Efficient Matrix Multiplication

Team 12

Elijah Smith, Garrett Spears, Karina Narotam, Nathanael Gaulke, Simon Weizman

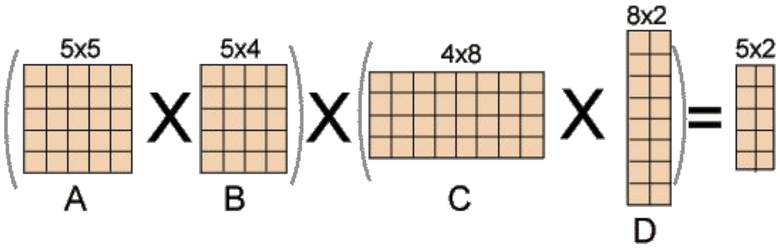
Our Project Goals

1. Parallelize matrix multiplication to speed up multiplication of large matrices
2. Create a simple and efficient library in Java for computing matrix chain multiplication

Background

Matrix Multiplication

- Res[i][j] = Row i of M1 · Col j of M2
- For a chain of n matrices multiplied together:
  - For each pair of adjacent matrices, inner dimensions must match!
  - Final matrix: (#rows M1 x #cols Mn)
- Associative



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

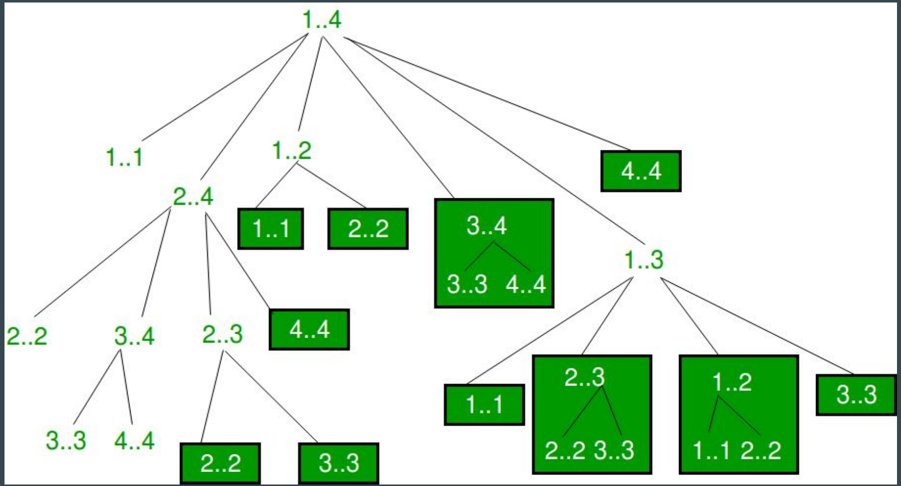
$$= \begin{bmatrix} 1x10 + 2x20 + 3x30 & 1x11 + 2x21 + 3x31 \\ 4x10 + 5x20 + 6x30 & 4x11 + 5x21 + 6x31 \end{bmatrix}$$

$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

Sequential Matrix Multiplication

```java
public Matrix multiply(Matrix other) {
    Matrix newMatrix = new Matrix(this.rows, other.cols);
    int innerDim = this.cols;

    for (int thisR = 0; thisR < this.rows; thisR++) {
        for (int otherC = 0; otherC < other.cols; otherC++) {
            for (int i = 0; i < innerDim; i++) {
                newMatrix.values[thisR][otherC] = newMatrix.values[thisR][otherC]
                        .add(this.values[thisR][i].multiply(other.values[i][otherC]));
            }
        }
    }

    return newMatrix;
}
```

Matrix Chain Multiplication Problem



- ➢ Optimization problem solved with **Dynamic Programming**
- ➢ Order in which the matrices are parenthesized affects the number of simple arithmetic operations needed to compute the product.
- ➢ Solution finds the optimal way to multiply a given sequence of matrices

Why optimize?

Matrix A: 10×30
Matrix B: 30×5
Matrix C: 5×60

$$(AB)C : (10 \times 30 \times 5) + (10 \times 5 \times 60) = 4500$$

$$A(BC) : (30 \times 5 \times 60) + (10 \times 30 \times 60) = 27000$$

Matrix multiplication is associative!

The first parenthesization is 6x more efficient than the other in terms of number of multiplication operations.

Optimization Sequential Algorithm

```java
// https://home.cse.ust.hk/~dekai/271/notes/L12/L12.pdf
protected int[][] getMinimumOrdering(int[] dims, int N) {
    int[][] dp = new int[N + 1][N + 1];
    int[][] s = new int[N + 1][N + 1];
    // l is the length of each matrix chain subproblem we tackle this iteration
    // Start smaller so that when we divide larger chains, answer is already done
    for (int l = 2; l <= N; l++) {
        for (int i = 1; i <= N - l + 1; i++) {
            int j = i + l - 1;
            dp[i][j] = Integer.MAX_VALUE;
            // For each possible partition point in this chain
            for (int k = i; k <= j - 1; k++) {
                // Total operations if we use this partition is the cost of multing left and
                // right + cost of combining at partition
                int q = dp[i][k] + dp[k + 1][j] + dims[i - 1] * dims[k] * dims[j];
                if (q < dp[i][j]) {
                    dp[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }

    // Print minimum number of operations required to out file
    System.out.println(dp[1][N]);
    return s;
}
```

Parallelization

Parallel Objectives

...

Explore methods of parallelization of the DP Optimization Algorithm

Efficiently multiply matrix pairs with a speedup compared to the standard sequential algorithm.

Implementation: Matrix Chain Optimization

- Use two threads to calculate subproblems for chains of length 2 -> N
  - Wait on
  - When using `dp[i][k] + dp[k + 1][j]` to calculate the total operations for each k, wait for previous thread to calculate either value if it is still null
    - Ignore if l == 2, and just use zero instead
  - Poll for a new value of l and start again

Implementation: Matrix Chain Optimization

Proposed implementation:

- Initialize two threads
- Each thread polls the next lowest matrix length uncalculated
- Conduct standard DP algorithm using this value of l
- When using `dp[i][k] + dp[k + 1][j]` to calculate the total operations for each k, wait for previous thread to calculate either value if it is still null
    - Ignore if l == 2, and just use zero instead
- Poll for a new value of l and start again

Why This Is Infeasible

- Cost outweighs benefit unless chain length is thousands long
  - This is unrealistic
  - Resultant matrices would hold enormous values
- Much smaller problem than actual matrix computations

```java
@Override
public void run() {
    while (true) {
        // Grab the next available length on the queue
        int l;
        try {
            l = spQueue.poll();
        } catch (NullPointerException e) {
            // If all subproblem lengths have been calculated, stop
            return;
        }
        for (int i = 1; i <= N - l + 1; i++) {
            int j = i + l - 1;
            for (int k = i; k <= j - 1; k++) {
                while (dp[i][k] == null && l > 2) {
                }
                while (dp[k + 1][j] == null && l < 2) {
                }
                BigInteger v1 = dp[i][k] == null ? BigInteger.ZERO : dp[i][k];
                BigInteger v2 = dp[k + 1][j] == null ? BigInteger.ZERO : dp[k + 1][j];
                BigInteger q = v1.add(v2)
                        .add(BigInteger.valueOf(dims[i - 1] * dims[k] * dims[j]));
                if (k == i || q.compareTo(dp[i][j]) < 0) {
                    dp[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }

}
```

Implementation: Matrix Multiplication

- 1st Task: How to split up matrix chain multiplication work across threads?
  - One approach: Split up each matrix pair multiplication on different threads
    - BUT Conflicts with efficient parenthesization of chain
      - Ex: A(B(CD))
  - **Final Approach:** Split up the row/column multiplication work in single matrix pair multiplication

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}\begin{bmatrix} j & k \\ l & m \\ n & o \end{bmatrix} = \begin{bmatrix} aj + bl + cn & ak + bm + co \\ dj + el + fn & dk + em + fo \\ gj + hl + in & gk + hm + io \end{bmatrix}$$

Thread 1 Thread 4

Thread 2 Thread 5

Thread 3 Thread 6

Implementation: Matrix Multiplication

- 2nd Task: How to parallelize multiplication between two matrices?
  - Use/reuse thread pool with fixed number of threads for each matrix pair multiplication
  - Steps:
    - Create empty result matrix
    - Assign tasks to thread pool to populate result matrix
    - Each task responsible for assigning value to cell (i, j) in result matrix
    - Wait until all tasks have finished
    - Return result matrix that was populated by different threads

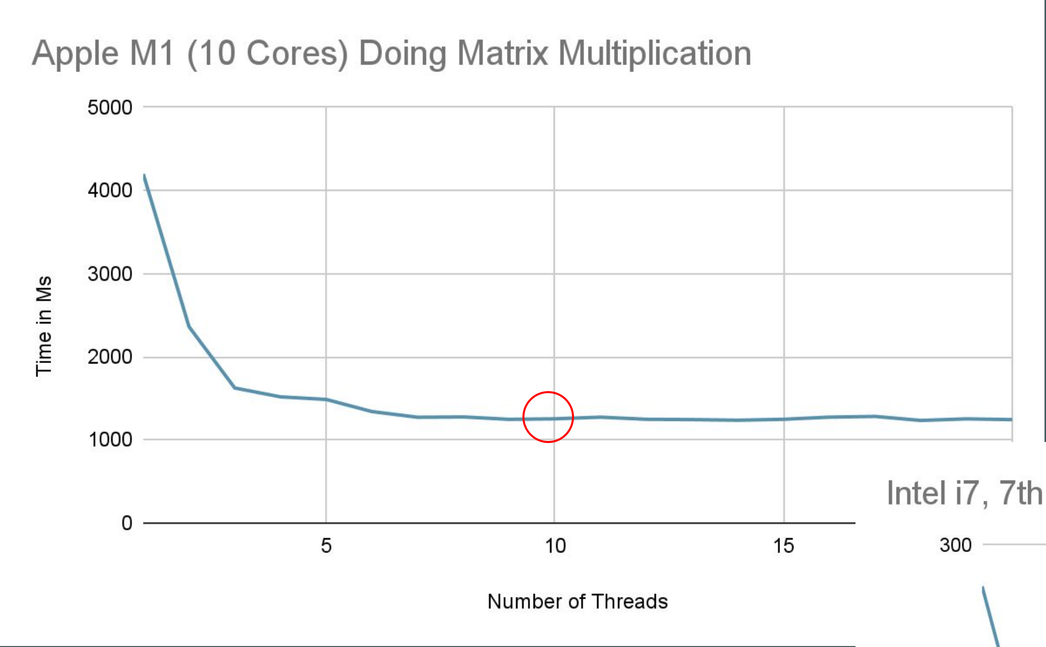
```
class MultThread implements Runnable {
    Matrix a;
    Matrix b;
    Matrix result;
```

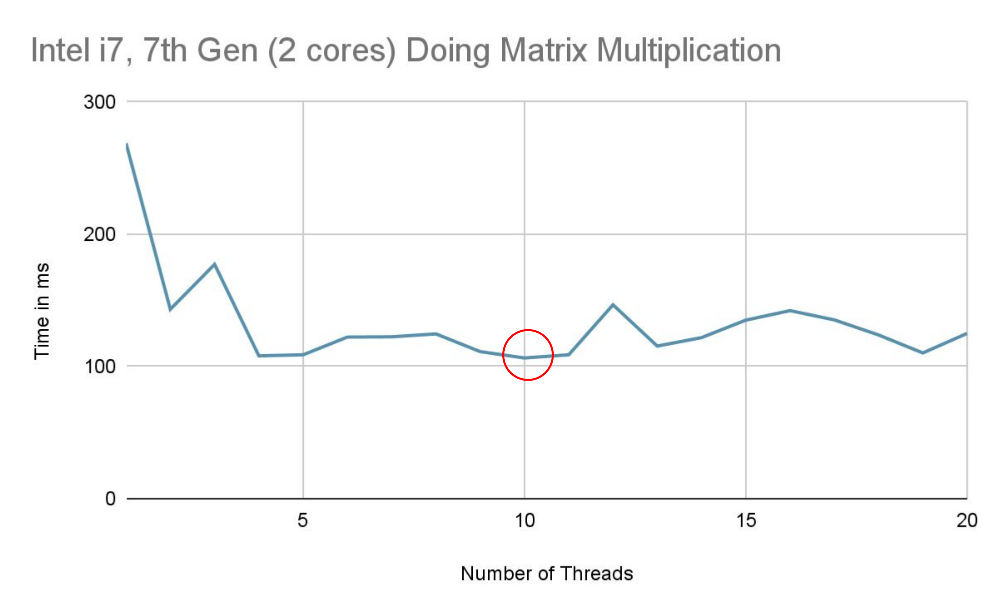
Implementation: Testing & Matrix Generation

- Created program to randomly generate matrices
- Finding Optimal Thread Count
  - Test program on same large test case (10-20 matrices, 1000 x 1000 max dimension)
  - Repeatedly run program on this test case but change number of threads used by thread pool each time
  - Pick the smallest thread count that achieves the best runtime and use for the rest of testing

```java
public class MatrixGenerator {
    static final int MAX_COLS = 100;
    static final int MAX_ROWS = 100;
    static final int MAX_VALUE = 100;
    static final int MIN_MATRICES = 50;
    static final int MAX_MATRICES = 50;
    static final String outPath = "in/";

    public static void main(String[] args) throws FileNotFoundException {
        int fileNum = 1;
        File f;
        do {
            f = new File(outPath + "test" + fileNum + ".txt");
            fileNum++;
        } while (f.exists() && !f.isDirectory());

        PrintStream stream = new PrintStream(f);
        System.setOut(stream);

        int lastC = -1;

        int numMatrices = (int) (Math.random() * (MAX_MATRICES - MIN_MATRICES + 1)) + MIN_MATRICES;
        System.out.println(numMatrices);
        for (int i = 0; i < numMatrices; i++) {
            int tr = (lastC != -1 ? lastC : (int) (Math.random() * MAX_ROWS + 1));
            int tc = (int) (Math.random() * MAX_COLS + 1);

            System.out.println(tr + " " + tc);
            for (int r = 0; r < tr; r++) {
                for (int c = 0; c < tc; c++) {
                    int val = (int) (Math.random() * (MAX_VALUE + 1));
                    System.out.print(val + (c < tc - 1 ? " " : ""));
                }
                System.out.println();
            }

            lastC = tc;
        }
    }
}
```

Thread Results





Note: Different Test Cases used, M1 using a much larger test case

Implementation: Testing & Matrix Generation

- 3 approaches to test
  - Brute force
  - Efficient ordering on single thread
  - Efficient ordering on multiple thread
- Testing Validity

```java
long bruteForceRuntimes = 0;
long syncSmartOrderingRuntimes = 0;
long multiThreadedSmartOrderingRuntimes = 0;

// Run multiple trials for each approach on same test case
for (int i = 0; i < NUM_PERFORMANCE_TRIALS; i++) {
    // Multiply chain out using brute force approach and record runtime
    long startTimeBruteForce = System.currentTimeMillis();
    Matrix resBruteForce = chain.multiplyOutBruteForce();
    long endTimeBruteForce = System.currentTimeMillis();
    bruteForceRuntimes += endTimeBruteForce - startTimeBruteForce;

    // Multiply chain out using efficient ordering on single thread approach and record
    runtime
    long startTimeSyncSmartOrdering = System.currentTimeMillis();
    Matrix resSyncSmartOrdering = chain.multiplyOut(useMultipleThreads:false);
    long endTimeSyncSmartOrdering = System.currentTimeMillis();
    syncSmartOrderingRuntimes += endTimeSyncSmartOrdering - startTimeSyncSmartOrdering;

    // Multiply chain out using efficient ordering and multiple threads approach and
    record runtime
    long startTimeMultiThreadedSmartOrdering = System.currentTimeMillis();
    Matrix resMultiThreadedSmartOrdering = chain.multiplyOut(useMultipleThreads:true);
    long endTimeMultiThreadedSmartOrdering = System.currentTimeMillis();
    multiThreadedSmartOrderingRuntimes += endTimeMultiThreadedSmartOrdering -
    startTimeMultiThreadedSmartOrdering;

    if (!matricesAreEqual(resBruteForce, resSyncSmartOrdering,
    resMultiThreadedSmartOrdering)) {
        System.out.println("Incorrect results from matrix multiplication.");
        return;
    }
```

Brute Force

1 Thread

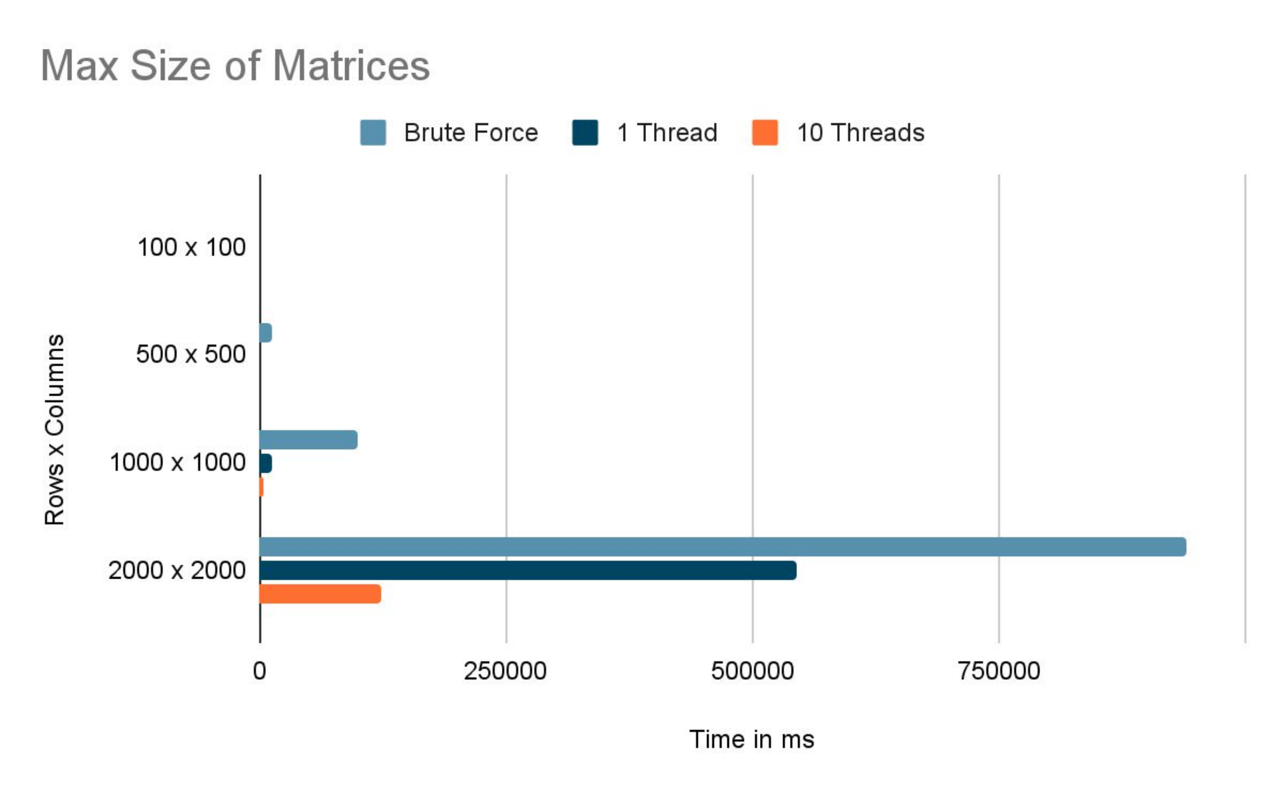
10 Threads

Check Results

Implementation: Testing & Matrix Generation

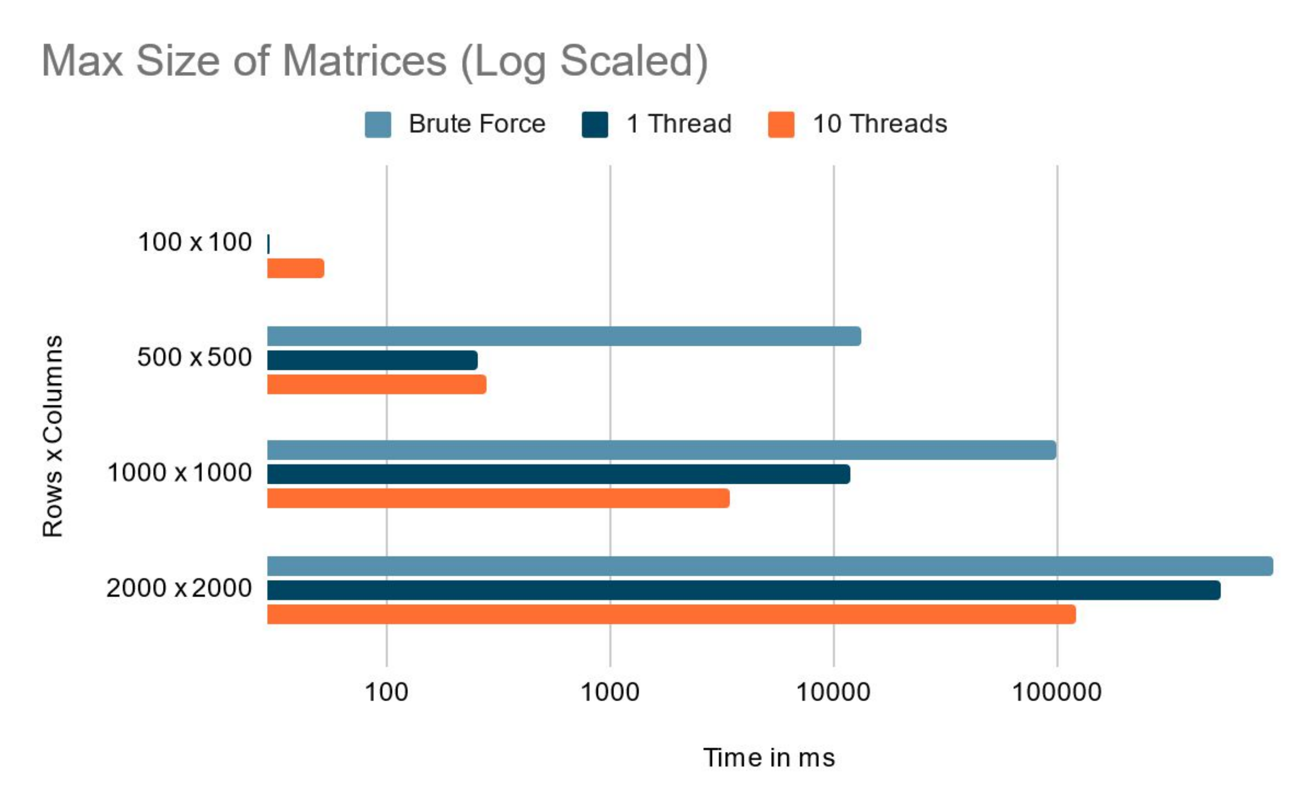
- Testing Performance
  - 2 independent variables: matrix chain size and individual matrix dimensions
  - Matrix Dimension Performance
    - Fix each matrix chain to size 8
    - Run trials on test cases with differing maximum matrix dimensions: (500 x 500), (1000 x 1000), (2000 x 2000)
  - Matrix Chain Size Performance
    - Fix max dimension of each matrix to (100 x 100)
    - Run trials on test cases with differing matrix chain lengths: 50, 100, 1000

Matrix Dimension Performance

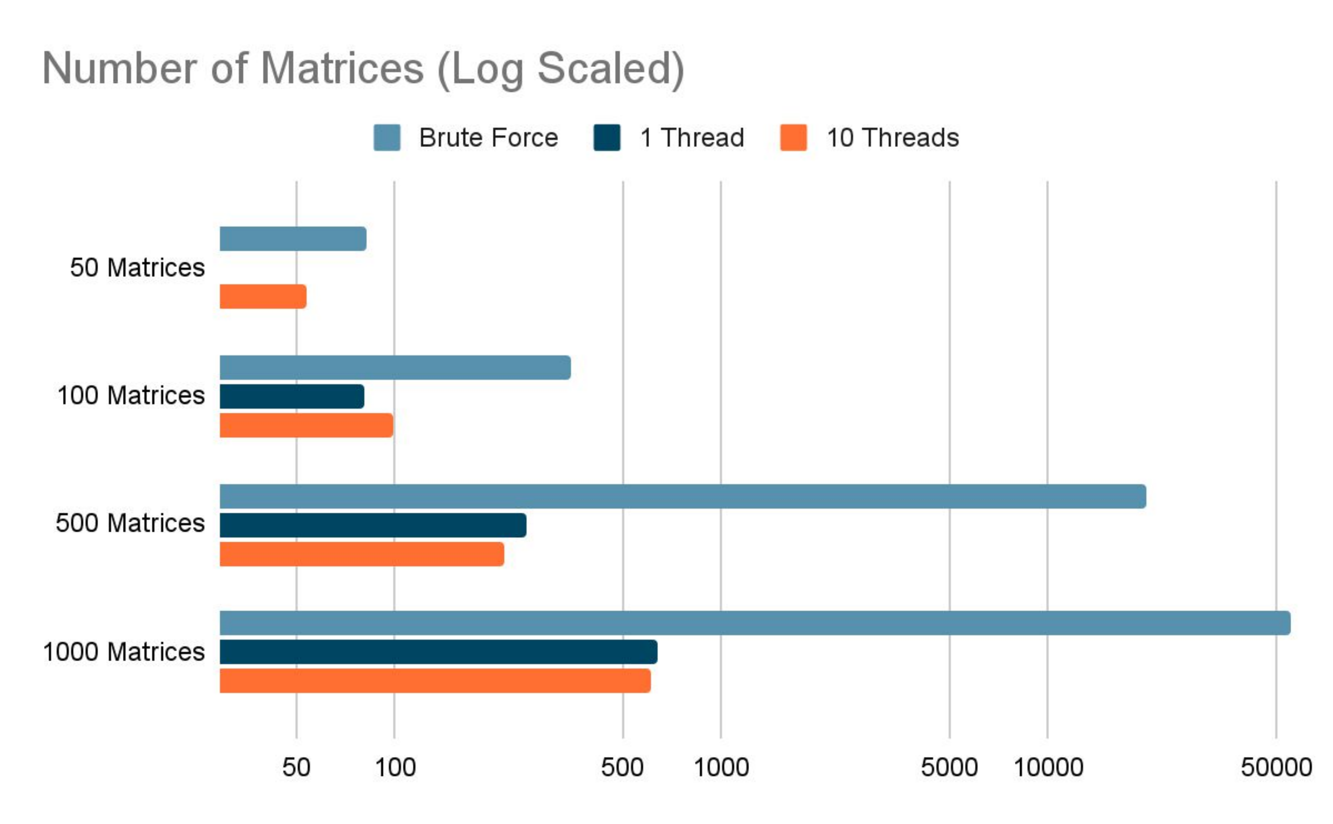


But wait! Brute Force hides all our data...

Matrix Dimension Performance (Log Scaled)



Matrix Chain Size Performance (Log Scaled)



Challenges

- Multiplication overflow
  - Converted matrices to use BigIntegers
- Identifying possible targets for parallelization
- Running large test cases on brute force approach

Conclusions

1. Stick to traditional DP optimization algorithm
2. Able to successfully parallelize matrix multiplication

Related Works for Further Study

- Hu, TC; Shing, MT (1981). Computation of Matrix Chain Products, Part I, Part II (PDF) (Technical report). Stanford University, Department of Computer Science. Part II, page 3. STAN-CS-TR-81-875.
- Wang, Xiaodong; Zhu, Daxin; Tian, Jun (April 2013). "Efficient computation of matrix chain". 2013 8th International Conference on Computer Science Education: 703–707. doi:10.1109/ICCSE.2013.6553999.
- Chin, Francis Y. (July 1978). "An O(n) algorithm for determining a near-optimal computation order of matrix chain products". Communications of the ACM. 21 (7): 544–549. doi:10.1145/359545.359556.
- Hu, T.C; Shing, M.T (June 1981). "An O(n) algorithm to find a near-optimum partition of a convex polygon". Journal of Algorithms. 2 (2): 122–138. doi:10.1016/0196-6774(81)90014-6