CS3811 - High Performance Computing and Big Data Lab

Lab 2

> Name: M K Lokesh Kumar

> Registration No.: 2201113026

> Class: Cyber Security(Semester 5)

Experiment 1

Objective

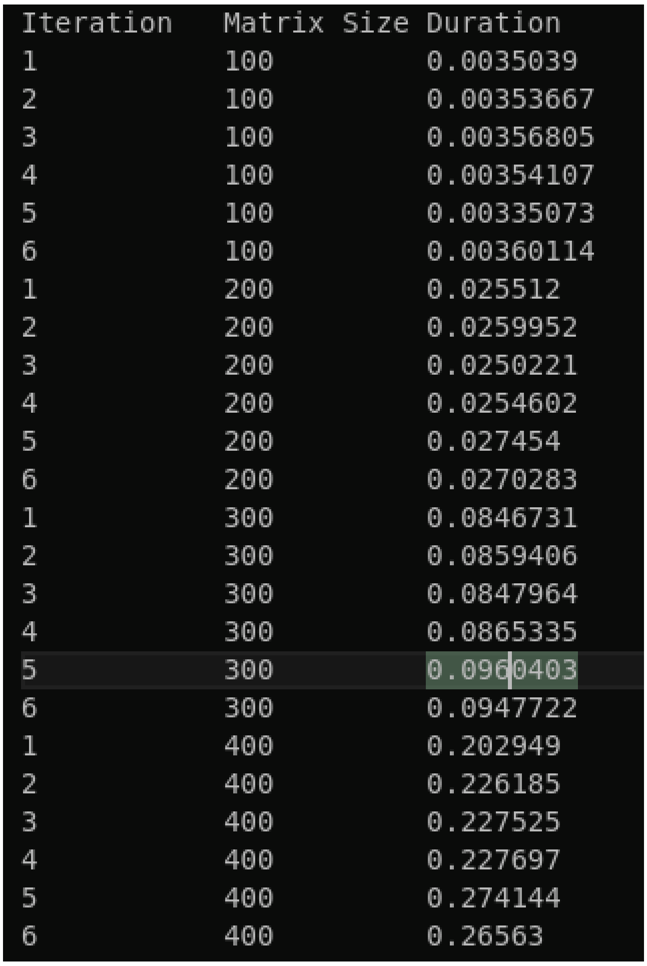
Write a C/C++ to find the optimal matrix multiplication ordering.

Code

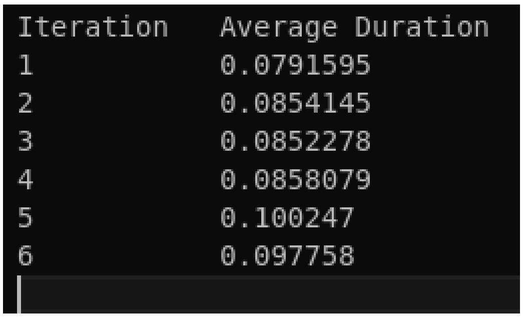
Written in C++.

```
#include <iostream>
#include <chrono>
#include <cstdlib>
#include <fstream>
#include <omp.h>
#include <map>
#include <vector>

using namespace std;
using namespace std::chrono;

int** gen_matrix(int size) {
    int** temp = new int*[size];
    for (int i = 0; i < size; i++){
        temp[i] = new int[size];
        for (int j = 0; j < size; j++) {
            temp[i][j] = 1;
        }
    }

    return temp;
}

void free_matrix(int** mat, int size) {
    for (int i = 0; i < size; i++) {
        delete[] mat[i];
    }
    delete[] mat;
}

double matrix_multiply(int** A, int** B, int** C, int matrix_size, int stat) {
    auto start = high_resolution_clock::now();

    for (int i = 0; i < matrix_size; i++) {
        for (int j = 0; j < matrix_size; j++) {
            C[i][j] = 0;
            for (int k = 0; k < matrix_size; k++) {
                switch(stat) {
                    case 1:
                    C[i][j] += A[i][k] * B[k][j];
                    break;

                    case 2:
                    C[i][k] += A[i][j] * B[j][k];
                    break;

                    case 3:
                    C[j][i] += A[j][k] * B[k][i];
                    break;

                    case 4:
                    C[j][k] += A[j][i] * B[i][k];
                    break;

                    case 5:
                    C[k][i] += A[k][j] * B[j][i];
                    break;

                    case 6:
                    C[k][j] += A[k][i] * B[i][j];
                    break;

                    default:
                    cout << "Invalid option" << endl;
                }
            }
        }
    }

    auto stop = high_resolution_clock::now();

    duration<double> duration = stop - start;

    return duration.count();
}

int main() {
    // Serial code
    map<int, vector<double>> iteration_durations;
    int iteration, matrix_size;
    double duration;

    ofstream outfile1("data_1_0.dat");
    outfile1 << "Iteration" << "\t" << "Matrix Size" << "\t" << "Duration"
<< endl;

    for (int size = 100; size <= 400; size += 100) {
        int** A = gen_matrix(size);
        int** B = gen_matrix(size);
        int** C = new int*[size];
        for (int i = 0; i < size; i++) {
            C[i] = new int[size];
        }
        for (int i = 1; i <= 6; i++) {
            double time = matrix_multiply(A, B, C, size, i);
            outfile1 << i << "\t\t\t" << size << "\t\t\t" << time << endl;
            iteration_durations[i].push_back(time);
        }

        free_matrix(A, size);
        free_matrix(B, size);
        free_matrix(C, size);
    }
    outfile1.close();

    ofstream outfile2("data_1_1.dat");
    outfile2 << "Iteration" << "\t" << "Average Duration" << endl;

    for (const auto& entry: iteration_durations) {
        int iter = entry.first;
        const vector<double>& durations = entry.second;

        double sum = 0.0;
        for (double d: durations) {
            sum += d;
        }
        double average = sum / durations.size();
        outfile2 << iter << "\t\t\t" << average << endl;
    }
    outfile2.close();

    return 0;
}
```

Output



```
Iteration    Matrix Size Duration
1            100         0.0035039
2            100         0.00353667
3            100         0.00356805
4            100         0.00354107
5            100         0.00335073
6            100         0.00360114
1            200         0.025512
2            200         0.0259952
3            200         0.0250221
4            200         0.0254602
5            200         0.027454
6            200         0.0270283
1            300         0.0846731
2            300         0.0859406
3            300         0.0847964
4            300         0.0865335
5            300         0.0960403
6            300         0.0947722
1            400         0.202949
2            400         0.226185
3            400         0.227525
4            400         0.227697
5            400         0.274144
6            400         0.26563
```



```
Iteration    Average Duration
1            0.0791595
2            0.0854145
3            0.0852278
4            0.0858079
5            0.100247
6            0.097758
```

Experiment 2

Objective

Write a C/C++ program to perform matrix-vector multiplication using parallelization.

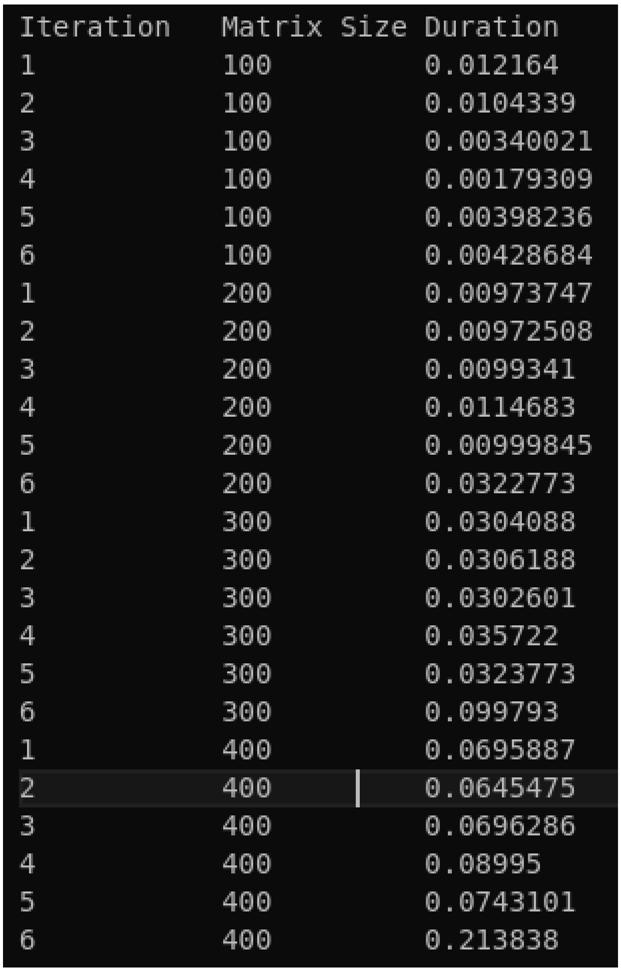
Code

Writen in C++.

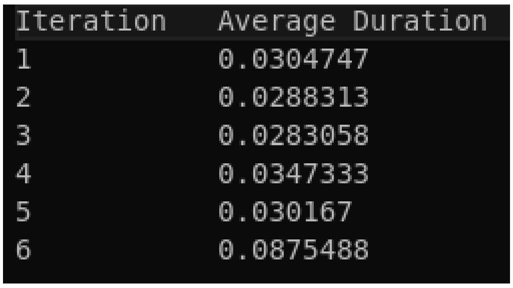
```
#include <iostream>
#include <chrono>
#include <cstdlib>
#include <fstream>
#include <omp.h>
```

```
#include <map>
#include <vector>

using namespace std;
using namespace std::chrono;

int** gen_matrix(int size) {
    int** temp = new int*[size];
    for (int i = 0; i < size; i++){
        temp[i] = new int[size];
        for (int j = 0; j < size; j++) {
            temp[i][j] = 1;
        }
    }

    return temp;
}

void free_matrix(int** mat, int size) {
    for (int i = 0; i < size; i++) {
        delete[] mat[i];
    }
    delete[] mat;
}

double matrix_multiply(int** A, int** B, int** C, int matrix_size, int
stat) {
    auto start = high_resolution_clock::now();

    #pragma omp parallel for
    for (int i = 0; i < matrix_size; i++) {

        #pragma omp parallel for
        for (int j = 0; j < matrix_size; j++) {
            C[i][j] = 0;

            #pragma omp parallel for
            for (int k = 0; k < matrix_size; k++) {
                switch(stat) {
                    case 1:
                    C[i][j] += A[i][k] * B[k][j];
                    break;

                    case 2:
                    C[i][k] += A[i][j] * B[j][k];
                    break;

                    case 3:
                    C[j][i] += A[j][k] * B[k][i];
                    break;

                    case 4:
                    C[j][k] += A[j][i] * B[i][k];
                    break;
                    case 5:
                    C[k][i] += A[k][j] * B[j][i];
                    break;

                    case 6:
                    C[k][j] += A[k][i] * B[i][j];
                    break;

                    default:
                    cout << "Invalid option" << endl;
                }
            }
        }
    }

    auto stop = high_resolution_clock::now();

    duration<double> duration = stop - start;

    return duration.count();
}

int main() {
    // Serial code
    map<int, vector<double>> iteration_durations;
    int iteration, matrix_size;
    double duration;

    ofstream outfile1("data_2_0.dat");
    outfile1 << "Iteration" << "\t" << "Matrix Size" << "\t" << "Duration" << endl;


    for (int size = 100; size <= 400; size += 100) {
        int** A = gen_matrix(size);
        int** B = gen_matrix(size);
        int** C = new int*[size];
        for (int i = 0; i < size; i++) {
            C[i] = new int[size];
        }
        for (int i = 1; i <= 6; i++) {
            double time = matrix_multiply(A, B, C, size, i);
            outfile1 << i << "\t\t\t" << size << "\t\t\t" << time << endl;
            iteration_durations[i].push_back(time);
        }

        free_matrix(A, size);
        free_matrix(B, size);
        free_matrix(C, size);
    }
    outfile1.close();

    ofstream outfile2("data_2_1.dat");
    outfile2 << "Iteration" << "\t" << "Average Duration" << endl;

    for (const auto& entry: iteration_durations) {
        int iter = entry.first;
        const vector<double>& durations = entry.second;

        double sum = 0.0;
        for (double d: durations) {
            sum += d;
        }
        double average = sum / durations.size();
        outfile2 << iter << "\t\t\t" << average << endl;
    }
    outfile2.close();

    return 0;
}
```

Output



```
Iteration   Matrix Size Duration
1           100         0.012164
2           100         0.0104339
3           100         0.00340021
4           100         0.00179309
5           100         0.00398236
6           100         0.00428684
1           200         0.00973747
2           200         0.00972508
3           200         0.0099341
4           200         0.0114683
5           200         0.00999845
6           200         0.0322773
1           300         0.0304088
2           300         0.0306188
3           300         0.0302601
4           300         0.035722
5           300         0.0323773
6           300         0.099793
1           400         0.0695887
2           400         0.0645475
3           400         0.0696286
4           400         0.08995
5           400         0.0743101
6           400         0.213838
```



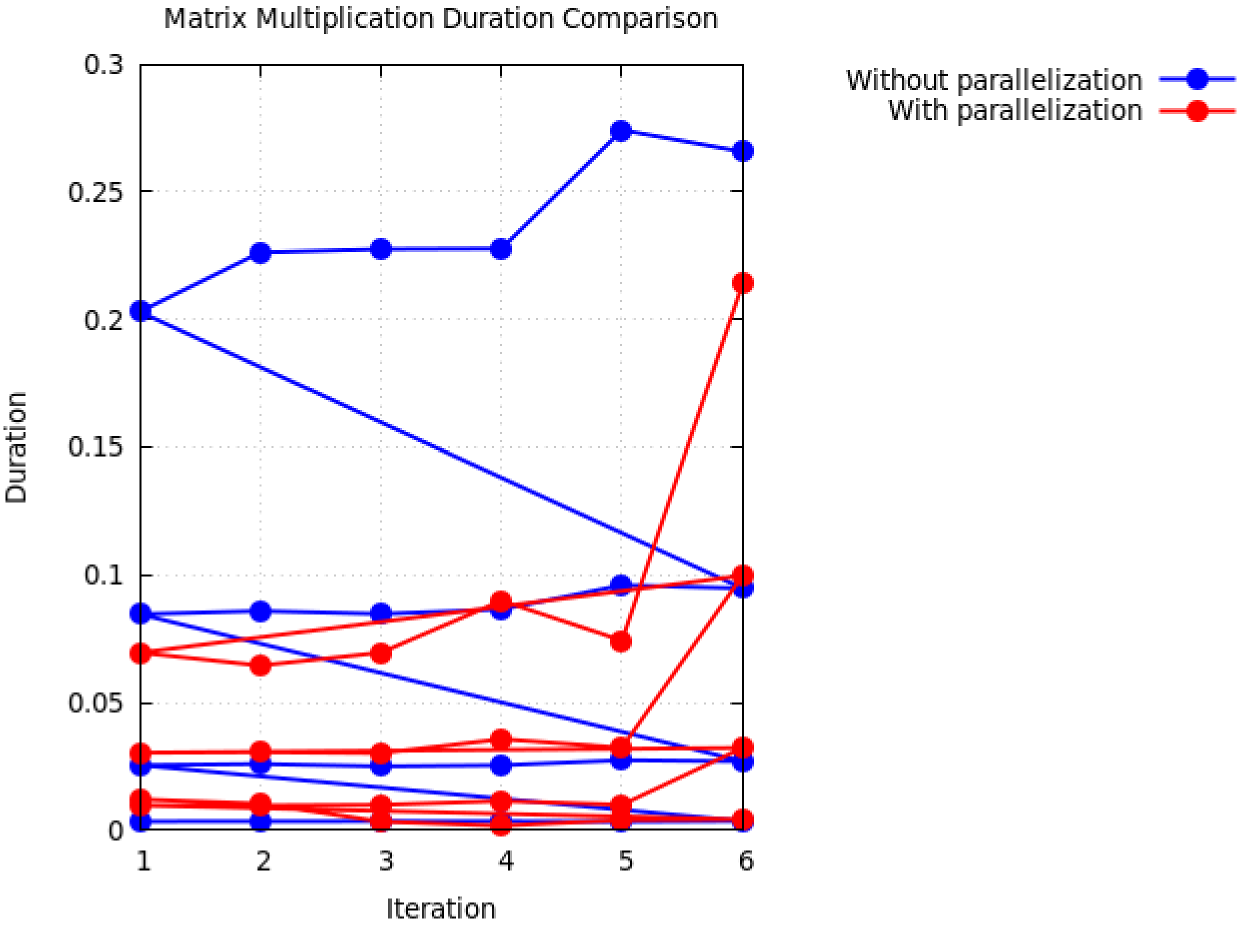
```
Iteration    Average Duration
1            0.0304747
2            0.0288313
3            0.0283058
4            0.0347333
5            0.030167
6            0.0875488
```

Gnuplot script for plotting performance graphs 1 and 2

```
set terminal pngcairo enhanced font 'Verdana,10'
set output 'ex_1_vs_2_plot.png'

set title "Matrix Multiplication Duration Comparison"
set xlabel "Iteration"
set ylabel "Duration"

set grid

set key outside

set style line 1 lt 1 lw 2 pt 7 ps 1.5 lc rgb "blue"
set style line 2 lt 1 lw 2 pt 7 ps 1.5 lc rgb "red"

plot 'data_1_0.dat' using 1:3 with linespoints linestyle 1 title 'Without parallelization', \
'data_2_0.dat' using 1:3 with linespoints linestyle 2 title 'With parallelization'
```

Performance Graphs

- Graph 1 - Time taken vs Iteration without and with parallelisation


Matrix Multiplication Duration Comparison
Duration
0
0.05
0.1
0.15
0.2
0.25
0.3
1
2
3
4
5
6
Iteration
Without parallelization
With parallelization