Practical Experience with Multiprocessing

Pedro Leite - 201906697

1. Is your parallel program slower than the sequential? Why?

r = 10 m = 5 n = 10
Sequential - 0.05 seconds
Parallel - 0.18 seconds

The parallel program is slower than the sequential, since the data provided is very small. The process creation in itself is already 6 times bigger than the sequential program.

2. What is the difference between Option #1 and Option #2? In other words, what is the difference between “apply” and “map”? Which one is slower? Why?

r = 10 m = 5 n = 10
Apply - 0.18 seconds
Map - 0.15 seconds

Using "Map" (option #1) instead of "Apply" (option #2), makes the program faster. In the "Apply" each process in the pool is assigned one task at a time, it waits for that task to complete before moving on to the next one. With map, the pool automatically distributes the rows of data among the worker processes in the pool, each process independently executes the function on its assigned data.

3. Try increasing the dimension of your data. Do you see any improvement in performance or not? Why?

r = 10 m = 200 n = 2000
Apply - 5.5 seconds
Map - 3.5 seconds

It shows again better results using "Map", where this time the difference between both is much bigger.

4. Write a summary about these different forms of running parallel code. In which situations would you use each one of those alternatives?

r = 10 m = 5 n = 10

Option 1: Using apply_async with multiprocessing.Pool, took 0.04 seconds.

This option, for every row in the data, asynchronously applies the function, this means that the function calls are concurrent across both CPUs, so every function call can be executed in parallel. Ideally for this problem, since we can divide our problem in small tasks, in this case the data matrix in rows.

Option 2: Directly invoking multiprocessing.Process, took 0.01 seconds with errors

This option, creates 2 separate processes that can be managed independently, enabling concurrency and parallelism as option 1.But is more suited to tasks where is not easy to separate the problem in smaller tasks, which is not our case. It might be faster, but it requires careful handling of synchronization and error management to ensure reliable execution.

Option 3: Batching tasks with multiprocessing.Process, took 0.22 seconds with errors.

This option divides the workload into batches and assigns each batch to a separate process. Despite attempts to parallelize tasks, the execution time is longer, possibly due to inefficiencies in task partitioning or resource utilization. Errors encountered during execution indicate issues with task assignment or result handling, affecting the reliability of the approach. It may be suitable for scenarios where the workload can be evenly divided into batches, but requires careful optimization to achieve optimal performance.

Option 4: Hybrid approach with task partitioning, took 0.06 seconds.

This approach combines batching tasks with dynamic task partitioning based on CPU count, aiming to optimize resource utilization. By dynamically adjusting the workload based on CPU count, it maximizes parallelism while avoiding excessive resource consumption. Its not the best option for this problem, because is more suited for scenarios with varying workload sizes.

5. Modify these scripts to run using multiple threads instead of processes (you will need to use another module: “threading”). Compare their performance when varying the matrix size.

Option 1: Uses the ThreadPool class from the multiprocessing.dummy module to create multiple threads.
Results 1 (r = 10, m = 1000, n = 10000): Using cpu_count threads took 0.002 seconds.
Results 2 (r = 10, m = 5, n = 30) : Using cpu_count threads took 0.90 seconds.

Option 2: Divides the workload manually by creating multiple threads.
Results 1 (r = 10, m = 1000, n = 10000): Using cpu_count threads took 2.6 seconds.
Results 2 (r = 10, m = 5, n = 30): Using cpu_count threads took 0.003 seconds.

For larger matrices, option 1 performs much better than option 2. Due to the fact that in option 1, for larger matrices, the automatic task distribution provided by ThreadPool ensures that the workload is evenly distributed among threads, maximizing parallelism and overall efficiency. While for smaller matrices, option 2 performs much better than option 1, likely due to reduced overhead associated with thread management.

6. For what kind of tasks should you use processes and when should you use threads?

Processes:
- Processes are preferable for tasks that are computationally intensive and spend most of their time performing CPU computations.
- Processes provide independent memory space for each instance, ensuring that one process cannot affect the memory of another process. This is beneficial for tasks requiring isolated execution environments.
- Processes can leverage multiple CPU cores efficiently, making them suitable for parallel execution of computationally intensive tasks.
- Processes are preferable when dealing with shared resources that need to be protected from concurrent access, as they offer separate memory spaces and avoid the complexities of shared memory synchronization.

Threads:
- Threads are suitable for I/O-bound tasks where the majority of time is spent waiting for I/O operations to complete.

- Threads have lower overhead compared to processes, making them more lightweight and efficient for tasks with frequent context switching or lightweight computations.
- Threads within the same process share memory space, allowing them to access shared data structures.
- Threads are useful when the hardware has limited parallelism as they enable concurrent execution within a single CPU core.