Министерство образования и науки Российской Федерации Федеральное агентство по образованию Федеральное государственное бюджетное образовательное учреждение высшего образования «Вятский государственный университет»

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

Лабораторная работа №2

по курсу «Параллельное программирование»

Выполнил студент группы ИВТ-31\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Птахова А.М/

Проверил доцент кафедры ЭВМ \_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Долженкова М.Л./

Киров 2023

**1. Цель**

Получение навыков многопоточной реализации вычислительно сложных алгоритмов с использованием базовых примитивов синхронизации, получение навыков оценки эффективности многопоточных реализаций алгоритмов.

**2. Задание**

- Выполнить разбиение исследованного в ходе первой лабораторной работы алгоритма на независимо выполняемые фрагменты;

- Реализовать многопоточную версию алгоритм с помощью языка С++ и потоков операционной системы, используя при этом необходимые примитивы синхронизации;

- Показать корректность полученной реализации путем осуществления тестирования на построенном в ходе первой лабораторной работы наборе тестов;

- Провести доказательную оценку эффективности многопоточной реализации алгоритма.

**3. Алгоритм**

Для разбиения был выбран простой алгоритм Краута по 2м причинам:

1) минимальное время выполнения операции по сравнению с усовершенствованным алгоритмом Краута;

2) есть возможность распараллеливания.

**3.1. Описание алгоритма разбиения**

При предположении, что LU - разложение матрицы существует, исходную матрицу А можно записать в виде произведения матриц L, U. Их произведение представлено на рисунке 1.

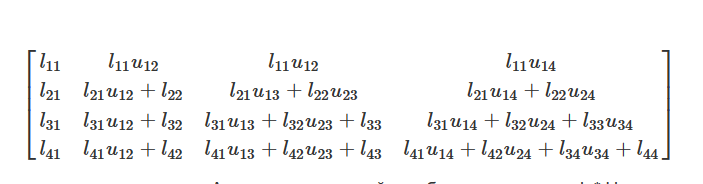


Рисунок 1 – Произведение матриц L, U

Сравнивая компоненты произведения, можно вывести формулы для нахождения элементов матриц. Эти формулы представлены на рисунке 2.

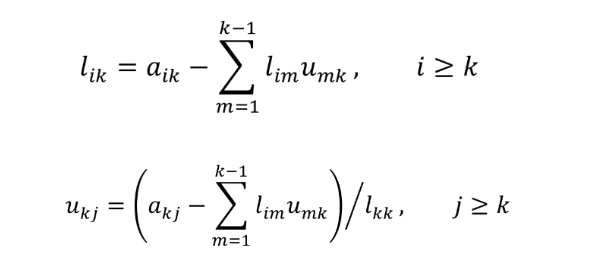


Рисунок 2 – Формулы для нахождения элементов матриц

Продолжая анализировать компоненты произведения и формулы, можно заметить, что вычисление элементов одной строки не зависит от вычисления элементов другой строки, поэтому их можно распараллелить.

Многопоточность алгоритма поддерживается 5 созданными потоками. Выбор именно такого количества потоков обусловлен следующими причинами:

1) наличием 6 ядерного процессора;

2) делимостью размерности матрицы для каждого теста.

Второй пункт необходимо было учитывать, так создание потока со значением, выходящим за пределы массива, приводит к возникновению ошибки. Если же использовать ограничения (проверка выполнения условия), это заведомо усложнит код и увеличит время выполнения подпрограммы, так как нелинейные фрагменты программы требуют больше времени на осуществление перехода.

Что касается синхронизации, то она здесь нужна в момент окончания вычислений. То есть запуск новых потоков должен выполняться лишь после того, как все потоки закончат свою работу.

**4. Листинг программы**

void lu\_decomposition(vector <vector <double>> &mas, long n, long i)

{

for (long k = i; k < n; k++)

{

double sum = 0;

for (long j = 0; j < i; j++)

{

sum += (mas[i][j] \* mas[j][k]);

}

mas[i][k] = mas[i][k] - sum;

}

for (long k = i; k < n; k++)

{

if (i != k)

{

//count++;

int sum = 0;

for (long j = 0; j < i; j++)

{

sum += (mas[k][j] \* mas[j][i]);

}

mas[k][i] = (mas[k][i] - sum) / mas[i][i];

}

}

}

void print\_matrix(vector <vector<double>>& mas, int n)

{

for (int i = 0; i < mas.size(); i++)

{

for (int j = 0; j < mas[i].size(); j++)

{

cout << mas[i][j] << ' ';

}

cout << '\n';

}

}

int main()

{

const unsigned long n = 5000;

vector <vector<double>> a(n, vector<double>(n));

for (unsigned long i = 0; i < a.size(); i++)

{

for (unsigned long j = 0; j < a[i].size(); j++)

{

a[i][j] = rand() % 10;

}

}

//начало выполнение

auto start = high\_resolution\_clock::now();

for (int i = 0; i < n; i+=5)

{

thread A(lu\_decomposition, ref(a), n, i);

thread B(lu\_decomposition, ref(a), n, i + 1);

thread C(lu\_decomposition, ref(a), n, i + 2);

thread D(lu\_decomposition, ref(a), n, i + 3);

thread E(lu\_decomposition, ref(a), n, i + 4);

A.join();

B.join();

C.join();

D.join();

E.join();

}

//конец выполнения

auto stop = high\_resolution\_clock::now();

//время выполнения

auto duraction = duration\_cast<microseconds>(stop - start);

ofstream outf{ "result.txt", ios::app };

outf << "rang: " << n << " time: " << duration<double, milli>(duraction).count() << " ms" << '\n';

cout << "end\n";

return 0;

}

**5. Сравнительные результаты**

Результаты сравнения простого и многопоточного алгоритма приведены в таблице 1.

Таблица 1 – Сравнительные результаты

|  |  |  |  |
| --- | --- | --- | --- |
| № | Размерность | Алгоритм (прост.) | Алгоритм (парал.) |
| 1 | 10 | time: 0.019 ms | time: 9.851 ms |
| 2 | 50 | time: 2.04 ms | time: 18.373 ms |
| 3 | 100 | time: 15.798 ms | time: 37.944 ms |
| 4 | 500 | time: 1890.83 ms | time: 577.73 ms |
| 5 | 1000 | time: 15242 ms | time: 3399.6 ms |
| 6 | 2000 | time: 148965 ms | time: 31956.3 ms |
| 7 | 5000 | time: 1.93568e+06 ms | time: 412307 ms |
| 8 | 7500 | time: 6.65623e+06 ms | time: 1.55738e+06 ms |
| 9 | 13000 | time: 2.31827e+07 ms | time: 4.9538e+06 ms |
| 10 | 15000 | time: 1.414706e+08 ms | time: 3.15265e+07 ms |

**6. Оценка эффективности алгоритма**

В таблице 1 приведены результаты тестов, из которых видно, что при размерности 100 параллельный алгоритм «проигрывает» примерно в 2 раза. В тоже время при размерности 500 этот же алгоритм «выигрывает» уже в 4 раза. Отсюда можно сделать вывод о том, что параллельный алгоритм эффективно применять на тестах с размерностью больше 300.

**7. Вывод**

В ходе выполнения лабораторной работы было реализовано разбиение исследованного в ходе первой лабораторной работы алгоритма на независимо выполняемые фрагменты и реализована многопоточная версия алгоритма. В результате были получены данные о времени выполнения подпрограммы на различных тестовых наборах. Как можно заметить, на матрицах с размерностью меньше 500 элементов многопоточный алгоритм значительно проигрывает последовательному. Это можно объяснить тем, что хоть и вычисления происходят быстрее, время на создание потоков и на ожидании их окончания превышает этот «выигрыш». При увеличении размерности матрицы многопоточный алгоритм выполняется быстрее, чем последовательный. Это связано с тем, что время задержки при создании потоков и при их синхронизации покрывается быстродействием многопоточности.