Отчёт по выполнению  
Лабораторной работы №4  
«Использование OpenMP»

Выполнил:  
студент гр. МОиАИС 184-1  
 Ибраев Ерлан Иржанович

**Цель работы:** с помощью OpenMP распараллелить несколькими способами и оценить их эффективность.

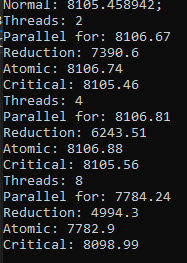
**Задание №1**

Работа происходит с квадратной матрицей А вещественных чисел размером NxN. Матрица вычисляется по формуле.

Найти среднее арифметическое элементов матрицы.  
Порядок выполнения:

1. В качестве N выбрать максимально доступное для реализации число, кратное 100.
2. Реализовать последовательный вариант и определить время его работы.
3. Выделить участки последовательного и распараллеливаемого кода.
4. Выполнить распараллеливание в OpenMP (четырьмя способами: parallel for, reduction, atomic, критические области) и оценить их эффективность (на основе среднего из 20 эксперимен-тов). Количество потоков М=2,4,8.
5. В качестве контроля корректности распараллеливания сравнить найденные средние с по-лученным в последовательном режиме.

Скриншот результата программы:



double Summ(int start, int end, long double\*\* matrix)

{

long double sum = 0.0;

for (int i = start; i < end; i++)

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

sum += matrix[i][j];

return sum;

}

void Task1(int k)

{

long double\*\* matrix = new long double\* [N];

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

{

matrix[i] = new long double[N];

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

matrix[i][j] = sin(i) + cos(j);

}

double sum = 0.0, average1 = 0.0, average2 = 0.0, average3 = 0.0, average4 = 0.0;

sum = Summ(0, N, matrix);

cin << "Thread count: " << k << endl;

const int Replays = 20;

for (int m = 1; m <= Replays; m++)

{

sum = 0.0;

int half = N / k;

int start = 0, end = half, i = 1;

#pragma omp parallel for num\_threads(k)

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

sum += Summ(i, i + 1, matrix);

average1 += sum / N \* N;

sum = 0.0;

half = N / k;

start = 0, end = half, i = 1;

#pragma omp parallel num\_threads(k) reduction(+: sum)

{

sum = Summ(start, end, matrix);

start = end;

i++;

end = half \* i;

if (i == k)

end = N;

}

average2 += sum / N \* N;

sum = 0.0;

#pragma omp parallel num\_threads(k)

{

#pragma omp atomic

sum += Summ(start, end, matrix);

start = end;

i++;

end = half \* i;

if (i == k)

end = N;

}

average3 += sum / N \* N;

sum = 0.0;

half = N / k;

start = 0, end = half, i = 1;

#pragma omp parallel num\_threads(k)

{

#pragma omp critical (crit)

{

sum += Summ(start, end, matrix);

}

start = end;

i++;

end = half \* i;

if (i == k)

end = N;

}

average4 += sum / N \* N;

}

cout << "Parallel for: " << (average1 / 20.0) << endl;

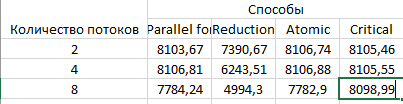
cout << "Reduction: " << (average2 / 20.0) << endl;

cout << "Atomic: " << (average3 / 20.0) << endl;

cout << "Critical: " << (average4 / 20.0) << endl;

}

Таблица результатов



Вывод: Смотря на результаты можно сказать, что самым оптимальным вариантом распараллеливания является использование reduction.

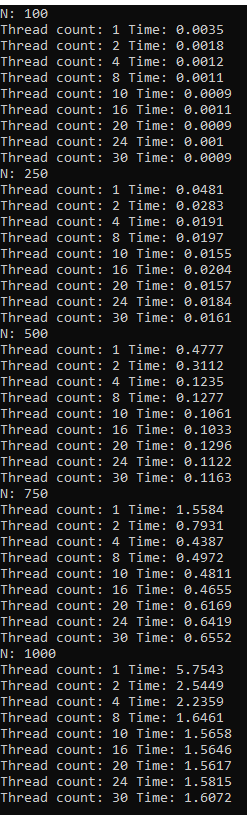
**Задание №2**

Проведите серию экспериментов на персональном компьютере по исследованию масштабируемости OpenMP-программ умножения квадратных матриц размера NхN. Привести данные в виде таблицы.

На основании данных таблицы постройте график масштабируемости для каждого значения параметра N. Определите для каждого графика, при каком количестве нитей достигается максимальное ускорение.

В качестве времени работы выбирайте среднее время для 20 испытаний по данным средних двух квартилей.

Скриншот результата программы:



void MatrixMult(double\*\* Matrix, double\*\* Result, int N, int Threads)

{

double sum = 0;

#pragma omp parallel for num\_threads(Threads) reduction(+:sum)

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

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

{

sum = 0;

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

sum += Matrix[i][k] \* Matrix[k][j];

Result[i][j] = sum;

}

}

void Task2() {

int N[] = { 100, 250, 500, 750, 1000 };

int Threads[] = { 1,2,4,8,10,16,20,24,30 };

srand(time(NULL));

double\*\* Array;

double\*\* Result;

for (int k = 0; k < 5; k++)

{

cout << "N: " << N[k] << endl;

Array = new double\* [N[k]];

Result = new double\* [N[k]];

for (int i = 0; i < N[k]; i++)

{

Array[i] = new double[N[k]];

Result[i] = new double[N[k]];

}

for (int i = 0; i < N[k]; i++)

for (int j = 0; j < N[k]; j++)

Array[i][j] = rand() % 20 + 1;

for (int l = 0; l < 9; l++)

{

double Average = 0;

for (int p = 0; p < 20; p++)

{

clock\_t start, end;

start = clock();

MatrixMult(Array, Result, N[k], Threads[l]);

end = clock();

double dt = (double)(end - start) / CLOCKS\_PER\_SEC;

if (p >= 5 && p <= 14)

Average += dt;

}

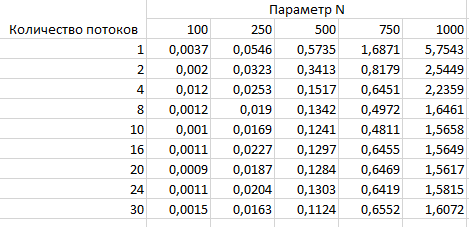
Average /= 10;

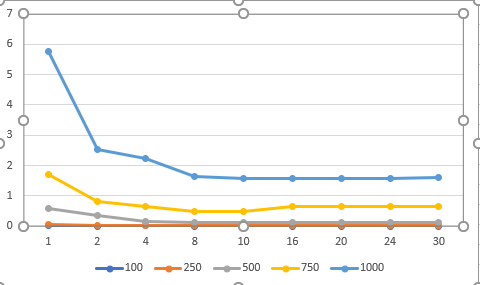
cout << "Thread count: " << Threads[l] << " Time: " << Average << endl;

}

}

}





Вывод: выполнялся на процессоре intel® core™ i7-8750H CPU @ 2.20GHz при 12 потоковых процессов. При любом значении N увеличение количество потоков приводит сокращение времени пропорционально, при этом наилучшее условие будет если количество потоков количеству потоковых процессов.