Implementation Of A Multithreaded Matrix Operations Program

Name: Swoyam Pokharel
Student Number: 2431342
Tutor: Bijaya Ghimire

Submitted On: December 19, 2025

Contents

SUMMARY

```
.
├── assets
│   ├── MatData.txt
│   └── Matrices.txt
├── outputs
│   ├── results_MatData.txt
│   ├── results_Matrices.txt
│   ├── Screenshot from 2025-12-19 09-01-08.png
│   ├── Screenshot from 2025-12-19 09-02-01.png
│   ├── Screenshot from 2025-12-19 09-02-08.png
│   ├── Screenshot from 2025-12-19 09-02-10.png
│   ├── Screenshot from 2025-12-19 09-02-12.png
│   ├── Screenshot from 2025-12-19 09-02-14.png
│   ├── Screenshot from 2025-12-19 09-02-19.png
│   ├── Screenshot from 2025-12-19 09-02-22.png
│   ├── Screenshot from 2025-12-19 09-02-32.png
│   ├── Screenshot from 2025-12-19 09-02-35.png
│   ├── Screenshot from 2025-12-19 09-02-39.png
│   └── Screenshot from 2025-12-19 09-02-43.png
├── perf_test
│   ├── perf.data
│   └── perf.txt
├── README.pdf
└── src
    ├── file_ops.c
    ├── file_ops.h
    ├── main.c
    ├── Makefile
    ├── matrix_ops.c
    └── matrix_ops.h

5 directories, 25 files
```

- **`README.pdf`** contains the explaination
- **`outputs/`** contains the `results_MatData.txt` & `results_Matrices.txt` generated by the program using the input datasets `MatData.txt` & `Matrices.txt`, along with screenshots
- **`assets/`** contains the provided input datasets.
- **`src/`** contains the actual source code
- **`perf_test/`** contains the results of the `perf record` command, along with the `perf.data` piped to a text file

1 · Program Architecture

The program is organized into three primary modules, each handling a distinct aspect of the application's functionality. The overall logic remains the same as the code that has been provided to us, but it improves upon error handling, modularization and implements functionality that the problem statement wants us to do.

1.1 · Module Structure

The codebase consists of six files organized into three logical units:

- **main.c**: Entry point and command-line interface
- **matrix_ops.c/h**: Matrix operations and memory management
- **file_ops.c/h**: File I/O and processing coordination

1.2 · Main Module

The main module serves as the program's entry point, handling command-line argument validation, file opening, and calling the rest of the functions:

```
int main(int argc, char* argv[]) {
    // Validate command line arguments
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <input_file> <num_threads>\n", argv[0]);
        fprintf(stderr, "Example: %s matrices.txt 4\n", argv[0]);
        return 1;
    }

    // Open input file
    FILE* in = fopen(argv[1], "r");
    if (!in) {
        fprintf(stderr, "Error: Cannot open input file '%s'\n", argv[1]);
        return 1;
    }

    // Parse thread count
    int threads = atoi(argv[2]);
    if (threads <= 0) {
        fprintf(stderr, "Error: Number of threads must be positive (got
%d)\n", threads);
        fclose(in);
        return 1;
    }

    // Open output file
    FILE* out = fopen("results.txt", "w");
    if (!out) {
        fprintf(stderr, "Error: Cannot create output file
'results.txt'\n");
        fclose(in);
        return 1;
    }
```

```
    // Process all matrix pairs
    int pairCount = processMatrixPairs(in, out, threads);

    // Cleanup
    fclose(in);
    fclose(out);

    // Report results
    printf("Processing complete. Results written to results.txt\n");
    printf("Processed %d matrix pair(s)\n", pairCount);

    return 0;
}
```

1.2.1 · Argument Validation

The program requires exactly two command line arguments: an input filename and a thread count.

- The first check verifies argument count. The `argc` variable includes the program name itself as `argv[0]`, so we need `argc >= 3` to have both required arguments.
- The second validation parses the thread count using `atoi`. This function converts a string to an integer, returning 0 if the string isn't a valid number.
- The check `threads <= 0` catches both explicit zero values and invalid inputs like "abc" or an empty string.

After all validations pass and files are open, the main function calls `processMatrixPairs`, passing the file handles and thread count. This function processes all matrix pairs found in the input file, returning the count of pairs successfully processed. The main function simply coordinates the high-level flow: validate inputs, open resources, call the processing function, close resources, report results.

2 · File Operations Module

The file operations module handles all interaction with the file system, including reading matrix data from the input file, validating parsed data, and writing computed results to the output file.

2.1 · Reading Matrices from Input File

The input file contains multiple matrices in a specific format, with each matrix preceded by dimension information.

2.1.1 · Input Format Structure

Each matrix in the file follows this pattern:

```
rows,cols
value1,value2,value3,...
value4,value5,value6,...
...
```

The dimensions appear on a single line as two integers seperated by a comma, followed by all matrix elements. Matrices are processed as pairs: the first matrix read becomes matrix A, the second becomes matrix B, then the third becomes the next pair's A, and so on.

Each dimension line and each value is terminated with a distinct character (newline for rows, comma for values), making it straightforward to parse with `fscanf`.

2.1.2 · The Processing Loop

The core processing function `processMatrixPairs` implements a loop that reads and processes matrix pairs until the input file is fully read:

```
int processMatrixPairs(FILE* in, FILE* out, int threads) {
    int pairNum = 0;

    while (1) {
        int r1, c1, r2, c2;

        // Try to read first matrix dimensions
        int result = fscanf(in, "%d,%d", &r1, &c1);
        if (result == EOF) {
            break;
        }

        if (result != 2) {
            fprintf(stderr, "Error: Invalid header format for matrix A in
pair %d\n", pairNum + 1);
            fprintf(stderr, "Expected format: rows,cols\n");
            break;
        }

        if (r1 <= 0 || c1 <= 0) {
            fprintf(stderr, "Error: Invalid dimensions %d,%d for matrix A
in pair %d\n", r1, c1, pairNum + 1);
            break;
        }

        double** A = allocMatrix(r1, c1);
        if (!A) {
            fprintf(stderr, "Error: Failed to allocate matrix A (%d x %d)
in pair %d\n", r1, c1, pairNum + 1);
            break;
        }

        if (!readMatrix(in, A, r1, c1, "A")) {
            freeMatrix(A, r1);
            break;
        }

        // Try to read second matrix dimensions
        result = fscanf(in, "%d,%d", &r2, &c2);
        if (result != 2) {
            fprintf(stderr, "Error: Invalid header format for matrix B in
```

```
pair %d\n", pairNum + 1);
            freeMatrix(A, r1);
            break;
        }

        if (r2 <= 0 || c2 <= 0) {
            fprintf(stderr, "Error: Invalid dimensions %d,%d for matrix B
in pair %d\n", r2, c2, pairNum + 1);
            freeMatrix(A, r1);
            break;
        }

        double** B = allocMatrix(r2, c2);
        if (!B) {
            fprintf(stderr, "Error: Failed to allocate matrix B (%d x %d)
in pair %d\n", r2, c2, pairNum + 1);
            freeMatrix(A, r1);
            break;
        }

        if (!readMatrix(in, B, r2, c2, "B")) {
            freeMatrix(A, r1);
            freeMatrix(B, r2);
            break;
        }

        pairNum++;
        performOperations(out, A, B, r1, c1, r2, c2, threads, pairNum);

        freeMatrix(A, r1);
        freeMatrix(B, r2);
    }

    if (pairNum == 0) {
        fprintf(out, "No valid matrix pairs found in input file.\n");
        fprintf(stderr, "Warning: No valid matrix pairs were processed\n");
    } else {
        fprintf(out, "Processed %d matrix pair(s) successfully.\n",
pairNum);
    }

    return pairNum;
}
```

Each iteration attempts to read one complete matrix pair, performing all operations on that pair before moving to the next.

2.1.3 · Dimension Parsing and Validation

Reading matrix dimensions is the first operation for each matrix. The `fscanf` function attempts to parse two integers separated by a comma:

```
int result = fscanf(in, "%d,%d", &r1, &c1);
if (result == EOF) {
    break; // End of file reached
}
if (result != 2) {
    fprintf(stderr, "Error: Invalid header format for matrix A in pair
%d\n",
            pairNum + 1);
    fprintf(stderr, "Expected format: rows,cols\n");
    break;
}
```

The return value from `fscanf` indicates how many values were successfully parsed. A return of `EOF` signals the end of the file, which is the normal termination condition for the processing loop. Any other return value that isn't exactly 2 indicates malformed input.

After successfully parsing dimensions, the program validates that they're positive and reasonable:

```
if (r1 <= 0 || c1 <= 0) {
    fprintf(stderr, "Error: Invalid dimensions %d,%d for matrix A in pair
%d\n",
            r1, c1, pairNum + 1);
    break;
}
```

This prevents attempting to allocate matrices with zero or negative dimensions, which would either fail or cause undefined behavior.

2.1.4 · Matrix Data Reading

Once dimensions are validated and memory is allocated, the `readMatrix` function handles reading the actual matrix values:

```
int readMatrix(FILE* in, double** matrix, int r, int c, const char* name) {
    int elementsRead = 0;

    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            if (fscanf(in, "%lf,", &matrix[i][j]) != 1) {
                fprintf(stderr, "Error: Failed to read element [%d][%d] of
matrix %s\n",
                        i, j, name);
                fprintf(stderr, "Expected %d total elements, only read
%d\n",
                        r * c, elementsRead);
                return 0;
            }
            elementsRead++;
        }
    }
```

```
    if (elementsRead != r * c) {
        fprintf(stderr, "Error: Matrix %s dimension mismatch. Expected %d
elements, read %d\n",
                name, r * c, elementsRead);
        return 0;
    }

    return 1;
}
```

This function reads values in order, filling each row completely before moving to the next. The nested loops iterate through all expected positions, and `fscanf` attempts to parse a double followed by a comma at each position.

The format string `"%lf,"` tells `fscanf` to skip any leading whitespace, parse a floating-point number, and expect and consume a comma. This format matches the input file structure exactly, where every value is terminated with a comma.

The element counter provides an additional validation layer. Even if all individual `fscanf` calls succeed, comparing the final count against the expected total `r * c` catches issues like extra or missing values that might occur if the file format is wrong.

2.1.5 · Error Recovery and Memory Cleanup

When parsing fails for any matrix, the program must clean up any memory already allocated before terminating the processing loop:

```
double** A = allocMatrix(r1, c1);
if (!A) {
    fprintf(stderr, "Error: Failed to allocate matrix A (%d x %d) in pair
%d\n", r1, c1, pairNum + 1);
    break;
}

if (!readMatrix(in, A, r1, c1, "A")) {
    freeMatrix(A, r1);
    break;
}

// Similar for matrix B...
if (!readMatrix(in, B, r2, c2, "B")) {
    freeMatrix(A, r1);
    freeMatrix(B, r2);
    break;
}
```

This pattern appears throughout the processing loop. If reading matrix A fails, we free A and break out of the loop. If reading matrix B fails, we free both A and B before breaking. This ensures no memory leaks occur even when processing is interrupted by malformed input.

2.2 · Writing Results to Output File

The output file format mirrors the input structure but includes clear labels and separators to make results easy to read. Each matrix pair gets its own section with all applicable operations performed and their results displayed.

2.2.1 · Operation Execution and Output

The `performOperations` function coordinates all operations for a single matrix pair:

```
void performOperations(FILE* out, double** A, double** B, int r1, int c1,
int r2, int c2, int threads, int pairNum) {
    fprintf(out, "================================\n");
    fprintf(out, "MATRIX PAIR %d\n", pairNum);
    fprintf(out, "Matrix A: %d x %d\n", r1, c1);
    fprintf(out, "Matrix B: %d x %d\n", r2, c2);
    fprintf(out, "================================\n\n");

    // Element-wise operations (require same dimensions)
    if (r1 == r2 && c1 == c2) {
        fprintf(out, "Addition - %d,%d\n", r1, c1);
        double** R = add(A, B, r1, c1, threads);
        if (R) {
            printMatrix(out, R, r1, c1);
            freeMatrix(R, r1);
        } else {
            fprintf(out, "Error: Memory allocation failed\n");
        }

        // Similar for subtraction, element-wise multiply, element-wise
divide
    } else {
        fprintf(out, "Addition cannot be done - different sizes\n");
        fprintf(out, "Subtraction cannot be done - different sizes\n");
        fprintf(out, "Element-wise Multiply cannot be done - different
sizes\n");
        fprintf(out, "Element-wise Divide cannot be done - different
sizes\n");
    }

    // Transpose operations (always valid)
    fprintf(out, "\nTranspose A - %d,%d\n", c1, r1);
    double** T = transpose(A, r1, c1, threads);
    if (T) {
        printMatrix(out, T, c1, r1);
        freeMatrix(T, c1);
    }

    // Matrix multiplication (requires c1 == r2)
    if (c1 == r2) {
        fprintf(out, "\nMatrix Multiply A x B - %d,%d\n", r1, c2);
        double** R = matMul(A, B, r1, c1, c2, threads);
        if (R) {
```

```
            printMatrix(out, R, r1, c2);
            freeMatrix(R, r1);
        }
    } else {
        fprintf(out, "\nMatrix Multiply cannot be done;  A columns (%d) !=
B rows (%d)\n", c1, r2);
    }
}
```

The function begins with a header showing the pair number and dimensions of both matrices. This header provides context for all subsequent operations, making it immediately clear which matrices are being processed.

Each operation follows a consistent pattern: check if the operation is valid based on dimensions, attempt to perform it if valid, check if the result matrix was successfully allocated, print the result if successful, and free the result immediately after printing. This pattern ensures consistent handling across all operations and prevents memory leaks.

2.2.2 · Output Format Compliance

The output format follows the problem's specification. Each operation result begins with a line showing the operation name and result dimensions in the format `"Operation - rows,cols"`:

```
fprintf(out, "Addition - %d,%d\n", r1, c1);
```

When an operation cannot be performed, a single-line message explains why:

```
fprintf(out, "Addition cannot be done - different sizes\n");
```

2.2.3 · Matrix Printing

The `printMatrix` function handles the actual output formatting:

```
void printMatrix(FILE* f, double** M, int r, int c) {
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            fprintf(f, "%lf", M[i][j]);
            if (j < c - 1) fprintf(f, ", ");
        }
        fprintf(f, "\n");
    }
}
```

This function iterates through rows and columns, printing each value with `%lf` format, which displays doubles with full precision. Commas separate values within a row, but the last value in each row gets no trailing comma. Each row ends with a newline, creating the matrix layout where each row appears on its own line.

3 · Matrix Operations Module

The matrix operations module implements all computational functionality, including memory management for matrices and the seven required matrix operations. Each operation is parallelized using OpenMP to distribute work across multiple threads.

3.1 · Memory Management

Matrices are stored as arrays of row pointers, where each row pointer points to an array of doubles.

3.1.1 · Dynamic Array Storage

The `allocMatrix` function allocates memory for a matrix with the specified dimensions:

```
double** allocMatrix(int r, int c) {
    if (r <= 0 || c <= 0) {
        fprintf(stderr, "Error: Invalid matrix dimensions %d x %d\n", r,
c);
        return NULL;
    }

    double** m = malloc(r * sizeof(double*));
    if (!m) {
        fprintf(stderr, "Error: Memory allocation failed for matrix
rows\n");
        return NULL;
    }

    for (int i = 0; i < r; i++) {
        m[i] = malloc(c * sizeof(double));
        if (!m[i]) {
            fprintf(stderr, "Error: Memory allocation failed for matrix row
%d\n", i);
            for (int j = 0; j < i; j++)
                free(m[j]);
            free(m);
            return NULL;
        }
    }
    return m;
}
```

The function begins by validating dimensions, rejecting zero or negative values that would represent errors. The validation occurs before any allocation attempts.

Memory allocation occurs in two stages. First, we allocate an array of `r` pointers, where each pointer will point to a row. Then, we allocate `c` doubles for each row. If allocating the pointer array fails, we return `NULL` immediately. If allocating any individual row fails, we must clean up all previously allocated rows before returning `NULL`. This cleanup happens in the failure path itself:

```
for (int j = 0; j < i; j++)
    free(m[j]);
```

```
free(m);
return NULL;
```

3.1.2 · Deallocation

The `freeMatrix` function deallocates a matrix in the reverse order of allocation:

```
void freeMatrix(double** m, int r) {
    if (!m) return;
    for (int i = 0; i < r; i++)
        free(m[i]);
    free(m);
}
```

The null check at the beginning makes this function safe to call on failed allocations. If `allocMatrix` returns `NULL` due to an error, calling `freeMatrix` on that `NULL` pointer does nothing. Each row must be freed individually because each was allocated with its own `malloc` call. Only after freeing all rows can the row pointer array itself be freed. This is important because freeing the pointer array first would lose access to the row pointers, making it impossible to free the row memory and causing a memory leak.

3.2 · Thread Capping

The assignment requires capping the actual thread count to prevent using more threads than the problem can effectively parallelize. This optimization prevents wasteful thread creation when the workload is small relative to the requested thread count.

3.2.1 · The Capping Function

```
int capThreads(int requested, int row) {
    if (requested > row) return row;
    return requested;
}
```

- If the user requests more threads than there are rows, cap to the number of rows. For a 3*100 matrix, using more than 3 threads provides no benefit since the parallel loop only has 3 iterations to distribute.

3.2.2 · Application in Operations

Each operation calls `capThreads` before creating the parallel region:

```
int actualThreads = capThreads(threads, r);

#pragma omp parallel for num_threads(actualThreads)
// ... operation logic
```

3.3 · Matrix Operations

The program implements seven distinct matrix operations, each parallelized using OpenMP. The operations handle matrices of arbitrary size, with dimension checking determining which operations are valid for each pair.

3.3.1 · Element-wise Operations

Element-wise operations require both input matrices to have identical dimensions.

Addition

```
double** add(double** A, double** B, int r, int c, int threads) {
    double** R = allocMatrix(r, c);
    if (!R) return NULL;

    int actualThreads = capThreads(threads, r);

    #pragma omp parallel for num_threads(actualThreads)
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            R[i][j] = A[i][j] + B[i][j];

    return R;
}
```

The function begins by allocating the result matrix with the same dimensions as the inputs. The null check allows graceful handling of allocation failures by returning `NULL` to the caller, who can then decide how to proceed.

Thread capping occurs before the parallel region, adjusting the requested thread count to match the number of rows. The `#pragma omp parallel for` directive tells the compiler to parallelize the outer loop over rows, with OpenMP automatically dividing work among threads.

The actual computation `R[i][j] = A[i][j] + B[i][j]` is straightforward: each result element is the sum of the corresponding input elements. No synchronization is needed because each thread writes to distinct memory locations in the result matrix. Different threads process different rows, so their writes never conflict.

Subtraction, Element-wise Multiplication

Subtraction and element-wise multiplication follow the identical structure to addition, differing only in the operation performed:

```
// Subtraction
R[i][j] = A[i][j] - B[i][j];

// Element-wise multiplication
R[i][j] = A[i][j] * B[i][j];
```

All three operations share the same parallelization strategy and error handling, differing only in their actual operation.

Element-wise Division

Division includes special handling for division by zero:

```
double** elemDiv(double** A, double** B, int r, int c, int threads) {
    double** R = allocMatrix(r, c);
    if (!R) return NULL;

    int actualThreads = capThreads(threads, r);

    #pragma omp parallel for num_threads(actualThreads)
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            if (B[i][j] == 0)
                R[i][j] = 0.0 / 0.0;  // NaN
            else
                R[i][j] = A[i][j] / B[i][j];
        }
    }

    return R;
}
```

When the divisor is zero, the result is set to NaN (not a number) using the expression `0.0 / 0.0`. Since C doesn't have a `NaN` as a built in literal, this produces an IEEE 754 NaN value, which propagates through subsequent calculations and is recognizable in the output as "nan" or "-nan".

The alternative approaches of skipping zero divisors or aborting processing would either leave result positions uninitialized or prevent computing other valid divisions. Storing NaN allows the operation to complete while clearly marking problematic positions.

3.3.2 · Transpose

Transposition swaps rows and columns, transforming an r*c matrix into a c*r matrix:

```
double** transpose(double** A, int r, int c, int threads) {
    double** T = allocMatrix(c, r);
    if (!T) return NULL;

    int actualThreads = capThreads(threads, r);

    #pragma omp parallel for num_threads(actualThreads)
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            T[j][i] = A[i][j];

    return T;
}
```

The transpose operation is always valid regardless of matrix dimensions. A 2*5 matrix transposes to a 5*2 matrix, and even a 1*100 matrix transposes to a 100*1 matrix.

3.3.3 · Matrix Multiplication

Matrix multiplication is the most computationally intensive operation, requiring $O(n^3)$ time for n*n matrices compared to $O(n^2)$ for element-wise operations:

```
double** matMul(double** A, double** B, int rA, int cA, int cB, int
threads) {
    double** R = allocMatrix(rA, cB);
    if (!R) return NULL;

    int actualThreads = capThreads(threads, rA);

    #pragma omp parallel for num_threads(actualThreads)
    for (int i = 0; i < rA; i++)
        for (int j = 0; j < cB; j++) {
            double sum = 0;
            for (int k = 0; k < cA; k++)
                sum += A[i][k] * B[k][j];
            R[i][j] = sum;
        }

    return R;
}
```

Matrix multiplication requires A's column count to equal B's row count. When multiplying an rA*cA matrix by a cA*cB matrix, the result is rA*cB. Each element of the result is computed as a dot product: `R[i][j]` = ∑`(A[i][k] * B[k][j])`.

4 · Compilation and Build System

The program uses a Makefile to manage compilation :

4.1 · Makefile Structure

```
CC = gcc
CFLAGS = -Wall -Wextra -O2 -fopenmp

TARGET = matrix_program

all:
  $(CC) $(CFLAGS) *.c -o $(TARGET)

clean:
  rm -f $(TARGET)

.PHONY: all clean
```

4.2 · Compilation Flags

The `-Wall -Wextra` flags enable comprehensive warning detection. These warnings catch common mistakes like unused variables, missing return statements, and so on.

The `-O2` optimization level provides significant performance improvements for numerical computations. The compiler performs various optimizations like loop unrolling and function inlining.

4.3 · Build Commands

To build the program:

```
make
```

To clean build artifacts:

```
make clean
```

This removes, the executable, and the results file, allowing a fresh build from scratch.

5 · Usage and Input Format

5.1 · Running the Program

The program requires two command-line arguments:

```
./matrix_program input.txt 4
```

Where:

- `input.txt` is the file containing matrix pairs
- `4` is the number of threads to use

The thread count should generally match the number of CPU cores available. Using more threads than cores provides diminishing returns, as threads must share CPU time through context switching. Using fewer threads than cores leaves processing power unused.

6 · Improvements Over Original Code

The original provided code was functional but lacked in some ways ways. Several changes were done to improve upon it's foundation.

6.1 · Modular Architecture

The original code existed in a single file with all functionality intermixed. The improved version separates concerns into three distinct modules:

- **main.c**: Command-line interface and program coordination
- **matrix_ops.c/h**: Pure computational logic
- **file_ops.c/h**: I/O and data transformation

This separation provides numerous benefits. Each module can be tested independently. Changes to file format don't affect computational code. Different developers can work on different modules without conflicts. The code becomes easier to understand because each file has a single, clear purpose.

6.2 · Comprehensive Error Handling

The original code checked some error conditions but not others. The improved version validates comprehensively:

- Command-line argument count and values
- File opening success for both input and output
- Dimension header format and values
- Every matrix element read
- Memory allocation success
- Total elements read matches expected count

6.3 · Memory Safety

The original code assumed all allocations succeeded and didn't handle partial allocation failures. The improved version checks every allocation:

```
double** R = allocMatrix(r, c);
if (!R) return NULL;
```

6.4 · Thread Capping Implementation

The original code used the user-provided thread count directly without validation or limiting. The improved version implements the assignment-required `capThreads` function:

```
int capThreads(int requested, int dimension) {
    if (requested > dimension)
        return dimension > 0 ? dimension : 1;
    return requested > 0 ? requested : 1;
}
```

7 · Performance Analysis

7.1 · Profiling Results

The program was tested using the `MatData.txt` dataset that was provided, running on an Arch Linux system with x86-64 architecture. The profiling results provide insight into where the program spends its time and how effectively it utilizes multiple cores.

7.1.1 · Execution Time

```
[wizard@archlinux ~/Projects/School/HPC/Assessment/t2/code] -> time ./
matrix_program ../assets/MatData.txt 4
Processing complete. Results written to results.txt
Processed 25 matrix pair(s)
./matrix_program ../assets/MatData.txt 4  0.01s user 0.00s system 259% cpu
0.003 total
```

The execution completes in just 0.003 seconds of real time, with 0.01 seconds spent in user mode and essentially zero time in kernel mode. The CPU utilization of 259% indicates strong multi-core usage, with the program utilizing approximately 2.6 cores actively working. This is a significant improvement over the word counter program's 128% utilization.

With 4 threads, the theoretical maximum CPU utilization would be 400% if all four threads were continuously busy. The actual utilization of 259% is reasonable given that:

- File I/O operations (reading input, writing output) are sequential
- Some matrix operations may complete too quickly to fully utilize all threads
- Thread synchronization at OpenMP barriers introduces some overhead
- The dataset size may not provide enough work to keep all threads busy continuously

The higher utilization compared to the word counter (259% vs 128%) suggests that OpenMP's work distribution is more efficient than the manual pthread-based approach; granted it's not an apples to apples comparison, as one is counting words and another one is operating on matrices.

7.1.2 · CPU Profile (`perf report`)

Top functions by CPU time:

```
# Overhead  Command         Shared Object
Symbol
# ........ ............... ..................... ..............................
    84.73%  matrix_program  libgomp.so.1.0.0      [.] gomp_barrier_wait_end
     2.52%  matrix_program  libgomp.so.1.0.0      [.]
gomp_team_barrier_wait_end
     1.46%  matrix_program  libgomp.so.1.0.0      [.] gomp_init_task
     1.44%  matrix_program  libc.so.6             [.] 0x000000000005b010
     1.44%  matrix_program  libc.so.6             [.] 0x000000000005c036
     1.37%  matrix_program  libc.so.6             [.] 0x00000000000540df
     1.17%  matrix_program  libgomp.so.1.0.0      [.] gomp_team_start
     1.10%  matrix_program  libc.so.6             [.] 0x000000000005b690
     0.99%  matrix_program  libc.so.6             [.] 0x0000000000066d74
     0.74%  matrix_program  libc.so.6             [.] 0x000000000005bc47
     0.73%  matrix_program  libc.so.6             [.] 0x000000000006e033
     0.71%  matrix_program  libc.so.6             [.] _IO_file_xsputn
     0.59%  matrix_program  libc.so.6             [.] 0x000000000005b6d0
     0.48%  matrix_program  libc.so.6             [.] 0x0000000000111783
     0.16%  matrix_program  libgomp.so.1.0.0      [.] gomp_thread_start
     0.14%  matrix_program  matrix_program        [.] transpose._omp_fn.0
     0.14%  matrix_program  ld-linux-x86-64.so.2  [.] 0x0000000000008e85
     0.02%  matrix_program  ld-linux-x86-64.so.2  [.] 0x00000000000147a1
     0.01%  matrix_program  [unknown]             [k] 0xffffffffb4600087
     0.01%  matrix_program  libc.so.6             [.] 0x0000000000065838
     0.00%  matrix_program  libgomp.so.1.0.0      [.] gomp_barrier_wait
     0.00%  matrix_program  libc.so.6             [.] 0x0000000000096786
     0.00%  matrix_program  [unknown]             [k] 0xffffffffb460008f
```

Expected Results

- **Barrier Synchronization Dominates**: The gomp_barrier_wait_end function consuming 84.73% of total CPU time aligns with OpenMP's behavior for small workloads. This

represents implicit barriers at the end of each parallel region. With 25 matrix pairs and multiple operations per pair, threads synchronize frequently. The high percentage indicates that synchronization overhead dominates actual computation for this dataset size.

- **Team Management Overhead is Present**: The combination of `gomp_team_barrier_wait_end (2.52%)`, `gomp_init_task (1.46%)`, and `gomp_team_start (1.17%)` totaling roughly 5.15% represents OpenMP's thread team management. This is unavoidable overhead as the program creates and destroys parallel regions for each operation on each matrix pair. The percentage is reasonable given the number of parallel regions created.
- **File I/O is Minimal**: The `_IO_file_xsputn` consuming only 0.71% indicates that output writing is not a bottleneck. This low overhead makes sense because the program writes relatively small amounts of formatted text, and the sequential I/O operations complete quickly compared to the parallelization overhead.

Unexpected Findings

- **Computation Time is Nearly Invisible**: The `transpose._omp_fn.0` function appears at only 0.14% of total time. This is surprisingly low and indicates that actual matrix operations complete so quickly that they barely register in the profile. The profiler samples captured almost entirely synchronization overhead rather than computational work. This suggests the dataset matrices are too small to benefit meaningfully from parallelization; the overhead of creating parallel regions and synchronizing threads exceeds the time saved by parallel execution.
- **High CPU Usage Despite Barrier Dominance**: The 259% CPU utilization seems contradictory with 84.73% barrier time, but this occurs because OpenMP uses active waiting (busy-spinning) rather than passive waiting. Threads consume CPU cycles spinning on barriers waiting for other threads to complete, which counts as CPU usage but accomplishes no useful work. This explains high CPU utilization without corresponding computational progress.
- **Kernel Time is Negligible**: The near-zero kernel time (0.00s) is somewhat surprising but indicates the program makes minimal system calls and doesn't block on I/O. OpenMP's user-space synchronization primitives avoid kernel involvement for most operations. This is excellent for avoiding context switch overhead but means threads are burning CPU cycles in user space during barrier waits.
- **Synchronization Overhead Dominates Computation**: With approximately 85% of time spent in barriers and only 0.14% in actual computation, the synchronization overhead is roughly 600x the computation time. This is a clear indication that for this particular dataset, sequential execution would be faster. The parallelization overhead has completely overwhelmed any potential speedup. This finding suggests implementing adaptive threading that uses sequential execution for small matrices and reserves parallelization for larger workloads where computation time would dominate synchronization overhead.