Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу

«Операционные системы»

Студент: Дудовцев Андрей Андреевич
Группа: М8О-208Б-22
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2024

Содержание

Репозиторий

AndreyDdvts/OS_LABS (github.com)

Цель работы

Приобретение практических навыков в:

- Управлении потоками в ОС
- Обеспечении синхронизации между потоками

Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.

Описание работы программы

Необходимо было написать программу для решения системы линейных уравнений методом Гаусса. В данном методе можно распараллелить прямой ход: а именно нахождение максимального элемента и приведение к ступенчатому виду. Параллелить обратный ход нет смысла, так как прирост эффективности слишком мал. Я представил систему линейных уравнений в матричном виде и распределил строчки по потокам. В итоге каждый поток обрабатывал определенное количество строк, и в результате получил итоговую матрицу. Далее по алгоритму нашел искомый вектор. В ходе выполнения лабораторной работы я использовал следующие системные вызовы:

- pthread_create() - создание потока
- pthread_join() - ожидание завершения поток

Исходный код

```
========================== lab2.hpp ==========================

#pragma once

#include <vector>
#include <cstdlib>
#include <algorithm>
#include <atomic>
#include <math.h>
#include <pthread.h>

using ldbl = long double;

using TVector = std::vector<ldbl>;
using TMatrix = std::vector<TVector>;

struct Args {
    int startRow = 0;
    int endRow = 0;
    TMatrix *lhs = nullptr;
    TVector *rhs = nullptr;
    int leadRow = 0;
};

struct MaxWithRow {
    ldbl value;
    int row;
};

struct ArgsForMax {
    int start = 0;
    int end = 0;
    std::vector<MaxWithRow> *maxElements = nullptr;
    const TMatrix *matrix = nullptr;
    long threadNum = 0;
```

```
};


void *MaxElem(void *arguments);
int MaxElemRowParal(const TMatrix &matrix, int start, long
threadAmount);
int MaxElemRow(const TMatrix &matrix, int start);
void SwapRows(TMatrix &lhs, TVector &rhs, int first, int second);
void *Normalization(void *arguments);
TVector GaussMethod(long threadAmount, const TMatrix &lhs, const
TVector &rhs);

========================== lab2.cpp ==========================

#include <iostream>

#include "lab2.hpp"

void *MaxElem(void *arguments) {
    const auto &args = *(reinterpret_cast<ArgsForMax *>(arguments));
    auto start = args.start;
    auto end = args.end;
    auto &maxElements = *args.maxElements;
    auto &matrix = *args.matrix;
    auto &threadNum = args.threadNum;
    ldbl maxElem = fabs(matrix[start][start]);
    int row = start;
    for (int i = start; i < end; ++i) {
        if (fabs(matrix[i][start]) > maxElem) {
            maxElem = fabs(matrix[i][start]);
            row = i;
        }
    }
    if (maxElem == 0) {
        maxElements[threadNum] = {0, -1};
```

```
        return nullptr;
    }
    maxElements[threadNum] = {maxElem, row};
    return nullptr;
}

int MaxElemRowParal(const TMatrix &matrix, int start, long
threadAmount) {
    std::vector<ArgsForMax> threadArgs(threadAmount);
    long threadAmountPerIter = std::min(threadAmount,
(long)(matrix.size() - start));
    if (threadAmountPerIter == 0) {
        return start;
    }
    long rowsPerThread = std::max(1L, (long)(((matrix.size()) - start)
/ threadAmountPerIter));
    std::vector<MaxWithRow> maxElements(threadAmountPerIter);
    ldbl absoluteMax = fabs(matrix[start][start]);
    int row = start;
    std::vector<pthread_t> threads(threadAmountPerIter);
    for (long n = 0; n < threadAmountPerIter; ++n) {
        threadArgs[n].start = start + n * rowsPerThread;
        threadArgs[n].end = (n == threadAmountPerIter - 1) ?
matrix.size() : (threadArgs[n].start + rowsPerThread);
        threadArgs[n].maxElements = &maxElements;
        threadArgs[n].matrix = &matrix;
        threadArgs[n].threadNum = n;
        pthread_create(&threads[n], nullptr, MaxElem,
reinterpret_cast<void *>(&threadArgs[n]));
    }
    for (auto &thread : threads) {
        pthread_join(thread, nullptr);
    }
    for (int i = 0; i < threadAmountPerIter; ++i) {
```

```
        if (maxElements[i].value > absoluteMax) {
            absoluteMax = maxElements[i].value;
            row = maxElements[i].row;
        }
    }
    return row;
}

int MaxElemRow(const TMatrix &matrix, int start) {
    int matrixSize = matrix.size();
    ldbl maxElem = fabs(matrix[start][start]);
    int row = start;
    for (int i = start; i < matrixSize; ++i) {
        if (fabs(matrix[i][start]) > maxElem) {
            maxElem = fabs(matrix[i][start]);
            row = i;
        }
    }
    if (maxElem == 0) {
        return -1;
    }
    return row;
}

void SwapRows(TMatrix &lhs, TVector &rhs, int first, int second) {
    lhs[first].swap(lhs[second]);
    std::swap(rhs[first], rhs[second]);
}

void *Normalization(void *arguments) {
    const auto &args = *(reinterpret_cast<Args *>(arguments));
    auto startRow = args.startRow;
    auto endRow = args.endRow;
    auto &leftMatrix = *args.lhs;
    auto &rightVector = *args.rhs;
```

```
    auto leadRow = args.leadRow;
    int matrixSize = leftMatrix.size();
    for (int i = startRow; i < endRow; ++i) {
        ldbl coef = -leftMatrix[i][leadRow] /
leftMatrix[leadRow][leadRow];
        leftMatrix[i][leadRow] = 0.0;
        for (int j = leadRow + 1; j < matrixSize; ++j) {
            leftMatrix[i][j] += leftMatrix[leadRow][j] * coef;
        }
        rightVector[i] += rightVector[leadRow] * coef;
    }
    return nullptr;
}

TVector GaussMethod(long threadAmount, const TMatrix &Mlhs, const
TVector &Vrhs) {
    TMatrix lhs = Mlhs;
    TVector rhs = Vrhs;

    long matrixSize = lhs.size();
    int leadRow = 0;
    threadAmount = std::min(threadAmount, matrixSize);
    std::vector<Args> threadArgs(threadAmount);
    for (int i = 0; i < matrixSize; ++i) {
        // Parallelization for max elem
        leadRow = (threadAmount > 1) ? MaxElemRowParal(lhs, leadRow,
threadAmount) : MaxElemRow(lhs, leadRow);
        if (leadRow == -1) {
            std::cout << "Unable to get the solution due to zero
column" << std::endl;
            return {0};
        }
        // Leading string conversion
        ldbl leadElem = lhs[leadRow][i];
```

```
        for (int k = 0; k < matrixSize; ++k) {
            lhs[leadRow][k] /= leadElem;
        }
        rhs[leadRow] /= leadElem;
        // Swap rows
        if (leadRow != i) {
            SwapRows(lhs, rhs, i, leadRow);
        } else {
            ++leadRow;
        }
        // Parallelization for strings
        if (threadAmount > 1) {
            if (i != matrixSize - 1) {
                long threadAmountPerIter = std::min(threadAmount,
matrixSize - i) - 1;
                long rowsPerThread = std::max(1L, (matrixSize - 1 - i)
/ threadAmountPerIter);
                std::vector<pthread_t> threads(threadAmountPerIter);
                for (int n = 0; n < threadAmountPerIter; ++n) {
                    threadArgs[n].startRow = i + 1 + n *
rowsPerThread;
                    threadArgs[n].endRow = (n == threadAmountPerIter -
1) ? matrixSize : (threadArgs[n].startRow + rowsPerThread);
                    threadArgs[n].lhs = &lhs;
                    threadArgs[n].rhs = &rhs;
                    threadArgs[n].leadRow = i;
                    pthread_create(&threads[n], nullptr,
Normalization, reinterpret_cast<void *>(&threadArgs[n]));
                }
                for (auto &thread : threads) {
                    pthread_join(thread, nullptr);
                }
            }
        } else {
```

```
            threadArgs[0].startRow = i + 1;
            threadArgs[0].endRow = matrixSize;
            threadArgs[0].lhs = &lhs;
            threadArgs[0].rhs = &rhs;
            threadArgs[0].leadRow = i;
            Normalization(&threadArgs[0]);
        }
    }
    // Reverse move
    TVector answer(matrixSize);
    for (int k = matrixSize - 1; k >= 0; --k) {
        answer[k] = rhs[k];
        for (int i = 0; i < k; ++i) {
            rhs[i] = rhs[i] - lhs[i][k] * answer[k];
        }
    }
    return answer;
}
```

========== main.cpp ==========

```
#include <iostream>

#include "lab2.hpp"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        std::cout << "Not enough arguments" << std::endl;
        return -1;
    }

    long threadAmount = std::atol(argv[1]);

    int n;
    std::cin >> n;
```

```
    TMatrix lhs(n, TVector(n));
    TVector rhs(n);

    auto readMatrix = [n](TMatrix &matrix) {
        for (int i = 0; i < n; ++i) {
            for(int j = 0; j < n; ++j) {
                std::cin >> matrix[i][j];
            }
        }
    };

    auto readVector = [n](TVector &vector) {
        for (int i = 0; i < n; ++i) {
            std::cin >> vector[i];
        }
    };

    auto printMatrix = [n](TMatrix &matrix, TVector &vector) {
        for (int i = 0; i < n; ++i) {
            std::cout << "| ";
            for (int j = 0; j < n; ++j) {
                std::cout << matrix[i][j] << ' ';
            }
            std::cout << '|';
            std::cout << " | x" << i + 1 << " | | " << vector[i] << "
|" << std::endl;
        }
    };

    readMatrix(lhs);
    readVector(rhs);
    std::cout << "The system of equations in matrix:" << std::endl;
    printMatrix(lhs, rhs);
```

```
    auto answer = GaussMethod(threadAmount, lhs, rhs);
    std::cout << "The solution:" << std::endl;
    for (int i = 0; i < n; ++i) {
        std::cout << "x" << i + 1 << " = " << answer[i] << std::endl;
    }

    return 0;
}
```

Тесты

```
#include <gtest/gtest.h>

#include "lab2.hpp"

#include <limits>
#include <chrono>

const ldbl LDBL_PRECISION = 0.0001;

namespace {
    TMatrix GenerateMatrix(int n) {
        TMatrix result(n, TVector(n));
        std::srand(std::time(nullptr));
        for(int i = 0; i < n; ++i) {
            for(int j = 0; j < n; ++j) {
                result[i][j] = std::rand() % 100;
            }
        }
        return result;
    }

    TVector GenerateVector(int n) {
        TVector result(n);
```

```
        std::srand(std::time(nullptr));
        for(int i = 0; i < n; ++i) {
            result[i] = std::rand() % 100;
        }
        return result;
    }
}

bool AreEqual(const TVector &first, const TVector &second) {
    if (first.size() != second.size()) {
        return false;
    }
    for(size_t i = 0; i < first.size(); ++i) {
        if (!(std::fabs(first[i] - second[i]) < LDBL_PRECISION)) {
            return false;
        }
    }
    return true;
}

TEST(SecondLabTests, SingleThreadYieldsCorrectResults) {
    ASSERT_TRUE( AreEqual(GaussMethod(1, TMatrix{{1, -1}, {2, 1}},
TVector{-5, -7}), TVector{-4, 1}) );

    ASSERT_TRUE( AreEqual(GaussMethod(1, TMatrix{{2, 4, 1}, {5, 2, 1},
{2, 3, 4}}, TVector{36, 47, 37}), TVector{7, 5, 2}) );

    ASSERT_TRUE( AreEqual(GaussMethod(1, TMatrix{{3, 2, -5}, {2, -1,
3}, {1, 2, -1}}, TVector{-1, 13, 9}), TVector{3, 5, 4}) );

    ASSERT_TRUE( AreEqual(GaussMethod(1, TMatrix{{1, 1, 2, 3}, {1, 2,
3, -1}, {3, -1, -1, -2}, {2, 3, -1, -1}}, TVector{1, -4, -4, -6}),
    TVector{-1, -1, 0, 1}) );
}
```

```
TEST(SecondLabTest, ThreadConfigurations) {
    auto performTestForGivenSize = [](int n, int maxThreadCount) {
        auto m = GenerateMatrix(n);
        auto v = GenerateVector(n);
        auto result = GaussMethod(1, m, v);

        for(int i = 2; i < maxThreadCount; ++i) {
            ASSERT_TRUE( AreEqual(GaussMethod(i, m, v), result) );
        }
    };

    performTestForGivenSize(3, 10);
    performTestForGivenSize(10, 10);
    performTestForGivenSize(100, 15);
    performTestForGivenSize(1000, 4);
}

TEST(SecondLabTest, PerfomanceTest) {
    auto getAvgTime = [](int threadCount) {
        auto m = GenerateMatrix(3000);
        auto v = GenerateVector(3000);

        constexpr int runsCount = 1;

        double avg = 0;

        for(int i = 0; i < runsCount; ++i) {
            auto begin = std::chrono::high_resolution_clock::now();
            GaussMethod(threadCount, m, v);
            auto end = std::chrono::high_resolution_clock::now();
            avg +=
std::chrono::duration_cast<std::chrono::milliseconds>(end -
begin).count();
```

```
        }

        return avg / runsCount;
    };

    auto singleThread = getAvgTime(1);
    auto multiThread = getAvgTime(4);

    std::cout << "Avg time for 1 thread: " << singleThread << '\n';
    std::cout << "Avg time for 4 threads: " << multiThread << '\n';

    EXPECT_GE(singleThread, multiThread);
}

int main(int argc, char *argv[]) {
    testing::InitGoogleTest(&argc, argv);

    std::cout.precision(15);
    std::cout.setf(std::ios_base::fixed, std::ios_base::floatfield);

    return RUN_ALL_TESTS();
}
```

Консоль

```
qwz@qwz-VirtualBox:~/OS_LABS/build/lab2$ ./lab2 10
2
1 -1
2 1
-5 -7
The system of equations in matrix:
| 1 -1 | | x1 | | -5 |
| 2 1 | | x2 | | -7 |
The solution:
x1 = -4
```

```
x2 = 1
qwz@qwz-VirtualBox:~/OS_LABS/build/lab2$ ./lab2 8
3
2 4 1
5 2 1
2 3 4
36 47 37
The system of equations in matrix:
| 2 4 1 | | x1 | | 36 |
| 5 2 1 | | x2 | | 47 |
| 2 3 4 | | x3 | | 37 |
The solution:
x1 = 7
x2 = 5
x3 = 2
qwz@qwz-VirtualBox:~/OS_LABS/build/lab2$ ./lab2 1
4
1 1 2 3
1 2 3 -1
3 -1 -1 -2
2 3 -1 -1
1 -4 -4 -6
The system of equations in matrix:
| 1 1 2 3 | | x1 | | 1 |
| 1 2 3 -1 | | x2 | | -4 |
| 3 -1 -1 -2 | | x3 | | -4 |
| 2 3 -1 -1 | | x4 | | -6 |
The solution:
x1 = -1
x2 = -1
x3 = 0
x4 = 1
```

Запуск тестов

```
qwz@qwz-VirtualBox:~/OS_LABS/build/tests$ ./lab2_test
[==========] Running 3 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 1 test from SecondLabTests
[ RUN      ] SecondLabTests.SingleThreadYieldsCorrectResults
[       OK ] SecondLabTests.SingleThreadYieldsCorrectResults (0 ms)
[----------] 1 test from SecondLabTests (0 ms total)

[----------] 2 tests from SecondLabTest
[ RUN      ] SecondLabTest.ThreadConfigurations
[       OK ] SecondLabTest.ThreadConfigurations (9812 ms)
[ RUN      ] SecondLabTest.PerfomanceTest
Avg time for 1 thread: 3036.000000000000000
Avg time for 4 threads: 2060.000000000000000
[       OK ] SecondLabTest.PerfomanceTest (5141 ms)
[----------] 2 tests from SecondLabTest (14953 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 2 test suites ran. (14953 ms total)
[  PASSED  ] 3 tests.
```

Выводы

В результате выполнения данной лабораторной работы была написана программа на языке C++ для решения системы линейных уравнений методом Гаусса, обрабатывающая данные в многопоточном режиме. Я приобрел практические навыки в управлении потоками в ОС и обеспечении синхронизации между потоками.