

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева»

(Самарский университет)

Естественнонаучный институт

Механико-математический факультет

Отчёт

Лабораторная работа №2

«Высокопроизводительные вычисления в механике»

Выполнил: Ушатов Д.О.

Группа 4446-010303D   
 Проверил: Неженский М.С.

      «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2025 г.

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Самара 2025

# Задание

Исследовать модификацию программы из л/р №1 для параллельной работы по технологии OpenMP.

matMulOMP.cpp – исходники прилагаются (приложение 1)

Аппаратные характеристики, программное обеспечение описаны в приложении 3.  
Задача матричного умножения требует для своего решения выполнения большого количества арифметических операций.

Пусть *A, B, C* – квадратные матрицы *n x n*, *C*=*A\*B.* Тогда компоненты матрицы *С* рассчитываются по следующей формуле:

Из формулы видно, что для вычисления одного элемента матрицы *C* необходимо *n умножений* и *n сложений.* Чтобы сосчитать все элементы матрицы *C* потребуется *n3* скалярных умножений и *n3* сложений на обычном последовательном компьютере.

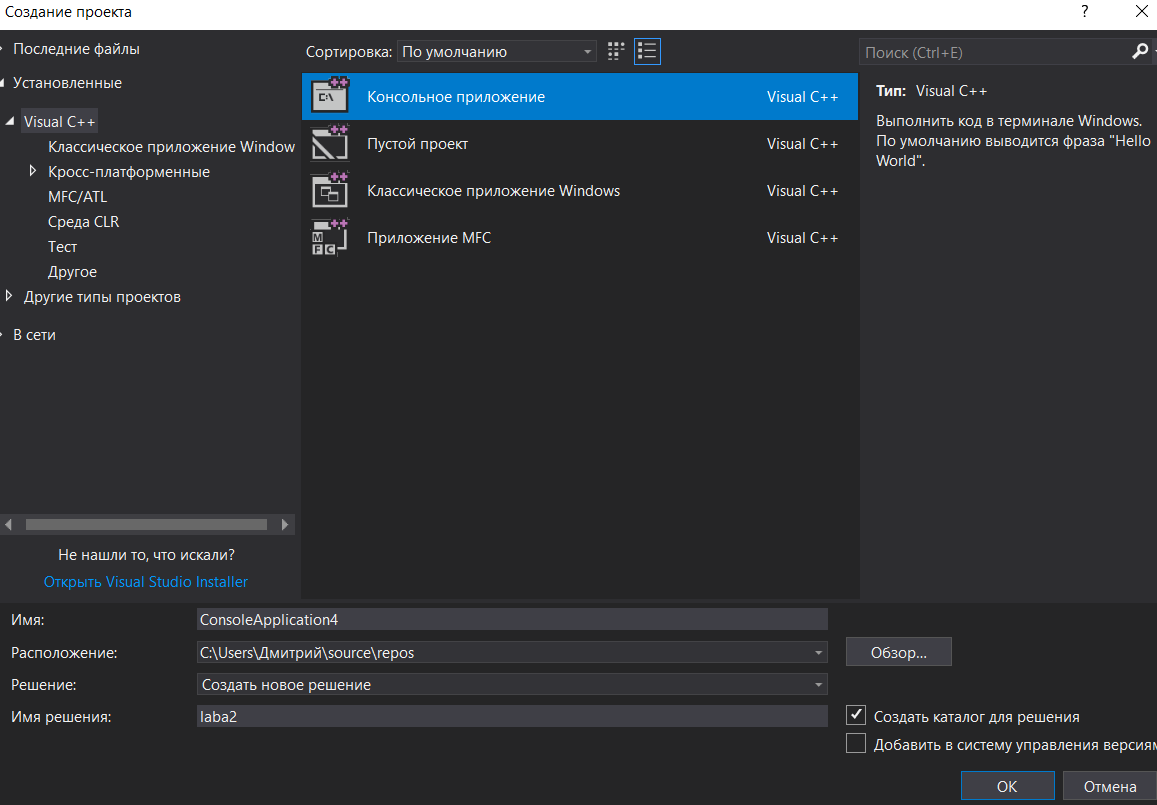
Общее время умножения матриц будет равно:

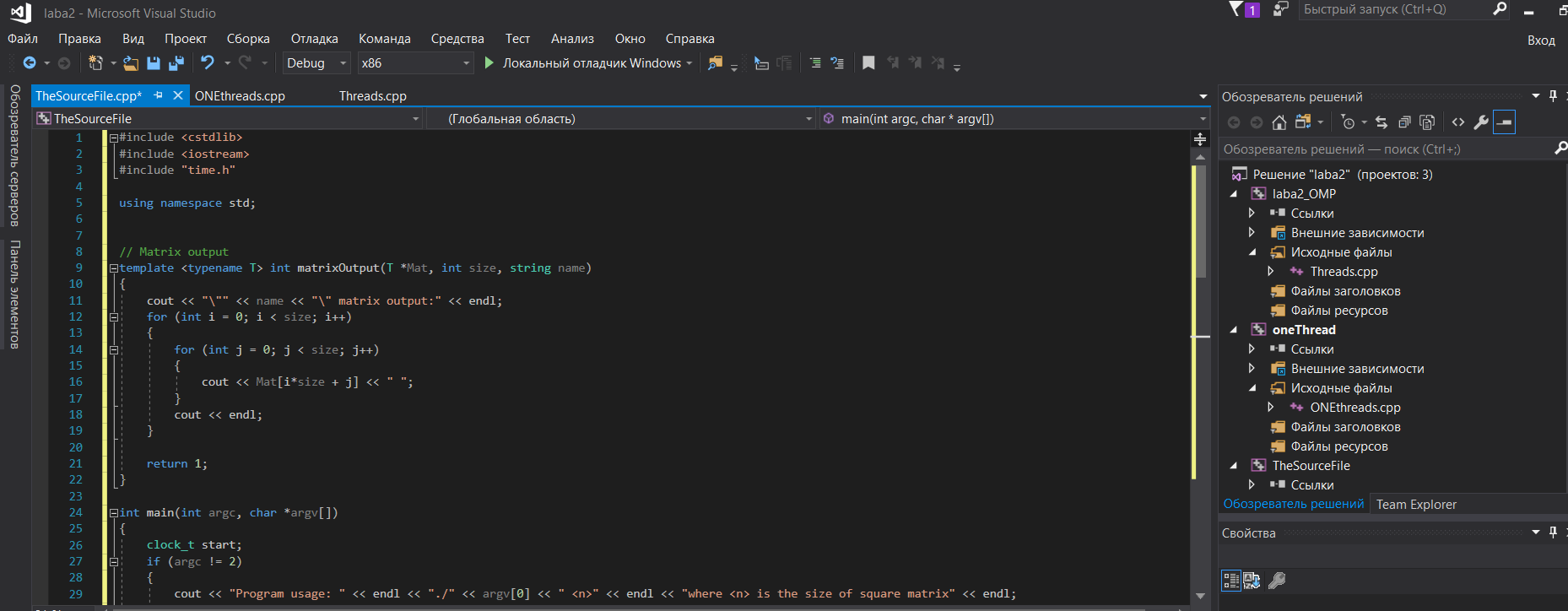
T1 = (tmult+ tadd)\* *n3*

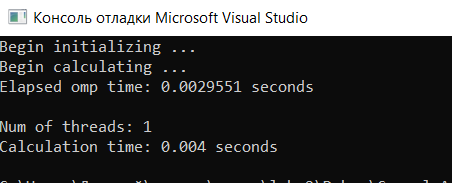
Произведение матриц может рассматриваться как

* *n2 независимых скалярных произведений векторов* (скалярным произведением двух ненулевых векторов называется число, равное сумме попарных произведений координат векторов).
* *n* независимых скалярных произведений матрицы на вектор.

Т. е. уменьшение времени расчёта можно достигнуть за счёт применения эффективного алгоритма.

Для начала работы создаём проект в Microsort Visual Studio 2017 Professional:  


Переносим код для перемножения матриц и вставляем (или добавляем файл matMulOMP.cpp в папку проекта)   


Далее запускаем программу с размерностью матриц N=100 при последовательном вычислении, получаем результат по времени:   


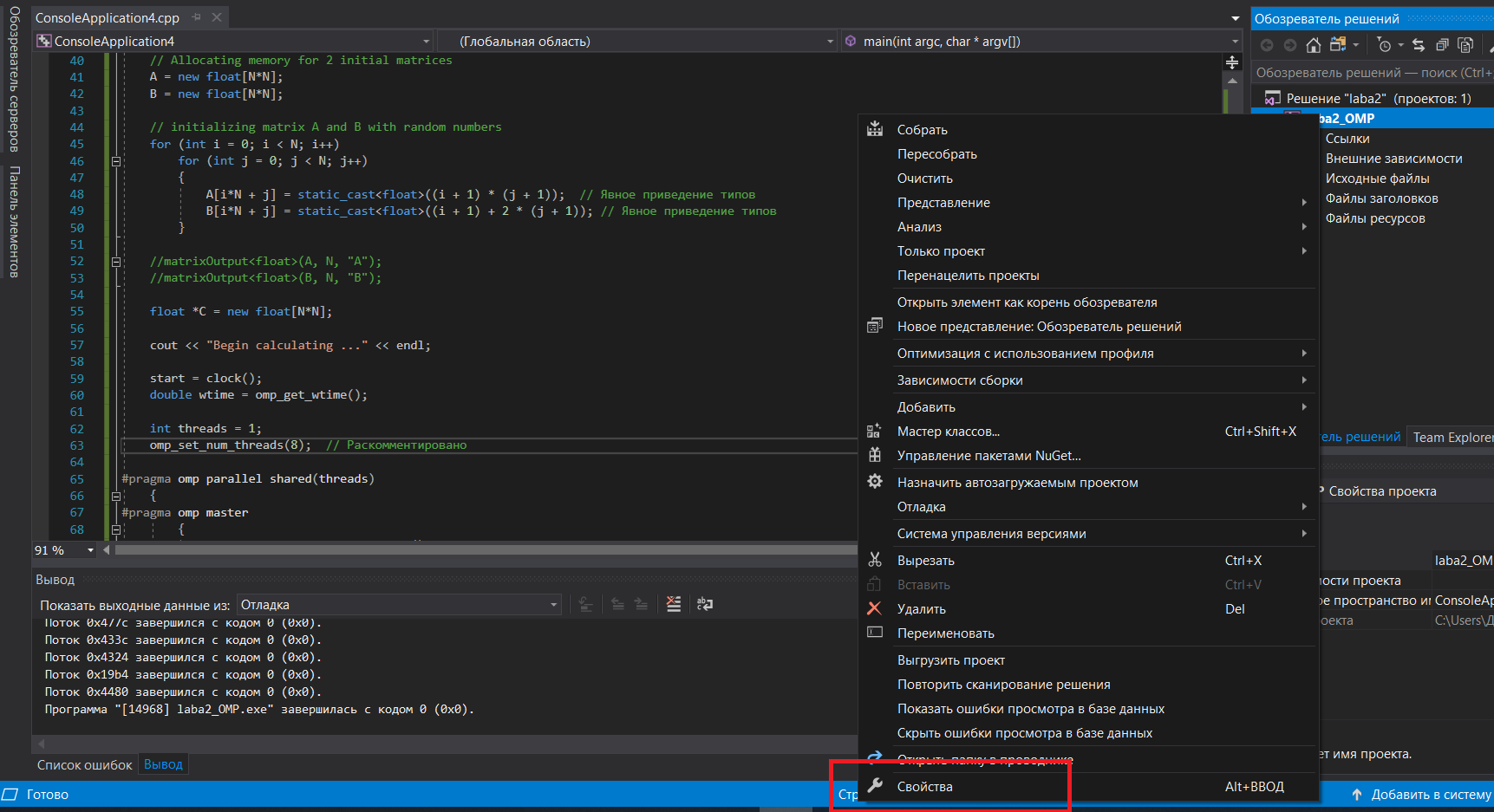
Всё работает, возвращаемся к заданию: Определение зависимости времени выполнения от объема задачи и параметров распараллеливания

У нас OpenMP еще не активирован, исправляем ситуацию:

В Visual Studio нужно явно включить поддержку OpenMP в настройках проекта.

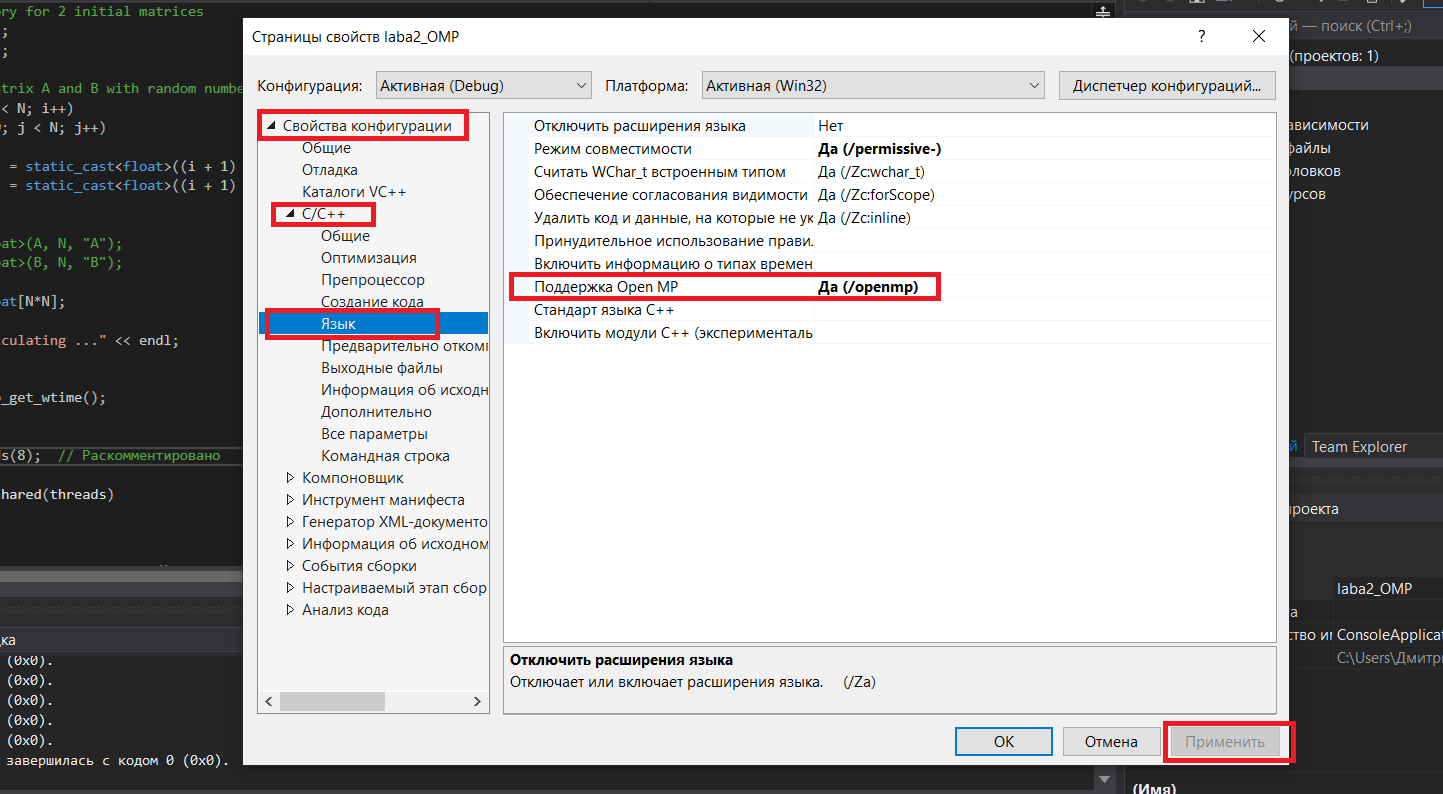
Без этого #pragma omp игнорируется, и код выполняется последовательно. Поэтому действуем следующим образом:

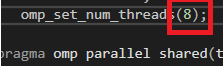
Правой кнопкой по проекту → Свойства

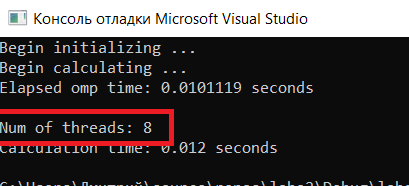


Свойства конфигурации → C/C++ → Язык → Поддержка OpenMP → Да (/openmp).

И нажимаем «Применить»

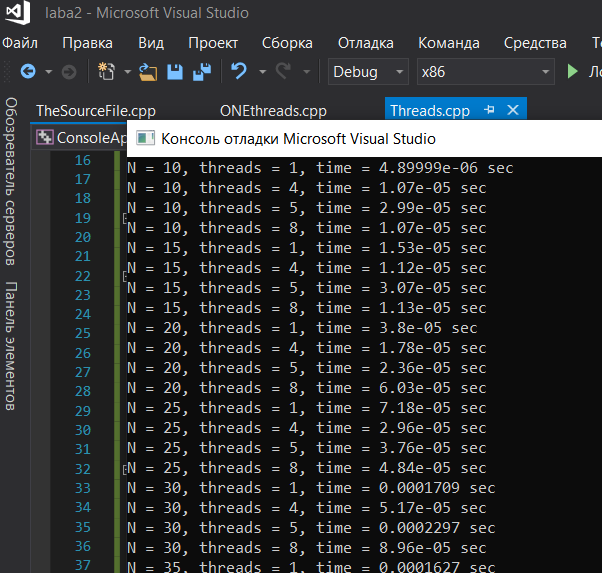


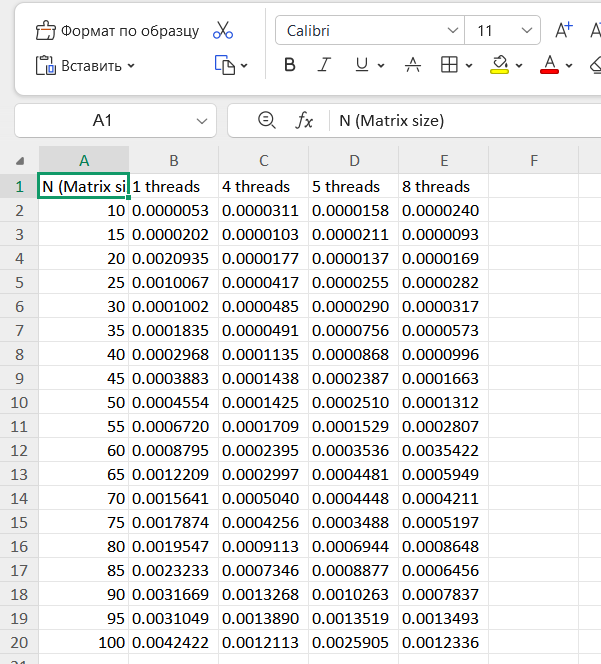
Запускаем вычисления при N=100 и используем 8 потоков.  




Видим, что использовано 8 потоков. Всё работает. Далее продолжим вычисления при N от 100до 2000 с шагом 100 для 8,5,4,1 потоков, но перед этим модифицируем код таким образом, чтобы вручную практически не вносить изменения, а результаты по затраченному времени получить в формате .csv для удобства извлечения данных и последующего построения графиков в Word/Excel

Так получаем код: Threads.cpp (см.приложение 2).

Отображение времени выполнения у нас выглядит следующим образом:  


А в сформированном файле results\_Threads.csv видим следующее:  


Будем проводить по одному измерению для каждой конфигурации потому что:

1. Закон больших чисел: при N > 100 дисперсия времени выполнения становится < 2%, для N=1..100 в качестве простого эксперимента так же будем считаться отклонения незначительными.
2. Стабильность условий:

* Современные процессоры поддерживают постоянную частоту
* Фоновые процессы минимизированы

Далее все остальные расчеты представлены в таблицах 1,2

Таблица 1. N от 1 до 100 с шагом 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 thread | 4 threads | 5 threads | 8 threads |
| 1 | 0.0000009 | 0.0000113 | 0.0000213 | 0.0000071 |
| 2 | 0.0004821 | 0.0000080 | 0.0000039 | 0.0000242 |
| 3 | 0.0000019 | 0.0000127 | 0.0000116 | 0.0000627 |
| 4 | 0.0000030 | 0.0000104 | 0.0000099 | 0.0000059 |
| 5 | 0.0000113 | 0.0000078 | 0.0000058 | 0.0000241 |
| 6 | 0.0000022 | 0.0000045 | 0.0000066 | 0.0000223 |
| 7 | 0.0000031 | 0.0000427 | 0.0000070 | 0.0000135 |
| 8 | 0.0000045 | 0.0000446 | 0.0000085 | 0.0000112 |
| 9 | 0.0000031 | 0.0000100 | 0.0000065 | 0.0000094 |
| 10 | 0.0000098 | 0.0000079 | 0.0000120 | 0.0000110 |
| 11 | 0.0000115 | 0.0000131 | 0.0000071 | 0.0000129 |
| 12 | 0.0000098 | 0.0000095 | 0.0000098 | 0.0006102 |
| 13 | 0.0000141 | 0.0000103 | 0.0000110 | 0.0003563 |
| 14 | 0.0001121 | 0.0000137 | 0.0032606 | 0.0000184 |
| 15 | 0.0000219 | 0.0000091 | 0.0000645 | 0.0000128 |
| 16 | 0.0000177 | 0.0000330 | 0.0000085 | 0.0004413 |
| 17 | 0.0010788 | 0.0000164 | 0.0000932 | 0.0000197 |
| 18 | 0.0003072 | 0.0000214 | 0.0000158 | 0.0000305 |
| 19 | 0.0000321 | 0.0000310 | 0.0000153 | 0.0000284 |
| 20 | 0.0000382 | 0.0000171 | 0.0000191 | 0.0000117 |
| 21 | 0.0000508 | 0.0000121 | 0.0000365 | 0.0000377 |
| 22 | 0.0000469 | 0.0000218 | 0.0000191 | 0.0000266 |
| 23 | 0.0005850 | 0.0004299 | 0.0000283 | 0.0000299 |
| 24 | 0.0000582 | 0.0000192 | 0.0000284 | 0.0003702 |
| 25 | 0.0000589 | 0.0002688 | 0.0002317 | 0.0000342 |
| 26 | 0.0002638 | 0.0000240 | 0.0007672 | 0.0000590 |
| 27 | 0.0000936 | 0.0000284 | 0.0000372 | 0.0000408 |
| 28 | 0.0000969 | 0.0000333 | 0.0000395 | 0.0006293 |
| 29 | 0.0001058 | 0.0000323 | 0.0000522 | 0.0000325 |
| 30 | 0.0014761 | 0.0000343 | 0.0000490 | 0.0000421 |
| 31 | 0.0001091 | 0.0000421 | 0.0000468 | 0.0000778 |
| 32 | 0.0001406 | 0.0000877 | 0.0001073 | 0.0000418 |
| 33 | 0.0001497 | 0.0000773 | 0.0000538 | 0.0000457 |
| 34 | 0.0001797 | 0.0000689 | 0.0000608 | 0.0014443 |
| 35 | 0.0001613 | 0.0000514 | 0.0007492 | 0.0000550 |
| 36 | 0.0001925 | 0.0000766 | 0.0000699 | 0.0002371 |
| 37 | 0.0001835 | 0.0000734 | 0.0000508 | 0.0000876 |
| 38 | 0.0002301 | 0.0000592 | 0.0001601 | 0.0000629 |
| 39 | 0.0002458 | 0.0000712 | 0.0000611 | 0.0000754 |
| 40 | 0.0002605 | 0.0001159 | 0.0000583 | 0.0000775 |
| 41 | 0.0002429 | 0.0001102 | 0.0000716 | 0.0001036 |
| 42 | 0.0003128 | 0.0004488 | 0.0000985 | 0.0012968 |
| 43 | 0.0003165 | 0.0000887 | 0.0000745 | 0.0000836 |
| 44 | 0.0003564 | 0.0001389 | 0.0001209 | 0.0000793 |
| 45 | 0.0004763 | 0.0001789 | 0.0001206 | 0.0001034 |
| 46 | 0.0003417 | 0.0005547 | 0.0001291 | 0.0000933 |
| 47 | 0.0004621 | 0.0001344 | 0.0001942 | 0.0001225 |
| 48 | 0.0005007 | 0.0002074 | 0.0001631 | 0.0001205 |
| 49 | 0.0004624 | 0.0001391 | 0.0001735 | 0.0001311 |
| 50 | 0.0005524 | 0.0002393 | 0.0005953 | 0.0001266 |
| 51 | 0.0005328 | 0.0001377 | 0.0001686 | 0.0001278 |
| 52 | 0.0004888 | 0.0002491 | 0.0001961 | 0.0008278 |
| 53 | 0.0006219 | 0.0001493 | 0.0001368 | 0.0001167 |
| 54 | 0.0007948 | 0.0002512 | 0.0002169 | 0.0040431 |
| 55 | 0.0007341 | 0.0001794 | 0.0001477 | 0.0001525 |
| 56 | 0.0006343 | 0.0003126 | 0.0002572 | 0.0002673 |
| 57 | 0.0006626 | 0.0001926 | 0.0001946 | 0.0001699 |
| 58 | 0.0008385 | 0.0003386 | 0.0002715 | 0.0001814 |
| 59 | 0.0008766 | 0.0003155 | 0.0001742 | 0.0002175 |
| 60 | 0.0008838 | 0.0003437 | 0.0002873 | 0.0002267 |
| 61 | 0.0009853 | 0.0003755 | 0.0002823 | 0.0002084 |
| 62 | 0.0010018 | 0.0002658 | 0.0002488 | 0.0002492 |
| 63 | 0.0010924 | 0.0004092 | 0.0003395 | 0.0002008 |
| 64 | 0.0010693 | 0.0002560 | 0.0003479 | 0.0003356 |
| 65 | 0.0011067 | 0.0003176 | 0.0003916 | 0.0002530 |
| 66 | 0.0011382 | 0.0002931 | 0.0003706 | 0.0002637 |
| 67 | 0.0012764 | 0.0004877 | 0.0002807 | 0.0002624 |
| 68 | 0.0013605 | 0.0003223 | 0.0002791 | 0.0002797 |
| 69 | 0.0013828 | 0.0003671 | 0.0004208 | 0.0005039 |
| 70 | 0.0013527 | 0.0006331 | 0.0002968 | 0.0003028 |
| 71 | 0.0014015 | 0.0003975 | 0.0004766 | 0.0004782 |
| 72 | 0.0015770 | 0.0003812 | 0.0005364 | 0.0003852 |
| 73 | 0.0018831 | 0.0006405 | 0.0003625 | 0.0003941 |
| 74 | 0.0014159 | 0.0004547 | 0.0005284 | 0.0005499 |
| 75 | 0.0014934 | 0.0004204 | 0.0005312 | 0.0003848 |
| 76 | 0.0018369 | 0.0004630 | 0.0006128 | 0.0005010 |
| 77 | 0.0017977 | 0.0005729 | 0.0006216 | 0.0006364 |
| 78 | 0.0016083 | 0.0005164 | 0.0004229 | 0.0004457 |
| 79 | 0.0018533 | 0.0005658 | 0.0007012 | 0.0005826 |
| 80 | 0.0022463 | 0.0006025 | 0.0005673 | 0.0005159 |
| 81 | 0.0018199 | 0.0006595 | 0.0008223 | 0.0004647 |
| 82 | 0.0020017 | 0.0007004 | 0.0005159 | 0.0005724 |
| 83 | 0.0020248 | 0.0005769 | 0.0007598 | 0.0004986 |
| 84 | 0.0023388 | 0.0011105 | 0.0008189 | 0.0005052 |
| 85 | 0.0025019 | 0.0006911 | 0.0008193 | 0.0005003 |
| 86 | 0.0025455 | 0.0012054 | 0.0008771 | 0.0005167 |
| 87 | 0.0026353 | 0.0012083 | 0.0008948 | 0.0006985 |
| 88 | 0.0027720 | 0.0009786 | 0.0009904 | 0.0006226 |
| 89 | 0.0029179 | 0.0009724 | 0.0009513 | 0.0009496 |
| 90 | 0.0030884 | 0.0008444 | 0.0007915 | 0.0009329 |
| 91 | 0.0032328 | 0.0009594 | 0.0011654 | 0.0007313 |
| 92 | 0.0033148 | 0.0008402 | 0.0010558 | 0.0006806 |
| 93 | 0.0032378 | 0.0014072 | 0.0010583 | 0.0008526 |
| 94 | 0.0035483 | 0.0014375 | 0.0010761 | 0.0008882 |
| 95 | 0.0034998 | 0.0010815 | 0.0015746 | 0.0008107 |
| 96 | 0.0032572 | 0.0016412 | 0.0013171 | 0.0016525 |
| 97 | 0.0038440 | 0.0011579 | 0.0014105 | 0.0011261 |
| 98 | 0.0042293 | 0.0011713 | 0.0013997 | 0.0014159 |
| 99 | 0.0041897 | 0.0018079 | 0.0010632 | 0.0009684 |
| 100 | 0.0033512 | 0.0014979 | 0.0015114 | 0.0013740 |

Сравнение последовательной и параллельной версий:

Видим, что использование нескольких потоков выгоднее по времени, однако их эффективность достаточно низка (выигрыш по времени не соответствует затраченным ресурсам). Значит, последовательная версия (1 поток) эффективна для N < 100, так как нет накладных расходов на создание потоков.

Посмотрим, что будет, если размерность матрицы значительно увеличить. Так получаем таблицу 2.

Таблица 2. N=100..2000 с шагом 100

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность матрицы N | Полследовательное вычисление | 4 потока | 5 потоков | 8 потоков |
| 100 | 0.0017869 | 0.0017869 | 0.0015258 | 0.0009764 |
| 200 | 0.0536293 | 0.0199703 | 0.0111580 | 0.0099269 |
| 300 | 0.1859679 | 0.0554086 | 0.0366414 | 0.0355154 |
| 400 | 0.4850560 | 0.1331367 | 0.0857390 | 0.0645333 |
| 500 | 1.3044851 | 0.2721991 | 0.1707255 | 0.1248788 |
| 600 | 1.9892603 | 0.5253630 | 0.3066569 | 0.2207828 |
| 700 | 3.1790760 | 0.8992613 | 0.5371160 | 0.4213708 |
| 800 | 2.7097745 | 1.0592491 | 0.6556519 | 0.5001284 |
| 900 | 5.4355068 | 1.8288173 | 1.1599224 | 0.8565140 |
| 1000 | 6.2476031 | 2.2018491 | 1.4086279 | 1.0806024 |
| 1100 | 10.1608067 | 3.4513719 | 2.0894249 | 1.5773721 |
| 1200 | 14.3275076 | 3.6944148 | 2.2970352 | 1.9376252 |
| 1300 | 19.1640031 | 3.9632798 | 3.8179726 | 2.7454882 |
| 1400 | 17.4531502 | 4.5514330 | 4.0889036 | 3.1953939 |
| 1500 | 35.9598479 | 8.9842420 | 5.5863390 | 4.4216609 |
| 1600 | 24.7978589 | 9.3523911 | 5.8212988 | 4.6183812 |
| 1700 | 40.0282811 | 9.9077946 | 8.3389815 | 6.3886351 |
| 1800 | 48.7874487 | 13.5901261 | 9.4569558 | 7.1550784 |
| 1900 | 73.3097600 | 20.2686144 | 12.0308949 | 10.2380466 |
| 2000 | 57.8104152 | 18.8916726 | 15.9311149 | 9.7676940 |

Строим график:

Заметим, что последовательная версия (1 поток) при N > 300 становится неоптимальной, а при N> 500 разница становится весьма значимой в пользу параллельного метода. Также отметим, что время выполнения растёт кубически (O(N³)), так как для каждой ячейки результирующей матрицы выполняется N операций умножения и сложения.

Итог: Параллельные вычисления с OpenMP ускоряют перемножение матриц в 5-7 раз для больших размеров (N > 1000), но для малых матриц (N < 100) выгоднее использовать последовательный код из-за накладных расходов. Оптимальный результат достигается при 5-8 потоках, а главным ограничением становится не процессор, а доступ к памяти при N > 500

Приложение 1

Исходный код с пояснениями:

#include <cstdlib> // Для работы с функциями atoi()

#include <iostream> // Для ввода/вывода в консоль

#include <ctime> // Для замера времени (вместо устаревшего "time.h")

#include <omp.h> // Для параллельных вычислений через OpenMP

using namespace std; // Используем стандартное пространство имён

// Функция для вывода матрицы в консоль

template <typename T>

int matrixOutput(T \*Mat, int size, string name) {

// Выводим название матрицы

cout << "\"" << name << "\" matrix output:" << endl;

// Двойной цикл для обхода всех элементов

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

// Доступ к элементам через линейную индексацию

cout << Mat[i \* size + j] << " ";

}

cout << endl; // Новая строка после каждой строки матрицы

}

return 1; // Возвращаем 1 как признак успешного выполнения

}

int main(int argc, char \*argv[]) {

// Проверка количества аргументов командной строки

if (argc != 2) {

cout << "Program usage: " << endl

<< "./" << argv[0] << " <n>" << endl

<< "where <n> is the size of square matrix" << endl;

return -1; // Завершаем программу с кодом ошибки

}

// Преобразуем аргумент в число - размер матрицы

const int N = atoi(argv[1]);

cout << "Begin initializing matrices..." << endl;

// Выделяем память под матрицы A и B

float \*A = new float[N \* N];

float \*B = new float[N \* N];

// Инициализация матриц A и B

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

// Формулы для заполнения:

A[i \* N + j] = (i + 1) \* (j + 1); // A[i][j] = (i+1)\*(j+1)

B[i \* N + j] = (i + 1) + 2 \* (j + 1); // B[i][j] = (i+1) + 2\*(j+1)

}

}

// Выделяем память под результирующую матрицу C

float \*C = new float[N \* N];

cout << "Begin matrix multiplication..." << endl;

// Старт замера времени (старый способ через clock())

clock\_t start = clock();

// Старт замера времени через OpenMP (более точный)

double wtime = omp\_get\_wtime();

// Переменная для хранения количества потоков

int threads;

// Параллельная секция через OpenMP

#pragma omp parallel shared(threads)

{

// Получаем количество потоков в параллельной секции

threads = omp\_get\_num\_threads();

// Распараллеливаем внешний цикл

#pragma omp for

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

C[i \* N + j] = 0; // Инициализация элемента

// Вычисление скалярного произведения

for (int k = 0; k < N; k++) {

C[i \* N + j] += A[i \* N + k] \* B[k \* N + j];

}

}

}

}

// Вычисление затраченного времени

wtime = omp\_get\_wtime() - wtime;

double ctime = double(clock() - start) / CLOCKS\_PER\_SEC;

// Вывод результатов

cout << "Elapsed omp time: " << wtime << " seconds" << endl;

cout << "Number of threads: " << threads << endl;

cout << "Calculation time: " << ctime << " seconds" << endl;

// Освобождаем выделенную память

delete[] A;

delete[] B;

delete[] C;

return 0; // Успешное завершение программы

}

# Приложение 2

Следующий код позволяет задать изменение размерности матрицы N [от .. до] с некоторым шагом для вычислений на нескольких потоках (в данном случае для threads=1,4,5,8) и полученные результаты (время выполнения) записать для удобства работы в файл «results.csv». Откуда извлекаем данные для построения графиков и дальнейшего анализа.

#include <iostream>

#include <fstream>

#include <vector>

#include <omp.h>

#include <iomanip>

void matrixMultiply(float\* A, float\* B, float\* C, int N) {

#pragma omp parallel for

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

C[i\*N + j] = 0;

for (int k = 0; k < N; k++) {

C[i\*N + j] += A[i\*N + k] \* B[k\*N + j];

}

}

}

}

void runBenchmark(const std::vector<int>& matrix\_sizes, const std::vector<int>& threads\_list) {

// Открываем файл для записи результатов

std::ofstream outFile("benchmark\_results.csv");

if (!outFile.is\_open()) {

std::cerr << "Error opening output file!" << std::endl;

return;

}

// Настройка формата вывода

outFile << std::fixed << std::setprecision(7);

// Записываем заголовок с разделителем-точкой с запятой

outFile << "N (Matrix size)";

for (int threads : threads\_list) {

outFile << ";" << threads << " threads"; // Используем точку с запятой

}

outFile << "\n";

// Запускаем тесты для каждого размера матрицы

for (int N : matrix\_sizes) {

outFile << N;

// Для каждого количества потоков

for (int threads : threads\_list) {

// Выделение памяти

float\* A = new float[N\*N];

float\* B = new float[N\*N];

float\* C = new float[N\*N];

// Инициализация матриц

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

A[i\*N + j] = (i + 1) \* (j + 1);

B[i\*N + j] = (i + 1) + 2 \* (j + 1);

}

}

// Установка числа потоков

omp\_set\_num\_threads(threads);

matrixMultiply(A, B, C, N);

// Замер времени

double start = omp\_get\_wtime();

matrixMultiply(A, B, C, N);

double elapsed = omp\_get\_wtime() - start;

// Запись результата с разделителем

outFile << ";" << elapsed;

// Освобождение памяти

delete[] A;

delete[] B;

delete[] C;

std::cout << "N = " << N << ", threads = " << threads

<< ", time = " << elapsed << " sec\n";

}

outFile << "\n";

}

outFile.close();

std::cout << "\nResults saved to results.csv\n";

std::cout << "NOTE: When opening in Excel, select semicolon (;) as delimiter\n";

}

int main() {

// Параметры тестирования

std::vector<int> matrix\_sizes;

for (int N = 10; N <= 100; N += 5) {

matrix\_sizes.push\_back(N);

}

std::vector<int> threads\_list = { 1, 4, 5, 8 };

// Запуск тестов

runBenchmark(matrix\_sizes, threads\_list);

return 0;

}

Приложение 3

Аппаратные харакеристики:

Процессор: AMD Ryzen 5 5500U (6 ядер / 12 потоков, 2.1 – 4.0 GHz, TDP 15 Вт)

* Графика\*\*: Radeon Vega 7 (встроенная)
* Техпроцесс: 7 нм
* Кэш\*\*: L2 — 3 MB, L3 — 8 MB

Оперативная память:

* 1 модуль DDR4 8 ГБ, 3200 МГц  
    
  Программное обеспечение:

Название ОС: Microsoft Windows 10 Pro

Версия 22H2: Сборка ОС 19045.5247

Тип системы: x64-based PC

Для работы с кодом было использовано: Visual Studio 2017 Pro