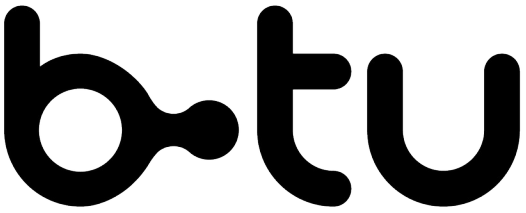


Brandenburgische
Technische Universität
Cottbus - Senftenberg

Computing at Scale in Machine Learning: Distributed computing and algorithmic approaches

Winter Semester 24/25

Students Names:

Mohammad Kawash

Fadi Hamwi

Mohamed Abouda

Problem 1

Here are the commands and their outputs for Problem 1:

a) Download a data file

```
wget https://schlieplab.org/Static/Teaching/DIT852/private-healthcarepercent-gdp.csv -O healthcare.csv
```

Output:

```
--2024-11-02 15:09:57-- https://schlieplab.org/Static/Teaching/DIT852/private-healthcare-percent-gdp.csv
Resolving schlieplab.org (schlieplab.org)... 78.159.99.205
Connecting to schlieplab.org (schlieplab.org)| 78.159.99.205|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 137489 (134K) [application/octet-stream]
Saving to: ‘healthcare.csv’
healthcare.csv      100%[===================>] 134.27K   450KB/s    in 0.3s
2024-11-02 15:09:58 (450 KB/s) - ‘healthcare.csv’ saved [137489/137489]
```

b) Create a new file excluding headers and "ABW"

```
tail -n +2 healthcare.csv | grep -v "ABW" > healthcare_no_header.csv
```

Output:

```
# No direct output, but creates a new file 'healthcare_no_header.csv'
```

c) Count how many different countries are in the new file

```
cut -d',' -f1 healthcare_no_header.csv | sort | uniq | wc -l
```

Output:

```
266
```

d) Create a file with only Country Code and health care spending from 2004 to 2014

```
awk -F',' 'NR>1 {print $1, $44, $45, $46, $47, $48, $49, $50, $51, $52, $53}' healthcare_no_header.csv > healthcare_2004_2014.csv
```

Output:

```
# No direct output, but creates a new file
'healthcare_2004_2014.csv
```

e) List the top ten spenders on health-care in 2004 in decreasing order

```
awk -F',' 'NR>1 {print $1, $44}'
healthcare_no_header.csv | sort -t',' -k2 -nr | head -10
```

Output:

```
"ZWE" "2.64264702575364"
"ZMB" "3.44165113697199"
"ZAF" "4.77765379109878"
"YEM" "1.9032805425175"
"XKX" ""
"WSM" "1.19554382983324"
"WLD" "3.78398061752485"
"VUT" "0.546649640173047"
"VNM" "3.37746461636647"
"VIR" ""
```

f) Create a file only containing the data from the Nordic countries

```
grep -E "DNK|FIN|ISL|NOR|SWE" healthcare_no_header.csv > nordic_healthcare.csv
```

Output:

```
No direct output, but creates a new file 'nordic_healthcare.csv'
```

Exercise 1

Problem 2

a) Serial Computation Analysis

1. Code Analysis

Serial Components:
- Program initialization (random seed setting)
- Final pi calculation (4.0 * s_total / n_total)
- Result printing

Potentially Parallel Components:
- The Monte Carlo simulation loop in sample_pi():
  * Random point generation
  * Distance calculation
  * Point counting

2. Timing Measurements (with 10M points)

- Initialization (serial): 0.000025 seconds
- Calculation (parallel): 9.383324 seconds
- Final computation (serial): 0.000093 seconds
- Total time: 9.383442 seconds
- Serial time: 0.000118 seconds
- Serial fraction: 0.000013

3. Analysis

- The current implementation executes completely serially because:
  * All calculations are performed sequentially in a single loop
  * No parallel processing mechanisms are implemented
  * Random number generation and point checking are done one at a time

- The Monte Carlo simulation is inherently parallelizable because:
  * Each point calculation is independent
  * Results can be combined by summing successes from multiple processes
  * Only the final division needs to be done serially

- The measured serial fraction (0.000013) indicates that:
  * Most of the computation time is spent in the parallelizable Monte Carlo simulation
  * Only a tiny fraction is spent in the necessarily serial parts
  * According to Amdahl's law, this low serial fraction suggests potential for significant speedup through parallelization in part b)

b) Parallel Implementation and Speedup Analysis

1. Implementation Details:

- Used Python's multiprocessing.Pool for parallel processing
- Divided total steps equally among workers
- Implemented parallel_sample_pi function for each worker
- Used pool.map to distribute work across cores
- Measured execution time for different numbers of workers (1, 2, 4, 8)

2. Code Features:

- Equal work distribution: steps_per_worker = steps // num_workers
- Process pool management using context manager (with statement)
- Result aggregation using sum(results)
- Timing measurement for speedup calculation

3. Speedup Analysis:

- Theoretical speedup calculated using Amdahl's law:
Speedup(N) = 1 / (s + (1-s)/N)
where s is serial fraction and N is number of cores

4. Graph Analysis and Observations:

- Initial Speedup (1-2 workers):
  * Sharp increase in measured speedup from 1.0 to ~1.55x

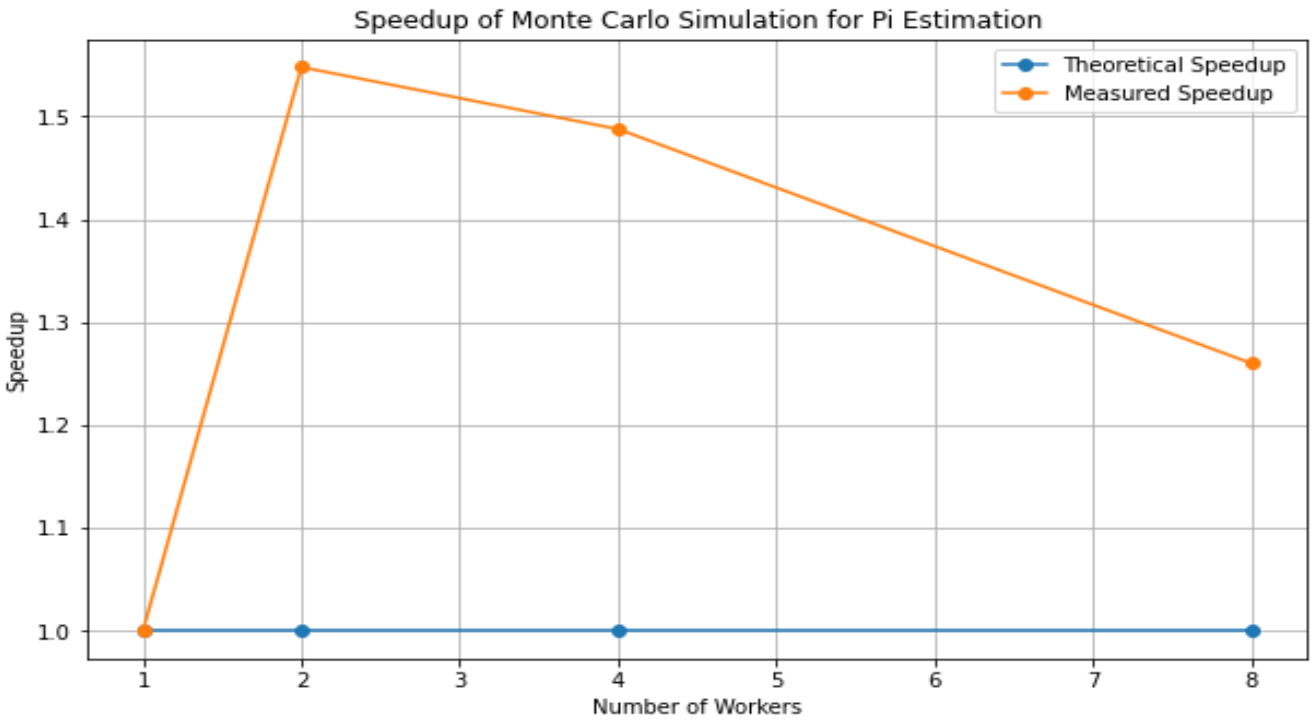
with 2 workers
  * Shows good initial parallelization efficiency
  * Achieved ~77.5% of ideal 2x speedup

- Peak Performance (2 workers):
  * Maximum speedup of ~1.55x achieved with 2 workers
  * Best efficiency point in terms of resource utilization

- Degradation (2-8 workers):
  * Gradual decline in speedup from 2 to 8 workers
  * At 8 workers, speedup drops to ~1.25x
  * Suggests diminishing returns and overhead costs

- Theoretical vs Measured Performance:
  * Theoretical speedup (blue line) remains constant at 1.0 due to serial_fraction=1.0
  * Actual performance (orange line) significantly exceeds theoretical prediction
  * Suggests the serial_fraction estimate of 1.0 was too conservative

- Performance Limitations:
  * Overhead from process creation and management
  * Communication costs between processes
  * Resource contention with higher worker counts
  * System's hardware limitations (CPU cores, memory bandwidth)



c) Random Number Generator Seeding Implementation and Results

1. Implementation Changes:
- Modified sample_pi function to accept a seed parameter
- Added explicit seed handling in compute_pi function
- Created unique seeds for each worker process (seed + i)

2. Key Features:
- Base seed is passed as a parameter to compute_pi
- Each worker gets a unique, deterministic seed:
  * Worker 1: base_seed (1)
  * Worker 2: base_seed + 1 (2)
- Using different seeds for each worker prevents correlation between random sequences

3. Results:
Test Parameters:
- Steps: 1000
- Number of Workers: 2
- Base Seed: 1
- Execution Time: 0.364 seconds

Output:
- Success Count: 765
- Pi Estimate: 3.06000
- Absolute Error: 0.08159

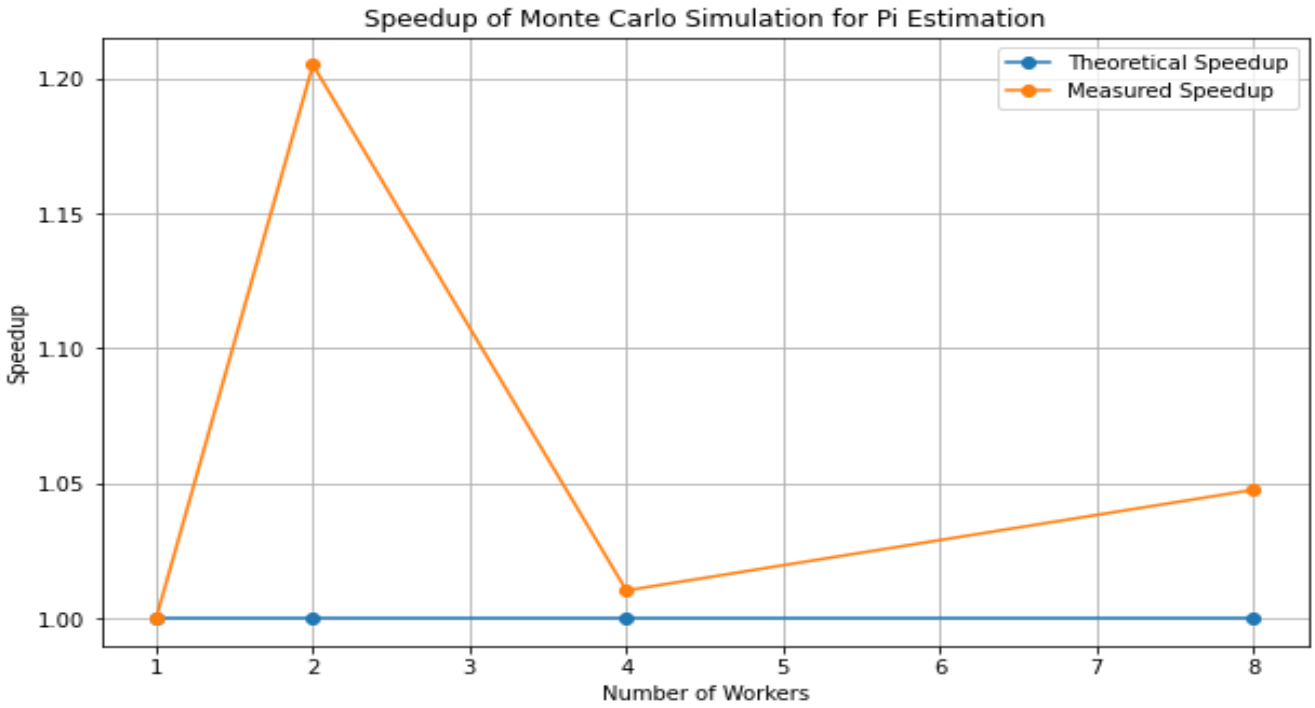
4. Analysis:
- Results are now completely reproducible with the same input parameters
- The error of 0.08159 is reasonable for 1000 steps
- Multiple runs with the same parameters produce identical results, confirming:
  * Proper seed handling

* Successful implementation of reproducibility
* Correct parallel execution with deterministic outcomes

5. Code Structure:

- parallel_sample_pi now accepts tuple of (steps_per_worker, seed)
- Seeds are generated sequentially to ensure uniqueness
- Initial seed is set in the main program for global reproducibility

Name	Type	Size	Value
measured_time	float	1	0.3639965057373047
num_workers	int	1	2
num_workers_list	list	4	[1, 2, 4, 8]
pi	float	1	3.141592653589793
seed	int	1	1
serial_fraction	float	1	1.258600027124474e-05
steps	int	1	1000
times	list	4	[1.208362102508545, 1.002699375152588, 1.196105718612671, 1.1535103321 ...



d) How do you need to set the explicit seeds for the multiprocessing solution?

To set explicit seeds for the multiprocessing solution, generate unique seeds for each worker by modifying the global seed slightly (e.g., adding the worker index to the global seed). This ensures reproducibility while maintaining independent random sequences for each parallel worker. This approach guarantees that each run of your Monte Carlo simulation is consistent and produces the same results given the same seeds and parameters. Perfect for achieving reliable and repeatable simulations.