**ASSIGNMENT ONE: PARALLEL SUPERMARKET CONCEPT**

by

Armando Eliseu Phacule

Antonio Andre

Luan Pachiesso

to

Computer science

**Parallel Computing**

**Lecture: Lars Albino Lemos**

University of Saint Thomas of Mozambique

Maputo, 06 June, 2024

**Supermarket Checkout Simulation**

**Introduction**

These assignments aims to simulate the process of checking out in a supermarket from three different checkout lanes, i.e. self-checkout, traditional cashier, and a 15-item express lane and to illustrate the benefits, cost and limitation of implementing parallel computing in this manner.

**Assignment 1**

The first assignment implements two versions of the supermarket simulation for both analysis and to support the claims of having achieved the implementation of parallel computing in this regard. It is worth noting that these simulations were conducted on a quad- core machine with each process taking one second to complete for emphasis, more on that later on.

**Parallel Computing Approach**

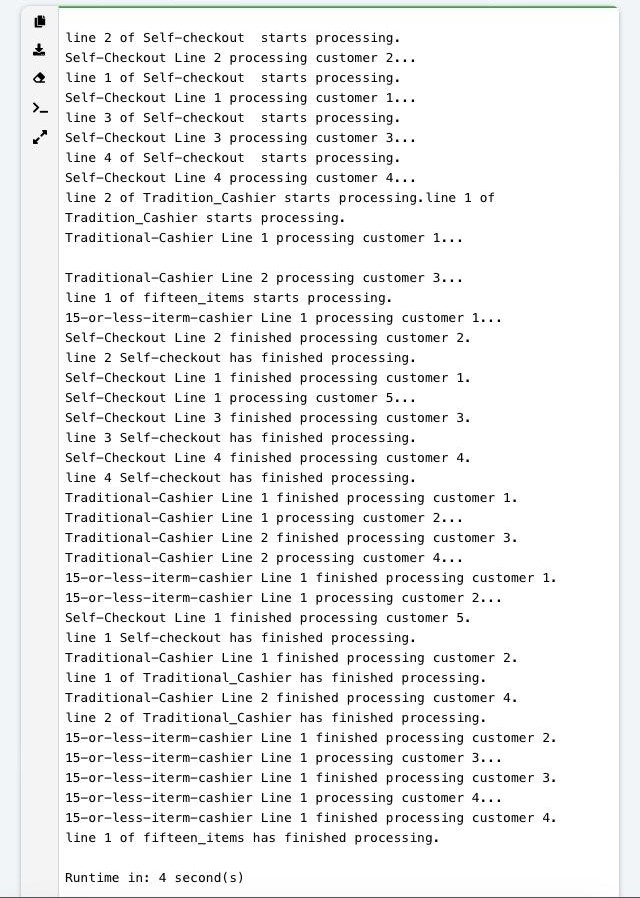
Our parallel computing leverages Python's *multiprocessing* module to create separate processes for each checkout process lane. These processes run concurrently, simulating parallel execution and allowing for efficient utilization of system resources.

**Implementation Details**

The parallel version consists of the following key components:

1. **Checkout Lane Functions**: The code defines three main functions to simulate the processing of customers in each checkout lane:
   * self\_checkout\_cashier(line\_id, customers): Simulates the self-checkout lane
   * traditional\_cashier(line\_id, customers): Simulates the traditional cashier lane
   * fifteen\_item\_cashier(line\_id, customers): Simulates the 15-item express checkout lane
2. **Customer Processing Functions**: Three additional functions are defined to simulate the actual processing of individual customers in each lane:
   * left\_processing\_customer(line\_id, customer): Simulates processing a customer in the self-checkout lane
   * central\_processing\_customer(line\_id, customer): Simulates processing a customer in the traditional cashier lane
   * right\_processing\_customer(line\_id, customer): Simulates processing a customer in the 15-item express lane
3. **Process Creation and Execution**: The main part of the code creates separate multiprocessing.Process instances for each checkout lane, passing the corresponding function and customer list as arguments. These processes are then started concurrently using the start() method, and the main process waits for them to complete using the join() method.
4. **Runtime Measurement**: The code measures the overall runtime of the parallel simulation using the time.perf\_counter() function, which provides a high-precision timer for performance benchmarking.

**Output**



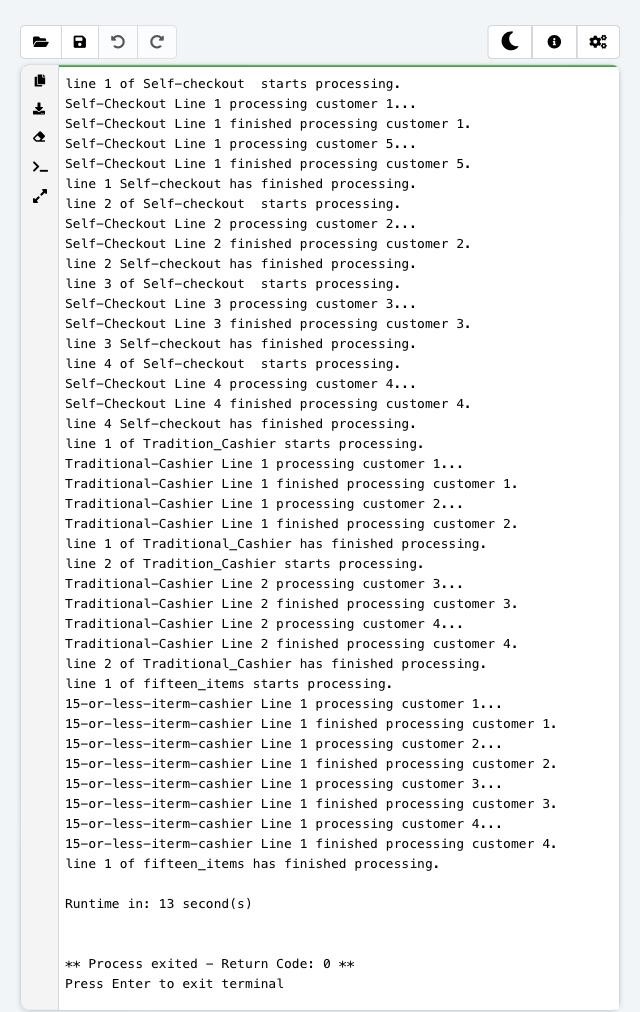
## Serial Computing Alternative

The serial computing version follows a single-threaded execution approach, calling each checkout lane function sequentially. This version serves as a baseline for comparison with the parallel version.

### Implementation Details

The serial version consists of the following key components:

1. **Checkout Lane Functions**: The same three functions are used to simulate the processing of customers in each checkout lane:
   * self\_checkout\_cashier(line\_id, customers)
   * traditional\_cashier(line\_id, customers)
   * fifteen\_item\_cashier(line\_id, customers)
2. **Customer Processing Functions**: The same three functions are used to simulate the actual processing of individual customers in each lane:
   * left\_processing\_customer(line\_id, customer)
   * central\_processing\_customer(line\_id, customer)
   * right\_processing\_customer(line\_id, customer)
3. **Sequential Execution**: In the main part of the code, the checkout lane functions are called sequentially, one after another, passing the corresponding customer lists as arguments.
4. **Runtime Measurement**: Similar to the parallel version, the code measures the overall runtime of the serial simulation using the time.perf\_counter() function.



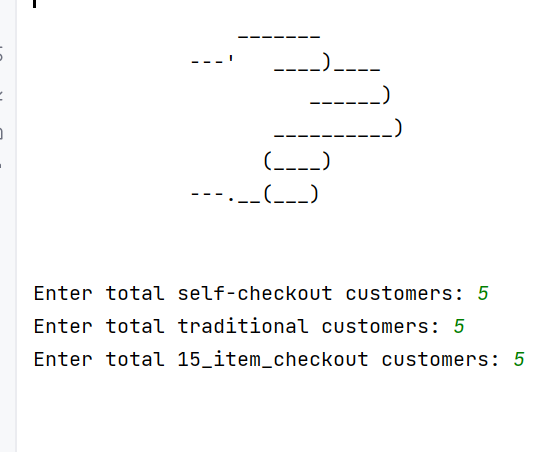
## Performance Comparison

By running both versions of the simulation and measuring their respective runtimes, you can compare the performance of the parallel and serial approaches. The parallel version leverages multiple processes running concurrently, potentially leading to faster execution times, especially on systems with multiple CPU cores or processors. In particular, the machine running these algorithms had four cores capable of hyper-threading. Virtually appearing as an eight core computer. Hence, execution time might very upon repeating this experiment on different computers.

As it is important to note that the performance gain from parallelization may be affected by factors such as the number of available CPU cores, the workload distribution among processes, and the overhead associated with process creation and communication.

As per results, assuming it takes five seconds to process a costumer in the supermarket, giving each checkout lane five costumers to process would theoretically take approximately fifteen seconds to process all costumers sequentially. However, processing these customers in parallel would substantially reduce this value as shown below.

Parallel processing time of fifteen costumers:  

Compared to processing the same the number of customers in serial.



## Conclusion

The parallel checkout simulation demonstrates the use of Python's *multiprocessing* module to achieve concurrent execution of tasks, simulating multiple checkout lanes in a store. By comparing the performance of the parallel and serial versions, you can observe the potential benefits of parallelization for this particular use case.

Future improvements could include exploring different workload distributions, optimizing the process creation and communication overhead, and investigating the scalability of the parallel approach with varying numbers of checkout lanes and customers.

## Appendix

The full source code for both the parallel and serial versions is up loaded on GitHub for reference.