
    - This avoids the **`posix.read`** bottleneck during IPC (serialization → pipe → deserialization).
4. **Switch to `Process` instead of `Pool`** if you want tighter control over data flow and memory sharing.
    - **`Pool`** can be convenient but often masks where the real bottlenecks are.

Would you like help writing a version of your worker setup that uses **`shared_memory`** explicitly and avoids data copying via **`Pool`**?

The funny thing is I am already using shared_memory, I don't know exactly what they are reading. Can you provide debugging suggestions for finding this out?

That's a great insight, Gabriel — and you're asking the right question. If you're already using `shared_memory` but still seeing high time in `posix.read`, then the issue is likely:

1. Unintended **serialization of large objects** (maybe through `multiprocessing.Pool`, `Queue`, or `Pipe`).
2. Hidden **data copying** via implicit mechanisms you're not controlling.
3. Some library function that’s doing its own **I/O or IPC under the hood**.

🛠️ Debugging Plan: “What are they reading?”

✅ 1. Add logging on process startup

Use `os.getpid()` + `time.time()` to timestamp each child process launch and track access points:

```python
import os, time

def worker(...):
    print(f"[PID {os.getpid()}] started at {time.time()}")
    ...
```

This lets you correlate activity and spot delays (especially if workers appear idle).

✅ 2. Patch `multiprocessing` serialization methods

If you're using `Pool`, arguments/results are **pickled and sent through pipes** (causing reads). You can log or monkey-patch pickling to detect this:

```python
import pickle

original_dump = pickle.dump
def patched_dump(obj, *args, **kwargs):
    print(f"[DEBUG] Pickling object of type {type(obj)} and size {len(pickle.dumps(obj))}")
    return original_dump(obj, *args, **kwargs)

pickle.dump = patched_dump
```

This helps catch any large object that is accidentally being serialized and sent through the pipe — even if you think you're using shared memory.

✅ 3. Print memory address of shared memory-backed arrays

To confirm your workers are using **the right buffer**, log the `.ctypes.data`:

```python
print(f"[{os.getpid()}] Shared A address: {shared_A.ctypes.data}")
```

If workers are using **copies**, you'll see different addresses.

✅ 4. Use `tracemalloc` to track memory allocations

```python
import tracemalloc
tracemalloc.start()

# In your worker
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')
print("[Top memory allocations]")
for stat in top_stats[:10]:
    print(stat)
```

This lets you spot any **unexpected memory allocation**, e.g., from loading a new array or deserializing data.

✅ 5. Use `strace` to monitor syscalls

This is super powerful when used on a suspicious process:

```bash
strace -p <worker_pid> -e trace=read,open
```

You'll see **what files or pipes are being read**, in real time.

If you see things like:

```bash
read(3, "....", 8192)
read(4, "....", 8192)
```

...you can cross-reference file descriptors with **`/proc/<pid>/fd`** to see *where* they're reading from.

✅ 6. Replace `Pool` with manual `Process` setup (as a test)

Even if temporary, this gives you 100% control of data flow.

```python
from multiprocessing import Process

p = Process(target=worker, args=(shared_name, ...))
p.start()
p.join()
```

This **bypasses all implicit `pickle`-and-pipe behavior** of **`Pool`**, and if the reads disappear, you've found your culprit.

🔍 Most Likely Culprit

Based on your `cProfile` trace:

```bash
20.542    0.934   41.253    2.292 {built-in method posix.read}
20.704   20.704 {method 'acquire' of '_multiprocessing.SemLock' objects}
```

My bet is on:

> `multiprocessing.Pool` **is serializing large arguments or results**, causing huge `posix.read()` calls over internal pipes.

Even if you're **using shared memory**, just passing a large `np.array` as a return value causes `pickle.dumps()` to fire.

✅ Quick Test

Try modifying your `worker` to **not return anything** and write results to another shared memory segment or a `multiprocessing.Array`. If `posix.read` usage drops, the issue is serialization during result collection.

Want to go further?

I can help you:

- Write a lightweight logger that **intercepts all IPC traffic**.
- Create a **SharedMemory wrapper** to ensure all arrays are passed by reference.
- Optimize a custom `ProcessPoolExecutor` where no pickling happens.

Let me know how deep you want to go 👇

I will try to replace using Pool with Process. This is the setup right now:

```python
    n_processes = mp.cpu_count()
    batch_size = len(portfolios_idxs) // n_processes + 1
    idxs_batches = [
        portfolios_idxs[i:i + batch_size] for i in range(0, len(portfolios_idxs), batch_size)
    ]

    shm = SharedMemory(create=True, size=daily_returns_matrix.nbytes)
    shm_returns = np.ndarray(
        daily_returns_matrix.shape,
        dtype=daily_returns_matrix.dtype,
        buffer=shm.buf
    )
    np.copyto(shm_returns, daily_returns_matrix)

    shm_info = (
        shm.name,
        daily_returns_matrix.shape,
        daily_returns_matrix.dtype
    )
    with mp.Pool(processes=n_processes) as pool:
        results = pool.starmap(
            maximize_sharpe_aux,
            [
                (
                    assets_per_portfolio,
                    max_weight_per_asset,
                    num_simulated_weights,
                    batch,
                    shm_info
                )
                for batch in idxs_batches
            ]
        )
```

```
python
def maximize_sharpe_aux(
        assets_per_portfolio: int,
        max_weight_per_asset: float,
        num_simulated_weights: int,
        tickers_idxs: np.ndarray,
        shm_info: np.ndarray
) -> Result[tuple[float, np.ndarray, np.ndarray], str]:
    """
    Auxiliary function to maximize the Sharpe ratio for a given set of assets.

    Args:
        assets_per_portfolio (int): Number of assets in each portfolio.
        max_weight_per_asset (float): Maximum weight allowed for each asset.
        num_simulated_weights (int): Number of weight combinations to generate.
        tickers_idxs (np.ndarray): Indices of the assets in the portfolio.
        daily_returns_matrix (np.ndarray): Daily returns matrix.

    Returns:
        Result[tuple[float, np.ndarray], str]: A Result object containing the maximum Sharpe ratio
                                                and the corresponding weights on success, or an error
                                                message on failure.
    """
    max_sharpe = float('-inf')
    optimal_weights = np.array([])
    optimal_tickers_idxs = np.array([])
    for ticker_idx in tickers_idxs:
        weights = generate_weights(
            assets_per_portfolio, max_weight_per_asset, num_simulated_weights
        )
        if isinstance(weights, Err):
            return weights
        sharpe_result = maximize_sharpe(
            ticker_idx,
            weights.value,
            shm_info
        )
        if isinstance(sharpe_result, Err):
            return sharpe_result
        sharpe, weights = sharpe_result.value
        if sharpe > max_sharpe:
            max_sharpe = sharpe
            optimal_weights = weights
            optimal_tickers_idxs = ticker_idx

    return Ok((max_sharpe, optimal_weights, optimal_tickers_idxs))
```

Could you introduce me to using Process and spot the points of code I need to change, without providing the complete new code.

Absolutely, Gabriel — you're on the right track. Replacing `multiprocessing.Pool` with `multiprocessing.Process` gives you full control over inter-process communication and avoids implicit pickling (and the `posix.read` problem). Below, I'll walk you through **what needs to change and why**, without giving you the final code.

🔧 1. Change from `Pool.starmap` to Manual Process Launch

What to change:

- Instead of using `Pool.starmap`, you'll:
  - Create a `Process` per batch.
  - Start each process manually.
  - Use a communication channel to collect results (e.g. `multiprocessing.Queue` or `Pipe`).

Why:

- `Pool` auto-serializes arguments/results — even your small `Result` wrappers.
- Manual `Process` setup avoids implicit pickling and gives you precise control.

📥 2. Set Up a Queue or Pipe to Collect Results

What to change:

- Add a `Queue` and pass it as an extra argument to each `Process`.

Why:

- Each child will write its result to the queue instead of returning it directly.
- Avoids `multiprocessing.Pool`'s implicit I/O (which uses `posix.read` under the hood).

🧠 3. Ensure `shm_info` is read-only and passed correctly

No change needed **if you're just reading from shared memory**, but:

What to watch:

- Avoid passing the entire `shm_returns` object — pass only `shm.name`, `shape`, and `dtype` (`shm_info`).
- In each process, **reconstruct the NumPy view** manually using `np.ndarray(..., buffer=shm.buf)`.

📛 4. Don't Forget to `join()` All Processes

What to change:

- After starting all processes, use `p.join()` to wait for them to finish.
- Collect all results from the queue once all processes are joined.

🚫 5. Avoid Global Variables and Closures

What to check:

- Make sure **everything each process needs is passed explicitly** (no closure over outer variables).
- This ensures portability and minimizes bugs when moving to `spawn`-based multiprocessing (default on macOS/Windows).

✂️ 6. Remove the `with mp.Pool()` block

Self-explanatory — you'll be managing the lifecycle of your workers manually now.

Optional: 🌱 Future-Proofing

If you want to improve maintainability later, you can encapsulate each child worker's setup in a helper function like:

```python
def start_worker(..., queue):
    p = mp.Process(target=worker_function, args=(..., queue))
    p.start()
    return p
```

🧩 Final Hint

In your `maximize_sharpe_aux` function, you'll need to:

- Add a `queue` argument.
- Replace the `return` statement with `queue.put(result)`.

Would you like a skeleton snippet that wires this up just for one process, so you can extrapolate from it?

Yes, provide a skeleton snippet

Great, Gabriel. Here's a minimal **skeleton snippet** showing how to replace your `Pool.starmap` logic with `Process`, `Queue`, and manual shared memory usage — without implementing your full logic.

🧱 Skeleton: Manual Multiprocessing with SharedMemory and Queue

```python
import multiprocessing as mp
from multiprocessing.shared_memory import SharedMemory
import numpy as np

def worker_function(batch, shm_info, other_args..., queue):
    # Reconstruct the shared NumPy array
    shm_name, shape, dtype = shm_info
    shm = SharedMemory(name=shm_name)
    shared_array = np.ndarray(shape, dtype=dtype, buffer=shm.buf)

    # Do your computation (e.g. call maximize_sharpe)
    result = compute_from_batch(batch, shared_array, other_args...)

    # Send result back to parent process
    queue.put(result)

    # Optional: clean up the view (not the shared memory itself)
    shm.close()
```

🚀 Launching Processes

```python
if __name__ == '__main__':
    # Setup shared memory
    shm = SharedMemory(create=True, size=daily_returns_matrix.nbytes)
    shm_array = np.ndarray(daily_returns_matrix.shape, dtype=daily_returns_matrix.dtype,
buffer=shm.buf)
    np.copyto(shm_array, daily_returns_matrix)
    shm_info = (shm.name, daily_returns_matrix.shape, daily_returns_matrix.dtype)

    # Prepare batches
    n_processes = mp.cpu_count()
    batch_size = len(portfolios_idxs) // n_processes + 1
    idxs_batches = [portfolios_idxs[i:i + batch_size] for i in range(0, len(portfolios_idxs),
batch_size)]

    # Setup queue and process list
    result_queue = mp.Queue()
    processes = []

    for batch in idxs_batches:
        p = mp.Process(
            target=worker_function,
            args=(batch, shm_info, ..., result_queue)
        )
        p.start()
        processes.append(p)

    # Collect results
    results = [result_queue.get() for _ in processes]

    # Wait for all to finish
    for p in processes:
        p.join()

    # Cleanup shared memory
    shm.close()
    shm.unlink()
```

🧠 Notes

- Replace `...` with your additional arguments (`assets_per_portfolio`, etc.).
- You can return `Ok(...)` or `Err(...)` just like before — no change in your `Result` type logic.
- Make sure your `worker_function` handles **only one batch** — similar to what your current loop in `maximize_sharpe_aux` does.

Let me know if you want a variant that works with `concurrent.futures.ProcessPoolExecutor` instead, or if you'd like to offload large results via shared memory as well.

Can you explain to me the workflow of using mp.Queue and mp.Process?

Absolutely, Gabriel. Let's break it down.

When you use `multiprocessing.Process` with a `multiprocessing.Queue`, you're building your **own manual worker system**. This gives you **full control** over how processes communicate, how they start/stop, and what memory is shared. Here's a conceptual and practical workflow:

🔄 The Workflow: `mp.Process` + `mp.Queue`

🧠 1. Main process sets everything up

- Creates shared memory (if needed).
- Splits the workload (e.g., into batches).
- Creates a `Queue` for collecting results.
- Spawns multiple `Process` objects, each running a worker function.
- Each process receives a **slice of work**, the `Queue`, and any shared memory info.

⚙️ 2. Each process runs independently

- It starts executing your `target` function (`worker_function`).
- Inside the worker:
  - It **reconstructs any shared memory views**.
  - It does the work (e.g., computes on a chunk of data).
  - It **puts the result on the `Queue`** to return it to the main process.
- The process then **exits cleanly**.

📦 3. Main process collects results

- Waits for all results via `queue.get()` — one for each process.
- Calls `join()` on each process to ensure they finish cleanly.

- Optionally cleans up shared memory.

🧊 Visual Metaphor

Think of this setup like a **kitchen**:

- 👨‍🍳 The **main process** is the head chef, preparing ingredients (data, memory).
- 👨‍🍳 Each **worker process** is a line cook:
  - Takes a set of ingredients (batch + shared memory view).
  - Cooks something (computes a result).
  - Puts the dish on the **serving window** (`Queue`).
- The head chef collects dishes and shuts down the kitchen.

🔁 Summary Table

Component	Role	Key Methods
`mp.Process`	Creates a new worker process	`start()`, `join()`
`mp.Queue`	Safe, cross-process communication channel	`put()`, `get()`
`SharedMemory` (opt.)	Efficient way to share large read-only data	`SharedMemory()`, `np.ndarray(..., buffer=shm.buf)`

🧪 Minimal Example

```python
import multiprocessing as mp

def worker(x, queue):
    queue.put(x**2)

if __name__ == "__main__":
    queue = mp.Queue()
    processes = []

    for i in range(4):
        p = mp.Process(target=worker, args=(i, queue))
        p.start()
        processes.append(p)

    results = [queue.get() for _ in processes]

    for p in processes:
        p.join()

    print(results)  # [0, 1, 4, 9] (order not guaranteed!)
```

🧭 When to Use This

You should use **`mp.Process`** + **`Queue`** when:

- You want full control over how workers are launched.
- You need to **avoid** the automatic pickling and I/O that happens with **`Pool`**.
- You're using **shared memory** or **large data**, and want to pass references instead of copies.
- You want precise logging/debugging.

Let me know if you'd like a diagram or example that includes error handling or timeout logic!

I need help understanding why the matrix multiplications are taking so long when using multiprocessing.

Here is the code I made to help me understand where the problem lies:

```
python
def maximize_sharpe_aux(  # noqa: PLR0914
        assets_per_portfolio: int,
        max_weight_per_asset: float,
        num_simulated_weights: int,
        tickers_idxs: np.ndarray,
        shm_info: np.ndarray
) -> Result[tuple[float, np.ndarray, np.ndarray], str]:
    """
    Auxiliary function to maximize the Sharpe ratio for a given set of assets.

    Args:
        assets_per_portfolio (int): Number of assets in each portfolio.
        max_weight_per_asset (float): Maximum weight allowed for each asset.
        num_simulated_weights (int): Number of weight combinations to generate.
        tickers_idxs (np.ndarray): Indices of the assets in the portfolio.
        daily_returns_matrix (np.ndarray): Daily returns matrix.

    Returns:
        Result[tuple[float, np.ndarray], str]: A Result object containing the maximum Sharpe
ratio
                                                and the corresponding weights on success, or an
error
                                                message on failure.
    """
    start = time.time()
    times_1 = []
    times_2 = []

    max_sharpe = float('-inf')
    optimal_weights = np.array([])
    optimal_tickers_idxs = np.array([])

    for ticker_idx in tickers_idxs:
        weights = generate_weights(
            assets_per_portfolio, max_weight_per_asset, num_simulated_weights
        )
        if isinstance(weights, Err):
            return weights

        weights = weights.value
        Rf_yearly = 0.05

        shm_name, shape, dtype = shm_info
        shm = SharedMemory(name=shm_name)
        daily_returns_matrix = np.ndarray(
            shape,
            dtype=dtype,
            buffer=shm.buf
        )
        ANNUALIZATION_FACTOR = 252

        t1 = time.time()

        R_daily = daily_returns_matrix[ticker_idx].T
        Rp_daily = R_daily @ weights.T
        Rp_yearly = np.mean(Rp_daily, axis=0) * ANNUALIZATION_FACTOR
        ER_yearly = Rp_yearly - Rf_yearly

        times_1.append(time.time() - t1)
        t2 = time.time()

        cov_matrix = np.cov(R_daily, rowvar=False)
        Var_daily = np.diag(weights @ cov_matrix @ weights.T)
        Vol_yearly = np.sqrt(Var_daily * ANNUALIZATION_FACTOR)

        times_2.append(time.time() - t2)

        SR = ER_yearly / Vol_yearly

        optimal_idx = np.argmax(SR)
        s, w = SR[optimal_idx], weights[optimal_idx]

        if s > max_sharpe:
            max_sharpe = s
            optimal_weights = w
            optimal_tickers_idxs = ticker_idx

        shm.close()

    print(
        f'Process {mp.current_process().name} completed at {time.strftime("%H:%M:%S")}\n'
        f'\tTotal time: {time.time() - start:.2f} seconds\n'
        f'\tAvg t1: {np.mean(times_1):.2f} seconds\n'
        f'\tAvg t2: {np.mean(times_2):.2f} seconds\n'
    )
    return Ok((max_sharpe, optimal_weights, optimal_tickers_idxs))
```

Here is the output for 8 processes:

```bash
Process ForkPoolWorker-8 completed at 23:37:36
        Total time: 21.03 seconds
        Avg t1: 0.09 seconds
        Avg t2: 0.09 seconds

Process ForkPoolWorker-4 completed at 23:37:38
        Total time: 23.07 seconds
        Avg t1: 0.09 seconds
        Avg t2: 0.09 seconds

Process ForkPoolWorker-6 completed at 23:37:38
        Total time: 23.26 seconds
        Avg t1: 0.09 seconds
        Avg t2: 0.09 seconds

Process ForkPoolWorker-1 completed at 23:37:38
        Total time: 23.32 seconds
        Avg t1: 0.09 seconds
        Avg t2: 0.09 seconds

Process ForkPoolWorker-5 completed at 23:37:38
        Total time: 23.44 seconds
        Avg t1: 0.09 seconds
        Avg t2: 0.09 seconds

Process ForkPoolWorker-7 completed at 23:37:38
        Total time: 23.50 seconds
        Avg t1: 0.09 seconds
        Avg t2: 0.09 seconds
```

```
Process ForkPoolWorker-3 completed at 23:37:38
        Total time: 23.56 seconds
        Avg t1: 0.09 seconds
        Avg t2: 0.09 seconds

Process ForkPoolWorker-2 completed at 23:37:38
        Total time: 23.59 seconds
        Avg t1: 0.09 seconds
        Avg t2: 0.09 seconds
```

Here is the output for 1 process:

```
bash
Process ForkPoolWorker-1 completed at 23:39:45
        Total time: 2.83 seconds
        Avg t1: 0.00 seconds
        Avg t2: 0.00 seconds
```

Thanks for sharing all the details, Gabriel — your experiment is very well instrumented. The **core issue** you're seeing is that using `multiprocessing` actually **slows things down** in your case, even though the actual matrix math (`t1` and `t2`) is fast. Let's break this down and explain **why 1 process takes ~2.8s**, but **8 processes take ~23s**.

🚨 Root Cause: Shared Memory Contention or Overhead

You're using a shared `numpy.ndarray` (backed by `SharedMemory`) across processes. While **reads are technically safe**, when multiple processes access the same memory block — especially with **strided slicing and transposition** (`.T`) — **CPU cache contention and memory bandwidth saturation** can become significant.

🔍 Deconstructing Your Timing

✅ `t1` and `t2`:

Each takes ~0.09s per ticker batch per process. This seems **stable and efficient**, so **the matrix math itself isn't the problem**.

❌ Total time:

Every process is taking ~23s. This is **far more than the actual matrix ops** and implies:

- **I/O bottleneck** reading from shared memory
- **Startup overhead** (`fork`, `shm` reconstruction, etc.)
- **Global memory bus saturation** from multiple readers simultaneously accessing large, strided slices
- **GIL contention** on setup code before hitting NumPy (even though NumPy itself releases the GIL)

🧪 Interpretation

Even though reads are supposed to be safe, **SharedMemory does not bypass the fact that all 8 processes are hitting the same physical memory** — and `np.ndarray.T` creates **non-contiguous views**, which stress the CPU's memory fetch unit.

You're likely also being hit by:

- **False sharing**: processors reading adjacent memory lines interfere with each other's caches
- **OS-level page faults or TLB misses** from fast multiprocess access to the same memory pages

---

✅ Why 1 Process Is Faster

With 1 process:

- There's **no overhead** from IPC, process creation, or shared memory reconstruction
- **Cache locality is preserved**, as data stays in the L1/L2 cache once loaded
- `np.ndarray.T` is fine because there's no memory contention

🧠 Recommendations

🟡 Short-Term

1. **Don't use `.T`**: Try to pre-transpose `daily_returns_matrix` **once** before writing to shared memory, to avoid doing it repeatedly in each worker with strided access.

   ```python
   daily_returns_matrix = daily_returns_matrix.transpose(0, 1)
   ```

2. **Benchmark without shared memory**:
   Temporarily copy slices instead of reading from `SharedMemory`, just to compare speed.

   ```python
   R_daily = np.copy(daily_returns_matrix[ticker_idx].T)
   ```

   If this is **faster**, it's a memory contention problem for sure.

3. **Try using `Process` manually** instead of `Pool`:
   The `Pool` can introduce overhead from internal task coordination (especially in forked subprocesses).

🟢 Long-Term

If the matrix fits in RAM:

- ✅ **Use `joblib.Parallel` with `shared_memory` or memory-mapped arrays (`np.memmap`)** — it handles slicing efficiently.
- ✅ Consider **splitting the input into chunks** and distributing those as standalone arrays (per-process copies, or better yet: chunks of memory-mapped files).
- ✅ Try **thread-based parallelism with NumPy multithreading** (e.g., using OpenBLAS/NumExpr) — NumPy is already very fast in single process due to multithreaded BLAS.

📈 Debugging Tips

You can confirm the shared memory contention hypothesis by:

- Timing how long `SharedMemory(...)` and `np.ndarray(...)` take to reconstruct in the worker.
- Checking CPU and memory utilization (`htop` or `top`) during execution.
- Using `numa` or `taskset` to pin processes to cores (advanced, but reveals memory bus issues).

Would you like help with rewriting your shared array to avoid `.T`, or benchmarking with `memmap` or process-local slices?

I tried sharing daily returns already transposed, but it didn't change the performance.
I did some more profiling and this is what I found out:
The two most expensive operations are:

- `Rp_daily = R_daily @ weights.T`
- `Var_daily = np.diag(weights @ cov_matrix @ weights.T)`

Why are those the two most expensive operations? Is it the `@` operator?

Excellent debugging, Gabriel — your profiling results point to exactly where the bottleneck is: the **two matrix multiplications** using the `@` operator.

Let's walk through what's happening computationally and **why these specific lines are expensive**, even when memory sharing isn't the issue anymore.

🔬 1. `Rp_daily = R_daily @ weights.T`

🧮 What's happening here?

- **`R_daily`**: shape `(num_days, num_assets)`
- **`weights.T`**: shape `(num_assets, num_simulations)`
- Result: **`Rp_daily`**: shape `(num_days, num_simulations)`

This is a **dense matrix-matrix multiplication**, typically implemented using BLAS (Basic Linear Algebra Subprograms), but it is **compute-heavy**:

Time complexity:
`O(num_days * num_assets * num_simulations)`

If:

- `num_days = 252`
- `num_assets = 10`
- `num_simulations = 10_000`

You're doing over **25 million** multiplications for *each portfolio batch*. Multiply that by how many ticker_idx groups you're processing and the numbers add up fast.

🔬 2. `Var_daily = np.diag(weights @ cov_matrix @ weights.T)`

🧮 What's happening here?

This is **matrix triple-product** of the form:

```yaml
weights         : (num_simulations, num_assets)
cov_matrix      : (num_assets, num_assets)
weights.T       : (num_assets, num_simulations)
```

- `weights @ cov_matrix`: shape → `(num_simulations, num_assets)`
- Resulting product → shape `(num_simulations, num_simulations)`
- Then you take `np.diag(...)` to get just the diagonal (i.e., the variance of each simulated portfolio)

Time complexity:
`O(num_simulations^2 * num_assets)`

So again, with `num_simulations = 10_000`, you're creating a **100 million element** intermediate result just to pull its diagonal.

🧨 So yes — `@` is expensive!

It's not the operator per se — it's the **scale** and the fact that:

- You're multiplying large matrices
- You're doing this **inside loops**, over and over again
- And NumPy allocates new arrays for each operation — which stresses memory and cache

---

🛠️ Recommendations

✅ Optimize the Variance Computation

Instead of:

```
python

Var_daily = np.diag(weights @ cov_matrix @ weights.T)
```

Use **row-wise vectorized variance**:

```
python

Var_daily = np.einsum('ij,jk,ik->i', weights, cov_matrix, weights)
```

This computes the **quadratic form** $w_i^T \Sigma w_i$ for each row w_i in **`weights`** directly — without forming a giant matrix first. It's often **10× faster** and uses far less memory.

✅ Optimize Expected Return

You can also improve the first multiplication slightly with **`einsum`**, although this isn't as dramatic:

```
python

Rp_daily = np.einsum('dt,st->ds', R_daily, weights.T)
```

But note: the bottleneck here is *less* about Python syntax and more about **data scale**. That's why parallelization needs to be structured to **minimize duplication** and reuse arrays efficiently.

✅ Other ideas

- **Batch weights:** Process **`weights`** in chunks if **`num_simulations`** is very large.
- **Use lower precision**: Try **`np.float32`** instead of **`np.float64`** — this can halve memory bandwidth and speed up operations on most CPUs.
- **Enable MKL/OpenBLAS threading:** Make sure NumPy is using optimized BLAS and set appropriate environment variables (e.g., **`OMP_NUM_THREADS`**).
- **Use GPU for these calculations** (e.g., with CuPy or PyTorch) if your matrices are huge and highly parallelizable.

Would you like a rewritten version of the Sharpe maximization loop using these optimizations (**`einsum`**, batching, float32)?

Ok, the Var_daily change reduced the average calculation time in almost 100%!
I still have a problem with the first dot operation, though. How can we tackle this effectively, and why is it worse when we have multiple processes doing it?

That's a **huge win** on the variance side — well done!

Now let's tackle the first matrix multiplication (`Rp_daily = R_daily @ weights.T`) and **why it's a bottleneck in multiprocessing**:

🧠 Why `R_daily @ weights.T` is still slow

✅ What's happening:

You're multiplying:

- `R_daily`: shape `(252, N_assets)` — a slice of the shared memory matrix
- `weights.T`: shape `(N_assets, N_simulations)`

Result is:

- `Rp_daily`: shape `(252, N_simulations)`

If `N_simulations` is large (e.g., 10,000), this is a **big matrix**: `252 x 10,000 = 2.5 million` entries per portfolio.

This dot product is:

- CPU-heavy
- Memory-bandwidth-heavy (streaming lots of data)
- Non-trivial to parallelize at small scales due to cache contention

---

🔥 Why it gets *worse* with multiprocessing

1. Shared memory ≠ shared cache

Each process has:

- Separate Python interpreter
- Separate memory space
- Separate CPU core (usually)

Although you've shared the data with `shared_memory`, the actual matrix slices (`R_daily`) must be read into each process's **L1/L2/L3 CPU cache**. If multiple cores try to read from the same RAM at once, it causes **cache line contention and memory bandwidth pressure**.

> ⚠️ Shared memory gives data consistency, but not shared *cache locality*.
> This makes NumPy's dot product slower in multiple processes.

---

2. BLAS thread contention

Most NumPy installations (especially with MKL/OpenBLAS) **use multithreading internally** for large matrix operations.

So when you spawn **8 Python processes**, each one might also spawn **8 internal BLAS threads**, leading to **64 competing threads** fighting over CPU cache and memory bandwidth — totally killing performance.

🛠️ Solutions

✅ 1. Limit NumPy threads inside each process

This is critical.

Add this **at the top of your script**, *before* any NumPy import or multiprocessing:

```python
import os
os.environ["OMP_NUM_THREADS"] = "1"
os.environ["OPENBLAS_NUM_THREADS"] = "1"
os.environ["MKL_NUM_THREADS"] = "1"
os.environ["VECLIB_MAXIMUM_THREADS"] = "1"
os.environ["NUMEXPR_NUM_THREADS"] = "1"
```

Why: You want **one process = one core = one thread**, no overlap.

✅ 2. Precompute dot products for all tickers if possible

If you're computing `R_daily @ weights.T` per ticker group but using the **same weights** every time, consider:

- Precomputing `weights.T` just once
- Precomputing `R @ weights.T` for all assets once
- Then slicing rows based on `ticker_idx` groups

If feasible, this converts **many small matrix ops** → **one big matrix op**, which is much faster on BLAS.

✅ 3. Use lower precision

Cast your arrays to `float32` before the dot product:

```python
R_daily = R_daily.astype(np.float32)
weights_T = weights.T.astype(np.float32)
Rp_daily = R_daily @ weights_T
```

Dot products are often memory-bound. Using half the memory doubles the effective bandwidth and speeds up computations, especially on CPUs.

✅ 4. Switch to `einsum` or batched dot

Try using `np.einsum` if you want to squeeze more control:

```python
Rp_daily = np.einsum('di,is->ds', R_daily, weights.T)
```

This lets NumPy use a more optimized contraction path under the hood.

✅ 5. Process fewer simulations per worker

You can split `weights` across multiple processes instead of portfolios:

- Each worker gets the full `R_daily`, but only a **subset of `weights`**
- This reduces `Rp_daily` size and CPU load per process

This might balance load better, especially if simulations are the most expensive part.

🧪 Summary

Problem	Solution
CPU/memory contention	Set `OMP_NUM_THREADS=1`, avoid nested threads
Large matrix op per process	Use float32, chunk weights, or precompute
Shared memory not caching well	Consider batching, or process in larger blocks
BLAS overload	Avoid thread explosion by controlling env vars

Let me know your current `weights` shape and whether you reuse them across tickers — I can suggest a restructuring strategy based on that.

Ok, could you do an ELI5 explaining the issues we tackled together?

Absolutely! Here's an **ELI5-style summary** of the performance issues we tackled together and what they mean in plain terms:

🔧 The Problem

You built a Python program that simulates a bunch of financial portfolios by doing lots of **math on big matrices** (like giant spreadsheets). To make it faster, you tried using **multiple workers at once** — like asking 8 people to help with the calculations.

But instead of going 8x faster... it got **slower**.

🧠 What We Found Out

🧵 1. NumPy secretly spins up its own helpers

When NumPy sees a big matrix operation like `matrix A @ matrix B`, it says:

> "Cool! Let me call my super-fast helpers from BLAS or MKL to handle this."

But guess what? Each NumPy helper **also starts its own mini-team of workers**.
So if you ask 8 people to help (with multiprocessing), and each one brings 8 friends... now you've got **64 people** bumping into each other in the same kitchen.

💥 Result: **Too many cooks in the kitchen**, and performance drops.

🧠 2. Shared memory isn't shared *nicely*

You used shared memory (👍 good choice!) so all your worker processes could read the same big array without copying it.

But computers have a brain (CPU) and a memory (RAM), and between them is a super-fast backpack (cache).

When all your workers grab the same shared array from memory at once, it's like:

> "Hey RAM, give me the same book!"
> "Wait your turn, I can only serve one at a time..."

💥 Result: **Cache fights** and **memory traffic jams** slow everything down.

🧠 3. Your two most expensive operations were:

- Matrix multiplication: `R_daily @ weights.T`
- Variance computation: `weights @ cov_matrix @ weights.T`

These are like asking, "What would happen if I tried all these combinations?" — very powerful, but **very math-heavy**.

🛠️ What We Did to Fix It

1. ✅ **Stopped the thread explosion**
   Told NumPy helpers: "You each get **one worker only**, no friends."
   (`OMP_NUM_THREADS = 1` and friends)
2. ✅ **Improved memory use**
   Converted arrays to `float32` → smaller = faster!
3. ✅ **Rewrote one of the expensive operations**
   Instead of doing fancy math the long way, we used a **simpler trick** that gives the same answer but is 10x faster.
4. ✅ **Found the root cause of slowdown with profiling**
   You timed things like a scientist with a stopwatch 🕒 and saw:

   > "Wait — when I run just **one** worker, it's super fast... but **many workers** make it slow!"

So we knew it wasn't your math — it was the way **your helpers were fighting each other**.

🧠 What This All Means

- **Parallel computing is tricky** — sometimes more workers = worse performance
- You have to watch out for **hidden work** (like NumPy helpers doing their own thing)
- **Memory sharing ≠ performance sharing** — just because they can read the same thing doesn't mean it's fast
- Using a profiler is like putting a GPS tracker on your code — you can see where it's stuck and why

---

Let me know if you want an ELI5 for how shared memory works or what `@` (matrix multiplication) really does — we can build intuition around that too.

Can I wrap the Shared Memory in a `with` clause to avoid memory leakage?

Great question, Gabriel — and you're right to be thinking about **resource cleanup**.

Here's the deal:

✅ Can You Use `with` for `SharedMemory`?

No, the built-in `multiprocessing.shared_memory.SharedMemory` object **does not implement the context manager protocol**, so you can't use it like this:

```python
with SharedMemory(...) as shm:  # ❌ this raises AttributeError
    ...
```

Why? Because `SharedMemory` doesn't define `__enter__` and `__exit__` methods, which are required for `with`.

🧽 So How Do You Avoid Memory Leaks?

You need to **manually** call:

- `shm.close()` — closes the *view* (in the current process)
- `shm.unlink()` — **removes the shared memory block from the system**

🧠 When to call `close()` and `unlink()`:

In which process?	Call `close()`	Call `unlink()`
✅ Main process	✅ Yes	✅ Yes
✅ Worker processes	✅ Yes	❌ No

🔥 If you `unlink()` too early, other processes won't find the shared memory anymore.

✅ Best Practice: Manual `try/finally` Block

Here's how to safely manage shared memory to avoid leaks:

```python
from multiprocessing.shared_memory import SharedMemory
import numpy as np

try:
    shm = SharedMemory(create=True, size=...)
    shm_array = np.ndarray(..., buffer=shm.buf)
    # Do stuff with shm_array here

finally:
    shm.close()   # Always close!
    shm.unlink()  # Only in the main process
```

🛡️ Optional: Make Your Own Context Manager

If you want a clean `with` experience, you can **wrap SharedMemory** in your own context manager:

```python
from multiprocessing.shared_memory import SharedMemory

class SafeSharedMemory:
    def __init__(self, *args, unlink=True, **kwargs):
        self.shm = SharedMemory(*args, **kwargs)
        self.unlink = unlink

    def __enter__(self):
        return self.shm

    def __exit__(self, exc_type, exc_value, traceback):
        self.shm.close()
        if self.unlink:
            self.shm.unlink()

# Usage
with SafeSharedMemory(create=True, size=...) as shm:
    arr = np.ndarray(..., buffer=shm.buf)
    # Use arr
```

This gives you leak protection + readability.

Let me know if you want this wrapped in a reusable helper for both main and child processes.