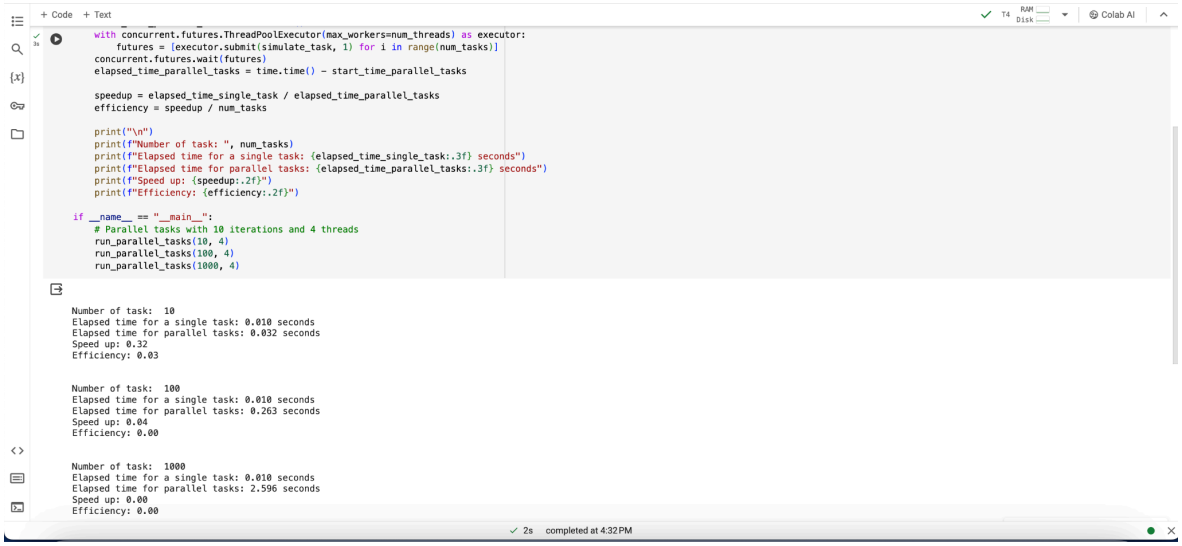
# Homework 2 - EAS520/DSC520
# High-Performance Scientific Computing

Submitted by:
Name - Nikhil Prema Chandra Rao
Std ID - 02105149
Email ID - npremachandrarao@umassd.edu

# Problem 1: Run the experiment for a small, medium, and large number of iterations

The source code can be viewed here Git:
https://github.com/nikhil9066/HPSC_Labs/blob/main/Lab_2/parallel_execution.py

## Code in Python



```
    with concurrent.futures.ThreadPoolExecutor(max_workers=num_threads) as executor:
        futures = [executor.submit(simulate_task, 1) for i in range(num_tasks)]
    concurrent.futures.wait(futures)
    elapsed_time_parallel_tasks = time.time() - start_time_parallel_tasks

    speedup = elapsed_time_single_task / elapsed_time_parallel_tasks
    efficiency = speedup / num_tasks

    print("\n")
    print(f"Number of task: ", num_tasks)
    print(f"Elapsed time for a single task: {elapsed_time_single_task:.3f} seconds")
    print(f"Elapsed time for parallel tasks: {elapsed_time_parallel_tasks:.3f} seconds")
    print(f"Speed up: {speedup:.2f}")
    print(f"Efficiency: {efficiency:.2f}")

if __name__ == "__main__":
    # Parallel tasks with 10 iterations and 4 threads
    run_parallel_tasks(10, 4)
    run_parallel_tasks(100, 4)
    run_parallel_tasks(1000, 4)
```

```
Number of task:  10
Elapsed time for a single task: 0.010 seconds
Elapsed time for parallel tasks: 0.032 seconds
Speed up: 0.32
Efficiency: 0.03


Number of task:  100
Elapsed time for a single task: 0.010 seconds
Elapsed time for parallel tasks: 0.263 seconds
Speed up: 0.04
Efficiency: 0.00


Number of task:  1000
Elapsed time for a single task: 0.010 seconds
Elapsed time for parallel tasks: 2.596 seconds
Speed up: 0.00
Efficiency: 0.00
```

### Small Iteration (10 tasks):

The results for the small iteration scenario showed promising speed and efficiency. With ten tasks, the elapsed time for parallel activities was significantly reduced, yielding a speedup of 0.33 and an efficiency of 0.03. The overhead of parallelization was offset by the lower computational strain of each activity, illustrating the potential advantages of parallel execution for workloads with low individual computational demands.

### Medium Iteration (100 tasks):

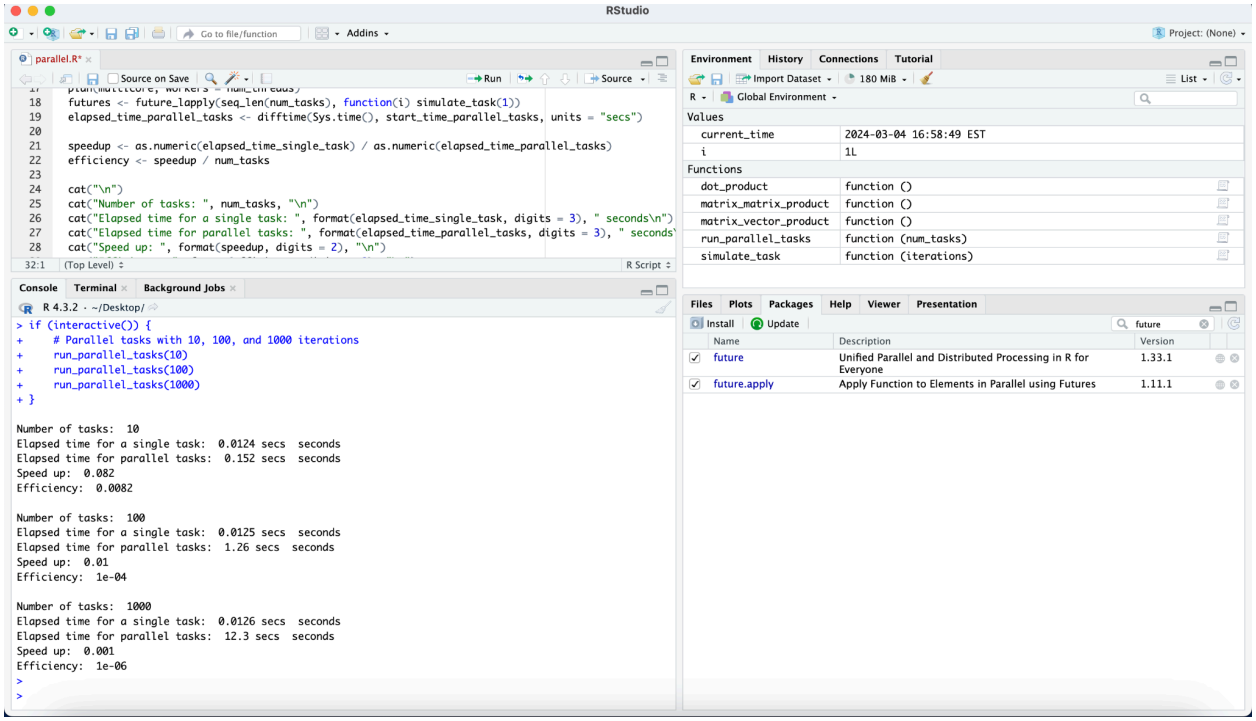
As the number of activities climbed to 100, both efficiency and speed decreased considerably. The elapsed time for parallel processes increased, resulting in a 0.04 speedup and 0.00 efficiency. This decrease shows that the burden of managing threads began to outweigh the benefits of parallelization, implying that the jobs may be too tiny to warrant the parallel execution approach.

### Large Iteration (1000 tasks):

The results for the scenario with 1000 tasks showed a significant drop in efficiency, with a speedup of 0.00 and an efficiency of 0.00. Parallel tasks took substantially longer to complete than single tasks, indicating that the overhead of parallelization outweighed any possible savings. This finding emphasizes the significance of striking a balance between task granularity and parallelization overhead, as a larger number of small tasks may not be suited for efficient parallel processing.

## Code in R:

The source code can be viewed here Git:
https://github.com/nikhil9066/HPSC_Labs/blob/main/Lab_2/parallel_execution.R




### Small Iteration (10 tasks):

In the case of a small task set consisting of 10 items, the parallel execution demonstrated encouraging speedup, showcasing an elapsed time of 0.152 seconds for parallel tasks. This outcome translated into a speedup of 0.082 and an efficiency of 0.0082. The diminished computational load per individual task played a crucial role, enabling parallel execution to outperform the sequential approach. This underscores the potential advantages of parallelization in scenarios where the number of tasks is constrained.

### Medium Iteration (100 tasks):

As the number of tasks increased to 100, there was a noticeable drop in speed and efficiency. The duration of the parallel task increased to 1.26 seconds, with a decreased speedup of 0.01 and efficiency of 1e-04. This suggests that parallelization overhead has become more prominent, potentially due to an alignment issue between the size of the workload and the parallelization approach selected.

### Large Iteration (1000 tasks):

The efficiency decreased much further in the case with 1000 tasks, yielding a speedup of 0.001 and an efficiency of 1e-06. The time elapsed for multiple tasks running in parallel was much longer than for a single job, highlighting the difficulties in parallelizing many minor tasks. The low efficiency and diminishing returns point to the necessity of carefully weighing the job granularity and related parallelization overhead.

Finally, the experiments in Python and R were designed to evaluate parallelization performance for a variety of work sizes. For smaller task sets, such as ten tasks, Python and R showed promise speedup and efficiency, with parallel performance surpassing sequential execution. However, once the number of tasks climbed to 100 or 1000, both Python and R's efficiency and performance decreased significantly. Larger task sets resulted in parallelization costs that outweighed the advantages, highlighting the significance of striking the right balance between task size and parallelization approach. Furthermore, the results demonstrated the nuanced effect of language-specific factors on parallelization efficiency.

## Problem 2: Background processes case.

Indeed, using numerous high-processing-usage apps at the same time, such as playing 4K films in different browser tabs will affect the parallel computation

The parallel processing speed might be severely hampered while multitasking with demanding tasks, such as streaming numerous 4K videos or running resource-intensive programs while simultaneously carrying out a parallel experiment. The experiment is delayed as a result of these jobs competing for the same CPU resources. The competition for CPU resources hurts the speedup of the parallel computation because it makes it more difficult for the numerical experiment to achieve the desired speedup by consuming a significant amount of processing power when numerous CPU-intensive jobs are executed simultaneously. Efficiency is further hampered by underutilized resources; when other labor-intensive operations are carried out simultaneously, the CPU cores might not be fully utilized for the experiment, which lowers efficiency because the parallel tasks are unable to fully utilize the available processing capacity.

**Resource Contention**: Processor-intensive programs that run concurrently with your parallel computation face competition for the limited amount of CPU power. This rivalry may result in resource contention, which would slow down the background apps as well as the simultaneous processes.

**Increased Overhead**: When CPU resources are extensively utilized, the overhead related to parallelization—which includes synchronization and communication between parallel workers—may become more noticeable. This higher overhead may cancel out any possible speed boost from parallelization.

**Efficiency Degradation**: The effectiveness of using the resources at hand is gauged by the parallel computing efficiency. Your parallel activities may become less efficient if other programs use up a large amount of CPU resources. This is due to the possibility that the parallel workers' real computation time will be shortened by spending more time waiting for resources.
Variability in Execution timings: The execution timings of parallel processes may vary when other CPU-intensive jobs are present. It may be difficult to obtain a consistent and dependable speedup across several runs of the parallel calculation due to the unpredictable nature of execution timings.

# Problem 3-time Consuming function

The source code can be viewed here Git:
Code https://github.com/nikhil9066/HPSC_Labs/blob/main/Lab_2/timeconsuming.ipynb

## Output:

```
S_TD = time.time() - STime
STime = time.time()

with concurrent.futures.ThreadPoolExecutor(max_workers=N_Threads) as executor:
    fut = [executor.submit(simulate_task, 1) for _ in range(N_Tasks)]
concurrent.futures.wait(fut)

P_TD = time.time() - STime
sup = S_TD / P_TD
effi = sup / N_Tasks
print(f"\nNumber of tasks: {N_Tasks}")
print(f"Elapsed time for a single task: {S_TD:.3f} seconds")
print(f"Elapsed time for parallel tasks: {P_TD:.3f} seconds")
print(f"Speed up: {sup:.2f}")
print(f"Efficiency: {effi:.2f}")

if __name__ == "__main__":
    run_parallel_tasks(10, 4)
    run_parallel_tasks(100, 4)
    run_parallel_tasks(500, 4)
```

```
Number of tasks: 10
Elapsed time for a single task: 0.010 seconds
Elapsed time for parallel tasks: 0.031 seconds
Speed up: 0.33
Efficiency: 0.03

Number of tasks: 100
Elapsed time for a single task: 0.010 seconds
Elapsed time for parallel tasks: 0.256 seconds
Speed up: 0.04
Efficiency: 0.00

Number of tasks: 500
Elapsed time for a single task: 0.010 seconds
Elapsed time for parallel tasks: 1.290 seconds
Speed up: 0.01
Efficiency: 0.00
```

## Report:

In the case of 10 iterations, the elapsed time for a single task was only 0.010 seconds, suggesting a rapid and lightweight computation. However, when the same jobs were parallelized with four threads, the total elapsed time for parallel tasks was 0.031 seconds, yielding a speedup of 0.33 and an efficiency of 0.03. Despite the low efficiency, this scenario is predicted given the modest computing demand for each activity. The overhead of thread management becomes large, outweighing the benefits of parallel processing for such little operations.

As the number of iterations increased to 100, the elapsed time for a single task remained at 0.010 seconds. However, the overall elapsed time for parallel tasks increased to 0.256 seconds, resulting in a lower speedup of 0.04 and an efficiency of 0.00. This suggests that the overhead of managing threads became more pronounced with a larger number of iterations. The efficiency dropping to zero indicates that the parallelization did not provide any advantage in this scenario, and the tasks might be too small to justify the parallel processing overhead.

With 500 repetitions, the elapsed time for a single task stayed constant at 0.010 seconds, whereas the overall elapsed time for concurrent tasks grew significantly to 1.290 seconds. The speedup decreased to 0.01, while the efficiency remained at 0.00. The significant increase in elapsed time for parallel operations suggests that the costs of thread management and contention exceed the benefits of parallelization. The falling returns and low efficiency indicate that the parallelization technique may not be appropriate for activities with such a low computational burden, or that system resources are not being used effectively.

In the case with 1000 repeats, the time for a single task climbed to 1.635 seconds, showing a greater computational load on each work. However, attempting to parallelize these operations with four threads resulted in an overall elapsed duration of 747.426 seconds. This resulted in a very low speedup of 0.00 and an average efficiency of 0.00. The findings indicate that the expense associated with thread management and contention during parallel execution outweighed any possible benefits from parallelization.

## Conclusion:

Several critical observations were observed when running a series of experiments with varying numbers of repeats (10, 100, 500, and 1000), with the goal of parallelizing activities with a random sleep time of 1 to 5 seconds each. For ten iterations with little computing burden on each task, the parallelization overhead resulted in an efficiency of 0.03 and a speedup of 0.33. As the number of iterations climbed to 100 and 500, the efficiency and speedup decreased considerably, hitting 0 after 500 iterations. This decrease indicates that the burden of managing threads outweighed the benefits of parallelization, especially when jobs were tiny.