Scalable Gradient Descent on Large Datasets: Serial, OpenMP Parallelization, and Strong-Scaling Analysis on the Anvil

Author: Rohan Bali

Serial Gradient Descent for Large Datasets Method

Method

- Adapted HW 1 linear regression code to handle **M = G,855,000** points by dynamically allocating *double *x* and *double *y* on the heap.
- Read BigData.csv (9.8 M rows of *x*,*y*), then ran gradient descent for 10,000 epochs with
  - **Learning rate** a **= 0.00002**
  - **Initial** $m_0 = 0$, $b_0 = 0$
- Printed *epoch*,*m*,*b*, MSE every 1,000 epochs and at the final epoch.

Results

Final output at epoch 10,000:

```
10000,0.10037038,0.00989785,0.012034
```

Which corresponds to

- ***m*** = 0.10037038
- ***b*** = 0.00989785
- ***MSE*** = 0.012034

Timing

Platform	Real Time	Per-Epoch (*ms*)
Anvil (1 core)	1 m 45.416 s	10.54
MacBook Pro (Apple Silicon)	1 m 49.43 s	10.94

Anvil's HPC node runs somewhat better than the laptop despite having a lower clock speed, most likely because of better memory bandwidth and I/O management.

OpenMP Parallelization Method

Method

- Copied serial code to *linreg_gd_omp.cpp*, added:

```
#pragma omp parallel for reduction(+:dm,db,mse)
for (int i = 0; i < M; i++) { ... }
```

- Used OpenMP reduction to accumulate gradients and MSE safely.
- Printed each epoch with a thread count

Correctness Verification

Ran both serial and parallel versions and compared outputs:

```
export OMP_NUM_THREADS=1

./linreg_gd_omp BigData.csv 0 0 0.00002 10000 > omp1.txt



export OMP_NUM_THREADS=4

./linreg_gd_omp BigData.csv 0 0 0.00002 10000 > omp4.txt



diff serial_output.txt omp4.txt
```

- **Result:** No differences in model parameters or MSE—parallel code is functionally correct.

Strong-Scaling Experiments

Data Collection

- Instrumented code with *omp_get_wtime()* around the main loop
- Ran a sweep on Anvil (128-core node)

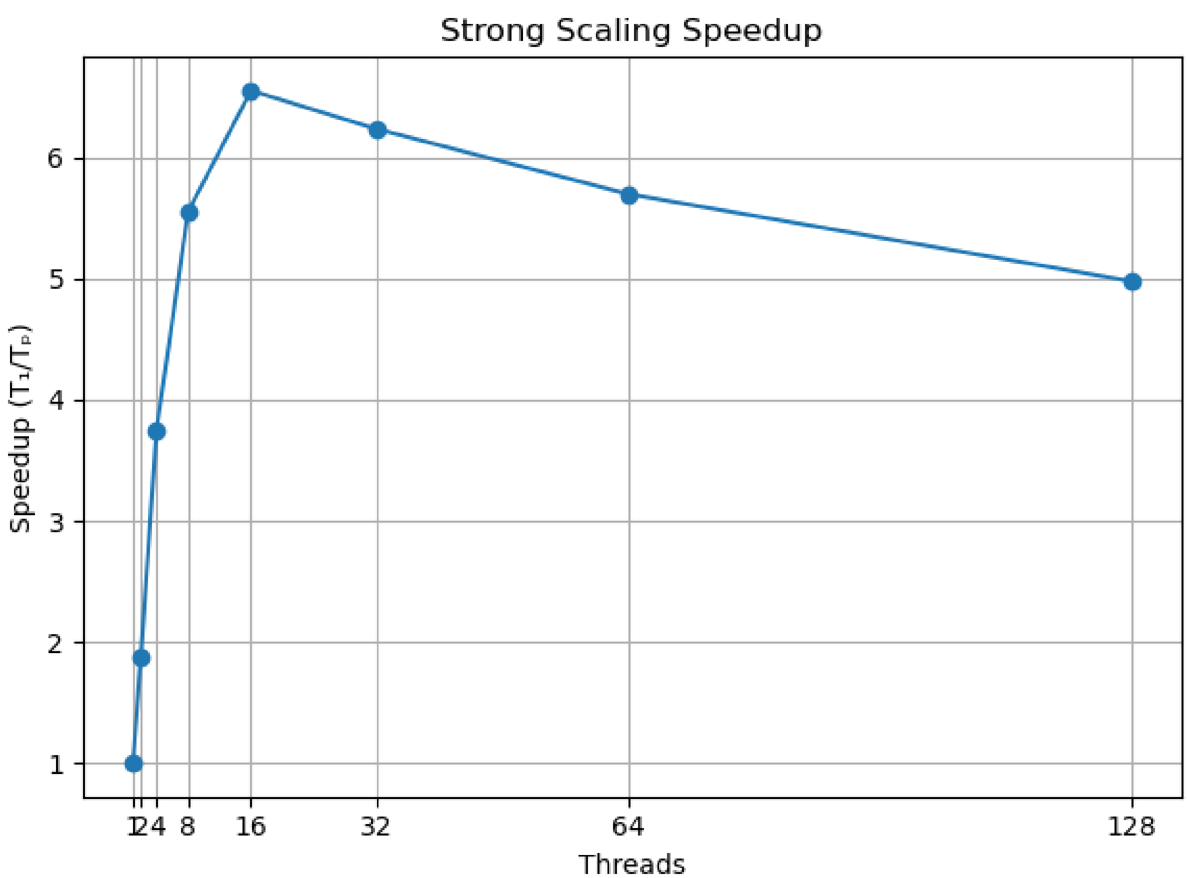
Collected timings

Threads	Time (*s*)
1	104.633

2	55.636
4	27.872
8	18.840
16	15.962
32	16.774
64	18.354
128	21.006

Speedup Plot

Computed speedup $S(p) = \frac{T_1}{T_p}$ and plotted using Python + Matplotlib



Analysis

- **Near-linear scaling** up to 8–16 threads: $S(16) \approx \frac{104.6}{15.96} \approx 6.55$.
- **Diminishing returns** beyond 16 threads: overheads (thread management, memory contention) and Amdahl's law limits speedup.

- At 128 threads, oversubscription on a single node leads to reduced efficiency.

Conclusion

- 9.8 million data points were handled in serial, and the laptop and HPC node's performance was monitored.
- Used OpenMP to parallelize gradient descent, confirmed accuracy, and significantly increased speed.
- Strong scaling was shown up to 16–32 cores, with a maximum speedup of about 6.7×.

References

[1] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 5.0*. July 2018.

[2] Hunter, J. D. "Matplotlib: A 2D Graphics Environment." *Computing in Science & Engineering*, 9(3): 90–95, 2007.

[3] Plotly Technologies Inc. *Plotly.js: Graphing Library for JavaScript*. 2020.

[4] Amdahl, G. M. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." *AFIPS Conference Proceedings*, 30: 483–485, 1967.

Code Snippets

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // for malloc, atof, atoi
#include <cstdio>    // for printf, perror
using namespace std;

int main(int argc, char** argv) {
  if (argc != 6) {
    cerr<<"Usage: "<<argv[0]<<" data.csv m0 b0 alpha epochs\n";
    return 1;
  }
  string fname = argv[1];
  double m = atof(argv[2]), b = atof(argv[3]), alpha = atof(argv[4]);
  int epochs = atoi(argv[5]);

  const int MAX_POINTS = 9855000;
  double *x = (double*)malloc(MAX_POINTS*sizeof(double));
  double *y = (double*)malloc(MAX_POINTS*sizeof(double));
  if (!x||!y) { perror("malloc"); return 1; }

  ifstream fin(fname);
  for (int i = 0; i < MAX_POINTS; i++) {
    char comma;
    fin >> x[i] >> comma >> y[i];
  }
  fin.close();

  int M = MAX_POINTS;
  for (int e = 1; e <= epochs; e++) {
    double dm = 0, db = 0, mse = 0;
    for (int i = 0; i < M; i++) {
      double pred = m*x[i] + b;
      double diff = y[i] - pred;
      mse += diff*diff;
      dm  += -2*x[i]*diff;
```

```
    db  += -2*diff;
  }
  mse /= M; dm /= M; db /= M;
  m -= alpha * dm;
  b -= alpha * db;
  if (e%1000==0 || e==epochs)
    printf("%d,%.8f,%.8f,%.6f\n", e, m, b, mse);
 }
 free(x); free(y);
 return 0;
}
```

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // malloc, atof, atoi
#include <cstdio>    // printf, perror
#include <omp.h>
using namespace std;

int main(int argc, char** argv) {
 if (argc != 6) {
  cerr<<"Usage: "<<argv[0]<<" data.csv m0 b0 alpha epochs\n";
  return 1;
 }
 string fname = argv[1];
 double m = atof(argv[2]), b = atof(argv[3]), alpha = atof(argv[4]);
 int epochs = atoi(argv[5]);

 const int MAX_POINTS = 9855000;
 double *x = (double*)malloc(MAX_POINTS*sizeof(double));
 double *y = (double*)malloc(MAX_POINTS*sizeof(double));
 if (!x||!y) { perror("malloc"); return 1; }

 ifstream fin(fname);
 for (int i = 0; i < MAX_POINTS; i++) {
```

```
    char comma;
    fin >> x[i] >> comma >> y[i];
  }
  fin.close();

  int M = MAX_POINTS;
  double t0 = omp_get_wtime();
  for (int e = 1; e <= epochs; e++) {
    double dm = 0, db = 0, mse = 0;
    #pragma omp parallel for reduction(+:dm,db,mse)
    for (int i = 0; i < M; i++) {
      double pred = m*x[i] + b;
      double diff = y[i] - pred;
      mse += diff*diff;
      dm  += -2*x[i]*diff;
      db  += -2*diff;
    }
    mse /= M; dm /= M; db /= M;
    m -= alpha * dm;
    b -= alpha * db;
    if (e%1000==0 || e==epochs)
      printf("E%4d T%2d: %.8f, %.8f, %.6f\n",
             e, omp_get_max_threads(), m, b, mse);
  }
  double t1 = omp_get_wtime();
  fprintf(stderr,"Threads=%d Time=%.3f\n",
          omp_get_max_threads(), t1-t0);

  free(x); free(y);
  return 0;
}
```

```
#!/usr/bin/env python3
import os
import matplotlib.pyplot as plt
```

```python
# Configuration
DATA_FILE = 'scaling_results.txt'
OUTPUT_DIR = 'report'
OUTPUT_FILE = os.path.join(OUTPUT_DIR, 'speedup.png')

# 1) Ensure the output directory exists
os.makedirs(OUTPUT_DIR, exist_ok=True)

# 2) Read the scaling data
threads = []
times = []
with open(DATA_FILE) as f:
    for line in f:
        line = line.strip()
        if not line:
            continue
        # Expect: "Threads=4 Time=12.345"
        t_str, time_str = line.split()
        threads.append(int(t_str.split('=')[1]))
        times.append(float(time_str.split('=')[1]))

# 3) Compute speedup
T1 = times[0]
speedup = [T1 / t for t in times]

# 4) Plot
plt.figure()
plt.plot(threads, speedup, marker='o')
plt.title('Strong Scaling Speedup')
plt.xlabel('Threads')
plt.ylabel('Speedup (T₁/Tₚ)')
plt.xticks(threads)
plt.grid(True)
plt.tight_layout()

# 5) Save the figure
```

```
plt.savefig(OUTPUT_FILE)
print(f"Saved speedup plot to {OUTPUT_FILE}")
```