Complete Analysis of the task (what algorithm did you use? Why?)

The algorithm here takes two matrices as an input and always multiplies two elements inside of the matrix together using threads. For each row we have we will start a new thread. To ensure that we can display the results all at once we will wait for all threads to finish executing with the .join() function. Finally, we will simply output the results that we have saved in a separate row.

What other algorithms apply to this task?

Another algorithm that we could have applied here is a single threaded approach. Instead of using multi-threading like we have for this task we will just use a single thread since it's a simpler implementation. However, this may extend the run time of the code. And the amount of time it will take will only become longer as the scale of the problem increases.

In what ways does the algorithm you used differ from these other algorithms?

The algorithm that I have used here utilizes multiple threads instead of just once or even none. While being a more complex solution it will allow our code to compile much faster and will make it overall a more stable solution as we decrease the chance of the code potentially running into an error and completely stopping.

How does the code work? A detailed explanation of the functions made.)

The entire code includes comments that explain what each function performs.

Complexity analysis of the algorithm:

The time complexity of this program is O(n). The reason for this is because this code is linear. So if the number of multiplications that we have to perform increases so will our n. Since n is the total elements. Lastly, it doesn't matter if we use threads or not we will still have the same amount of computational work. It is just a matter of if we can speed up the process.

Code:

```
#include <iostream>
#include <thread>
#include <vector>
#include <iomanip>
using namespace std;

// This function here will multiply two numbers of both rows and then store the results in a
seperate row
void add_multipication_matrix_row(int* mat1, int* mat2, int* result, int cols) {
    for (int c = 0; c < cols; c++) {
        result[c] = mat1[c] * mat2[c];
    }
}

// Function to print a matrix with a label
void printMatrix(int* matrix, int rows, int cols, const string& label) {
    cout << label << endl;  // Print the label for the matrix
    for (int r = 0; r < rows; r++) {  // Loop through each row
        for (int c = 0; c < cols; c++) {  // Loop through each column in the row
            cout << setw(4) << matrix[r * cols + c] << " ";  // Print each element with formatted
width
        }
        cout << endl;  // New line after each row
    }
}

int main() {
    // Declaring a variable to hold the number of rows
    int rows = 3;
    // Declaring a variable to hold the number of columns
    int cols = 4;
    // Then we will create both of our matrices and one to store all of the results.
    int array1[3][4] = { {100, 20, 3, 50}, {40, 33, 56, 2}, {150, 23, 17, 22} };
    int array2[3][4] = { {56, 13, 9, 100}, {22, 15, 55, 60}, {19, 200, 27, 14} };
    int result[3][4];

    // Now we declare a vector to store the threads
    vector<thread> threads;

    // This for loop will start a thread for each calculation that is done
    for (int r = 0; r < rows; r++) {
        threads.push_back(thread(add_multipication_matrix_row, array1[r], array2[r], result[r],
cols));
    }
    // This for loop will iterate over every thread and wait for all of the threads to finish so we can
display all of the results
    // Since some threads may finish sooner we need to wait for all of them to finish
    for (auto& th : threads) {
        th.join();
    }

    // Now will will print out the all the rows including the result row.
    printMatrix(&array1[0][0], rows, cols, "Matrix 1:");
    cout << "         *\n";
    printMatrix(&array2[0][0], rows, cols, "Matrix 2:");
    cout << "         =\n";
    printMatrix(&result[0][0], rows, cols, "Result:");



}
```