Parallelizing the Monte Carlo Simulation on MecSimCalc

Harry Peng[1], Prof. Samer Adeeb[2]
[1]Department of Civil and Environmental Engineering, University of Alberta

BACKGROUND

MONTE CARLO SIMULATION

- Computational technique used to estimate the probability of different outcomes in a process that involves variables that change by random amounts.
- Used where the underlying system is too complex to be solved analytically
- Has applications in civil engineering (ie. Strain demand in pipes subjected to ground movement [1])
- **However, computationally heavy and slow**

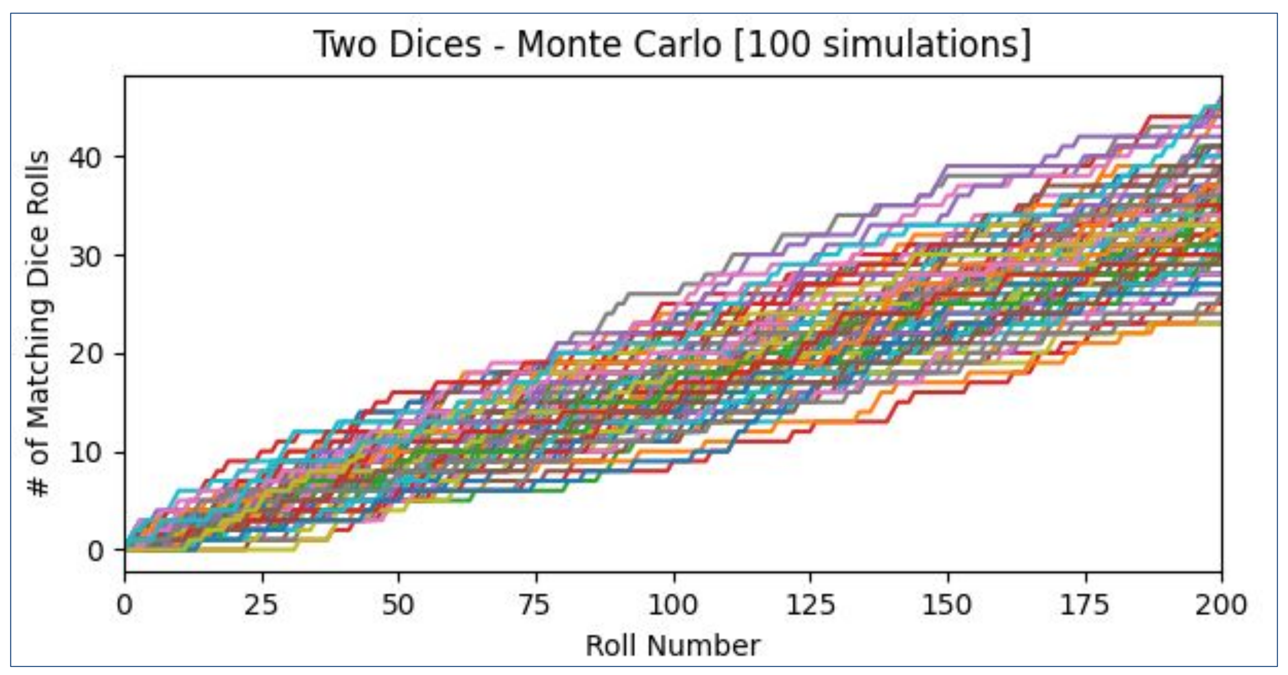

Figure 1: Graph of Monte Carlo Simulation results adapted from [2]. This graph illustrates the number of matching dice rolls in 100 sets of 200 consecutive rolls.

PARALLEL COMPUTING

MULTIPROCESSING

- Can significantly reduce the computation time of Monte Carlo Simulations
- One way of achieving parallelism through using multiple CPU cores for processing
- Supported on MecSimCalc with multiple virtual CPUs

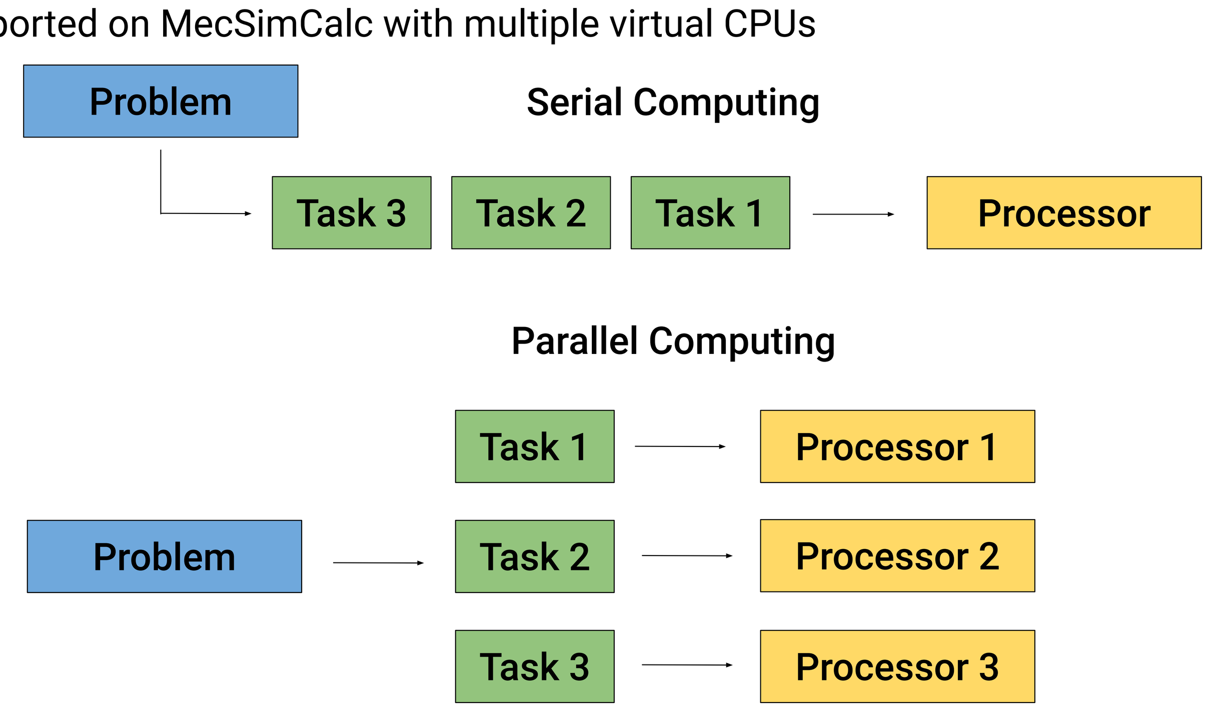

Figure 2: Diagram comparing serial computing and parallel computing

OBJECTIVES

- Develop a procedure to implement multiprocessing on pre-existing Monte Carlo simulations in Python
- Assess the impact of multiprocessing on Monte Carlo simulation execution time
- Evaluate the cost feasibility of multiprocessing on MecSimCalc

METHOD

IMPLEMENTING MULTIPROCESSING

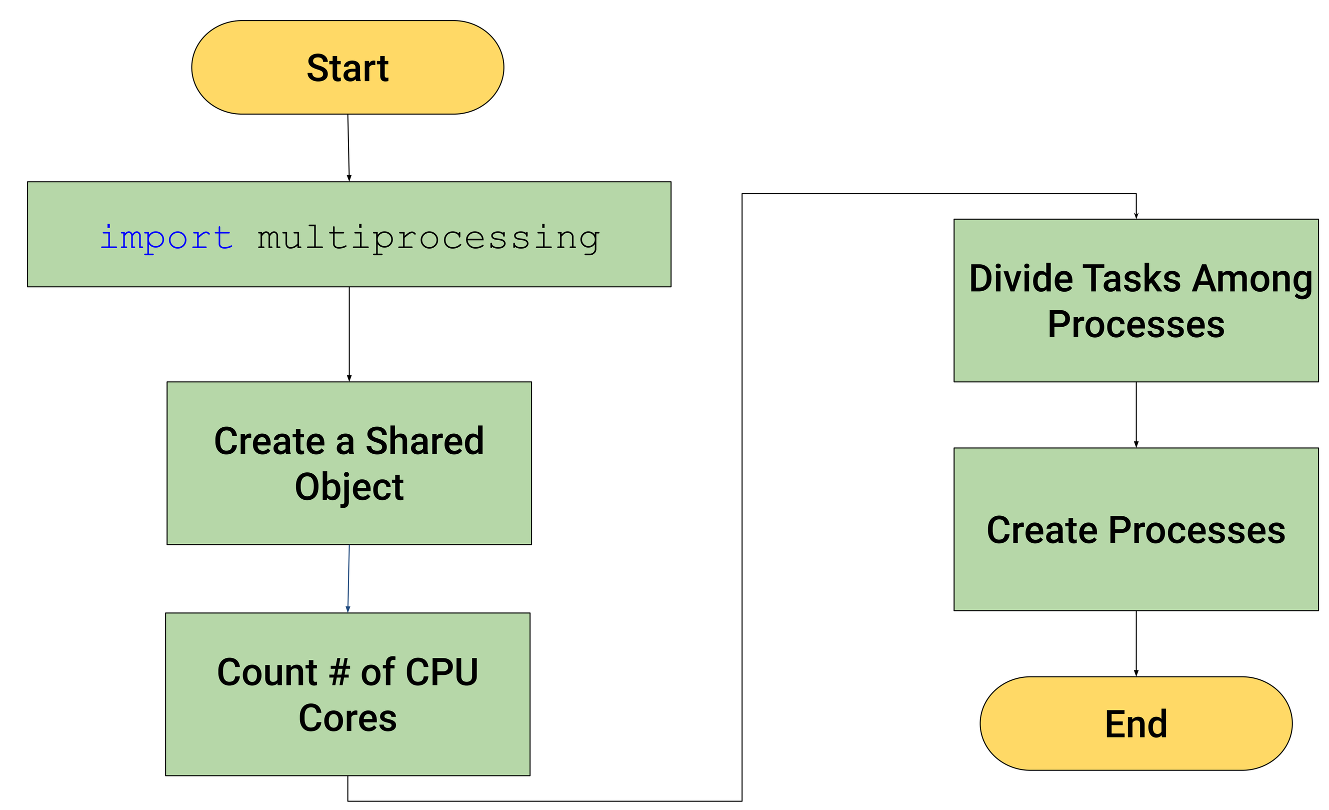

Figure 3: Flowchart of multiprocessing implementation in Python on MecSimCalc.

1. **Modularize the Program**
   Arrange the program into functions so there is one function that runs the simulation. This will make it easier to implement multiprocessing.

1. **Import the Multiprocessing Module**

```
import multiprocessing
```

1. **Create a Shared Object**
   An object with shared memory between the processes will be required to store any function return data. This is done using the manager object from the multiprocessing module.

```
def main(inputs):
    ### Other Code
    manager = mp.Manager() # This is case-sensitive
    data = manager.list()
    ### Other Code
```

1. **Count the Number of CPU Cores**

```
def main(inputs):
    ### Other Code
    num_cores = multiprocessing.cpu_count()
    ### Other Code
```

1. **Divide Tasks Among Each Process**

```
def main(inputs):
    ### Other Code
    num_processes = num_cores # num_processes can also be a manually chosen int
    simulations_per_process = num_simulations // num_processes # Floor divide the tasks into 
each process
    remainder = num_simulations % num_processes # Determine the remainder if there is one
    ### Other Code
```

1. **Create the Processes**

```
def main(inputs):
    ### Other Code
    processes = []
    for i in range(num_processes):
        # Create the process
        if i < remainder:
            p = mp.Process(target=simulation, args=(simulations_per_process+1, 
simulation_args))
            # Add another simulation for each remainder found in step 5
        else:
            p = mp.Process(target=simulation, args=(simulations_per_process, 
simulation_args))
        processes.append(p) # Add the process to the list of processes
        p.start() # Starts the process

    for p in processes:
        p.join()
        # This waits for other processes to finish executing before continuing
    ### Other Code
```

RESULTS

PROGRAM EXECUTION TIMES

Table 1: Relationship between # of CPU cores and execution time and cost of sequential and multiprocessed Monte Carlo simulation for 1,000 iterations.

Execution Time (1,000 iterations)				
Cores	Sequential (s)	Cost (USD)	Multiprocessed (s)	Cost (USD)
2	1292.293322	0.11	1201.466	0.11
4	1312.954062	0.18	636.8041	0.09
8	1257.982531	0.28	315.3643	0.09
16	1296.069194	0.58	132.832	0.08

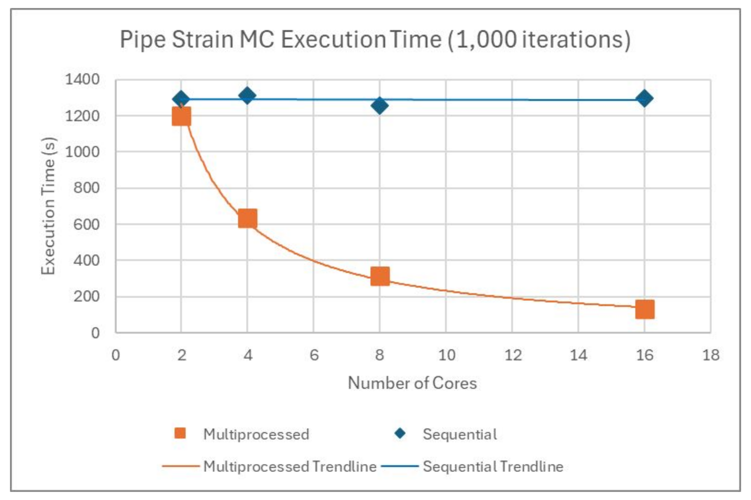

Figure 4: Graph comparing # of CPU cores to execution time for a complex Monte Carlo simulation with 1,000 iterations using code adapted from [1].

Table 2: Relationship between # of CPU cores and execution time and cost of sequential and multiprocessed Monte Carlo simulation for 10,000 iterations.

Execution Time (10,000 iterations)				
Cores	Sequential (s)	Cost (USD)	Multiprocessed (s)	Cost (USD)
2	12620.36506	1.03	12504.90886	1.02
4	12946.5503	1.97	6542.589321	1.04
8	12937.67383	2.99	3218.02439	0.82
16	12999.22588	4.78	1605.587904	0.71

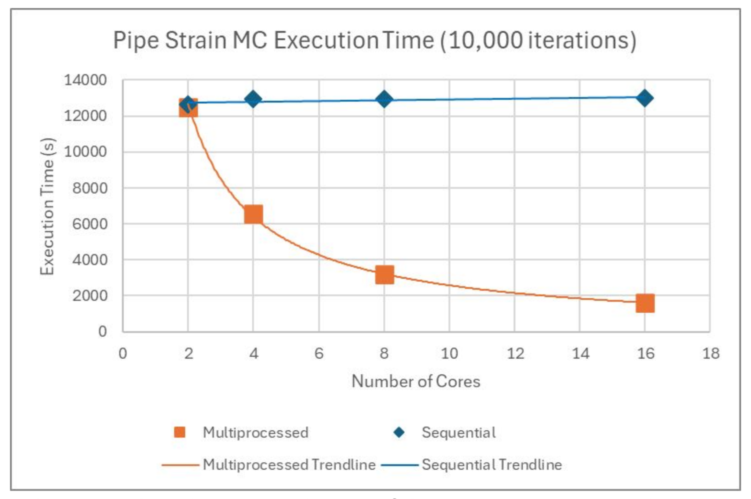

Figure 5: Graph comparing # of CPU cores to execution time for a complex Monte Carlo simulation with 10,000 iterations using code adapted from [1].

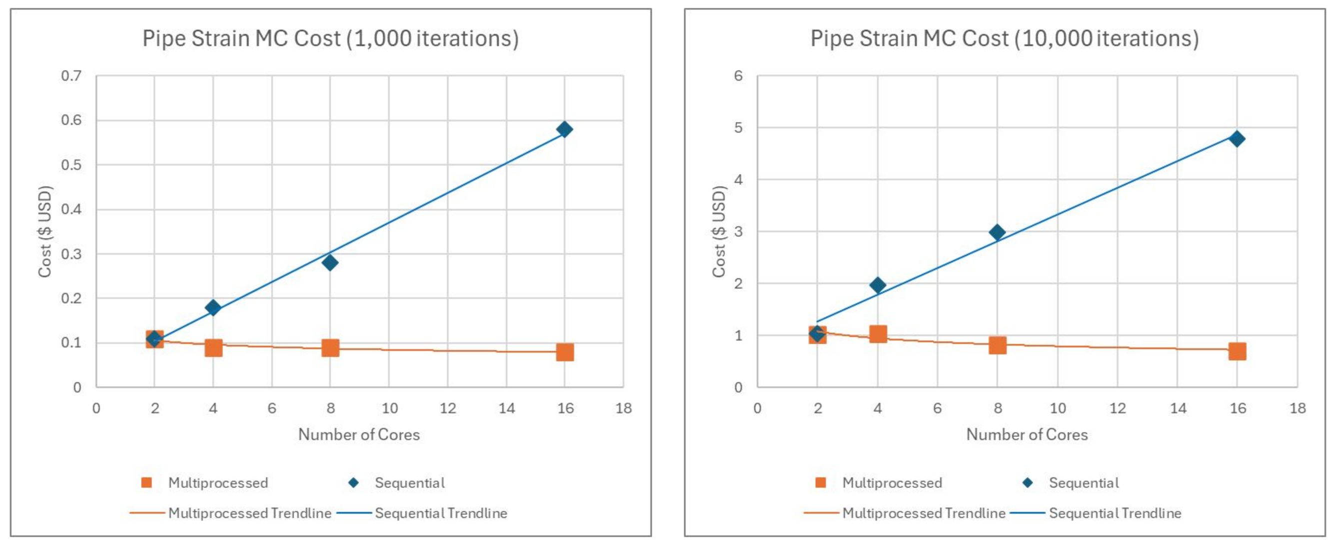

Figure 6 & 7: Graph comparing # of CPU cores to cost for a complex Monte Carlo simulation with 1,000 and 10,000 iterations using code adapted from [1].

CONCLUSION

The ratio of execution time between multiprocessed and sequential computations remains constant regardless of the number of iterations. In the results, the multiprocessed approach takes approximately 10% of the sequential execution time when using 16 CPU cores.

Additionally,, running the multiprocessed program with 16 cores is more cost-effective compared to using fewer cores or running sequentially with just 2 cores. Despite the higher cost per unit time, the speed of 16 cores in parallel makes this the most cost-effective and time-efficient choice.

Implementing multiprocessing is essential for efficiently executing long and computationally heavy Monte Carlo simulations, a task that would not be practical otherwise.

[1] Q. Zheng, “Stress- and Strain-Based Reliability Assessment of Pipelines Subjected to Internal Pressure and Permanent Ground Movement,” Ph.D. Thesis, Department of Civil and Environmental Engineering, University of Alberta, Edmonton, 2023.
[2] J. Matthew, “An introduction to Monte Carlo simulations using Python,” Medium, Oct. 14, 2023. https://medium.com/@matthew1992/an-introduction-to-monte-carlo-simulations-using-python-46c07eb11b6d (accessed Mar. 20, 2024).