Лабораторная работа №3.  
«Основы OpenMP»

1. **Информация о студенте** – Поляков Игорь Андреевич, 184 МОиАИС;
2. **Цель работы**: Знакомство с базовыми элементами технологии распараллеливания OpenMP.
3. **Задания и ответы к ним:**
4. **Код проекта: https://github.com/polyakovGit/AaToPP\_LabWork\_3**

**Задание 1. Общие и частные переменные в OpenMP: параметр reduction.**

А) Напишите программу, в которой две нити параллельно вычисляют сумму чисел от 1 до N. Распределите работу по нитям с помощью оператора if языка С. Для сложения результатов вычисления нитей воспользуйтесь OpenMP-параметром reduction. Выполните расчёты для N ∈ {1,2,5,10,100,1000}

Входные данные: целое число N – количество чисел.

Выходные данные:

каждая нить выводит свою частичную сумму в формате

«[Номер\_нити]: Sum = <частичная\_сумма>»,

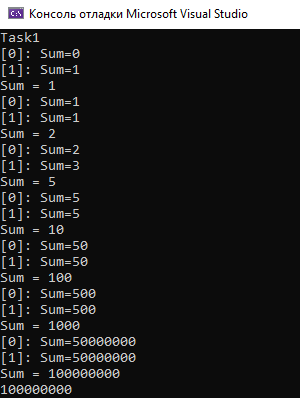
один раз выводится общая сумма в формате «Sum = <сумма>».

**[0]: Sum = 3**

**[1]: Sum = 7**

**Sum = 10**

**Скриншоты результатов**



**Код метода**

void Task1(int N) {

int sum = 0;

#pragma omp parallel num\_threads(2) reduction(+:sum)

{

int currThread = omp\_get\_thread\_num();

//sum+=Work(omp\_get\_thread\_num()\*N/omp\_get\_num\_threads();(omp\_get\_thread\_num()+1)\*N/omp\_get\_num\_threads()); для k нитей

if (currThread == 0) {

sum += WorkSum(0, N / 2);

printf("[%d]: Sum=%d\n", currThread, sum);

}

if (currThread == 1) {

sum += WorkSum(N / 2, N);

printf("[%d]: Sum=%d\n", currThread, sum);

}

}

printf("Sum = %d\n", sum);

}

**Выводы по заданию.**

Можно передавать работу конкретному потоку.

**Задание 2. Общие и частные переменные в OpenMP: параметр reduction.**

Б) Модифицируйте программу таким образом, чтобы она работала для k нитей. Покажите работу программы для N ∈ {2,10,100,1000}, k ∈ {2,4,8,16}.

Входные данные: целое число k – количество нитей, целое число N – количество чисел.

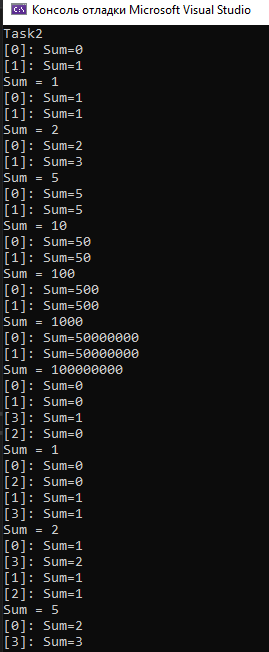
**[0]: Sum = 1**

**[1]: Sum = 2**

**[2]: Sum = 0**

**Sum = 3**

**Скриншоты результатов.**



**Код.**

printf("Task2\n");

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

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

start = clock();

Task2(arrNumbers[i], arrThreads[k]);

end = clock();

printf("time %f\n", (double)(end - start) / CLOCKS\_PER\_SEC);

}

}

void Task2(int N, int k) {

int sum = 0;

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

{

//sum += WorkSum(omp\_get\_thread\_num()\*N/omp\_get\_num\_threads(),(omp\_get\_thread\_num()+1)\*N/omp\_get\_num\_threads());

int currThread = omp\_get\_thread\_num();

sum += WorkSum(currThread \* N / k, (currThread + 1) \* N / k);

printf("[%d]: Sum=%d\n", currThread, sum);

}

printf("Sum = %d\n", sum);

}

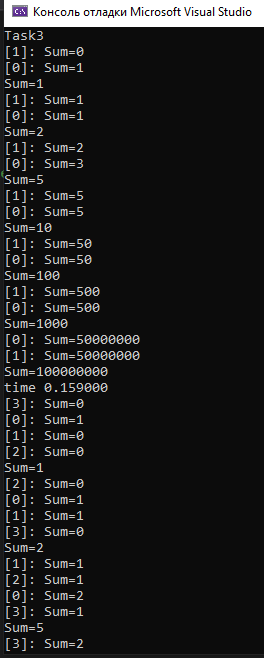
**Задание 3. Распараллеливание циклов в OpenMP: программа «Сумма чисел»**

Изучите OpenMP-директиву параллельного выполнения цикла for. Напишите программу, в которой k нитей параллельно вычисляют сумму чисел от 1 до N. Распределите работу по нитям с помощью OpenMP-директивы for. Покажите работу программы для N ∈ {2,10,100,1000}, k ∈ {2,4,8,16}.

Входные данные: целое число k – количество нитей, целое число N – количество чисел.

Выходные данные: каждая нить выводит свою частичную сумму в формате «[Номер\_нити]: Sum = <частичная\_сумма>», один раз выводится общая сумма в формате «Sum = <сумма>».

**Скриншоты результатов.**



**Код.**

printf("Task3\n");

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

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

start = clock();

Task3(arrNumbers[i], arrThreads[k]);

end = clock();

}

printf("time %f\n", (double)(end - start) / CLOCKS\_PER\_SEC);

}

void Task3(int N, int k) {

int sum = 0;

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

{

#pragma omp for

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

sum += 1;

printf("[%d]: Sum=%d\n", omp\_get\_thread\_num(), sum);

}

printf("Sum=%d\n", sum);

}

**Выводы по заданию.**

Для распределения между потоками итераций используется директива for.

**Задание 4. Распараллеливание циклов в OpenMP: параметр schedule**

Изучите параметр schedule директивы for. Модифицируйте программу «Сумма чисел» из задания 3 таким образом, чтобы дополнительно выводилось на экран сообщение о том, какая нить, какую итерацию цикла выполняет:

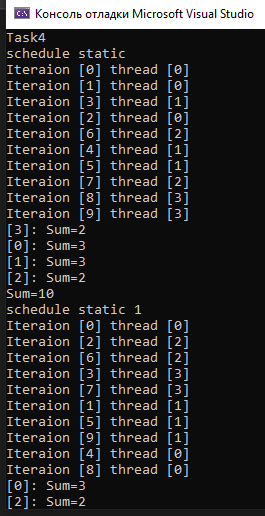
[<Номер нити>]: calculation of the iteration number <Номер итерации>.

Задайте k = 4, N = 10. Заполните следующую таблицу распределения итераций цикла по нитям в зависимости от параметра schedule:



Объясните полученные результаты.

**Скриншоты результатов.**



**Код.**

void Task4() {

int k = 4, N = 10;

int sum = 0;

printf("schedule static\n");

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

{

#pragma omp for schedule(static)

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

sum += 1;

printf("Iteraion [%d] thread [%d]\n", i, omp\_get\_thread\_num() );

}

printf("[%d]: Sum=%d\n", omp\_get\_thread\_num(), sum);

}

printf("Sum=%d\n", sum);

sum = 0;

printf("schedule static 1\n");

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

{

#pragma omp for schedule(static,1)

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

sum += 1;

printf("Iteraion [%d] thread [%d]\n", i, omp\_get\_thread\_num());

}

printf("[%d]: Sum=%d\n", omp\_get\_thread\_num(), sum);

}

printf("Sum=%d\n", sum);

sum = 0;

printf("schedule static 2\n");

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

{

#pragma omp for schedule(static,2)

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

sum += 1;

printf("Iteraion [%d] thread [%d]\n", i, omp\_get\_thread\_num());

}

printf("[%d]: Sum=%d\n", omp\_get\_thread\_num(), sum);

}

printf("Sum=%d\n", sum);

sum = 0;

printf("schedule dynamic\n");

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

{

#pragma omp for schedule(dynamic)

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

sum += 1;

printf("Iteraion [%d] thread [%d]\n", i, omp\_get\_thread\_num());

}

printf("[%d]: Sum=%d\n", omp\_get\_thread\_num(), sum);

}

printf("Sum=%d\n", sum);

sum = 0;

printf("schedule dynamic 2\n");

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

{

#pragma omp for schedule(dynamic,2)

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

sum += 1;

printf("Iteraion [%d] thread [%d]\n", i, omp\_get\_thread\_num());

}

printf("[%d]: Sum=%d\n", omp\_get\_thread\_num(), sum);

}

printf("Sum=%d\n", sum);

sum = 0;

printf("schedule guided\n");

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

{

#pragma omp for schedule(guided)

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

sum += 1;

printf("Iteraion [%d] thread [%d]\n", i+1, omp\_get\_thread\_num());

}

printf("[%d]: Sum=%d\n", omp\_get\_thread\_num(), sum);

}

printf("Sum=%d\n", sum);

sum = 0;

printf("schedule guided 2\n");

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

{

#pragma omp for schedule(guided,2)

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

sum += 1;

printf("Iteraion [%d] thread [%d]\n", i + 1, omp\_get\_thread\_num());

}

printf("[%d]: Sum=%d\n", omp\_get\_thread\_num(), sum);

}

printf("Sum=%d\n", sum);

}

**Таблица.**

****

**Выводы по заданию.**

Нельзя предсказать распределение итераций для потоков при динамическом или

управляемом расписании.

**Код.**

printf("Task5\n");

long long n = 2;

int i = 0;

double eps = 0.001, time;

printf("\t");

for (int k = 1; k <= 20; ++k)

printf("\t%d\t", k);

do {

printf("\n%.10f\t", eps);

for (int k = 1; k <= 20;) {

start = clock();

double myPi = Task5(n, k);

end = clock();

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

if (fabs(M\_PI - myPi) > eps)n \*= 2;

else {

printf("%.10f\t", time);

if (time > 120)break;

++k;

}

}

eps \*= 0.1;

} while (time <= 120);

double Task5(long long n, int k) {

double sum = 0, x, step = 1. / n;//a=0,b=1 границы интеграла;(b-a)/(double)n в общем виде

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

for (long long i = 0; i < n - 1; ++i) {//n-1 для левых

x = i \* step;//i+0.5 если средних; a+i\*step но здесь частный случай для pi 0, сумма не нужна

sum += 4 / (1 + x \* x);

}

return sum / n;

}

**Задание 5. Распараллеливание циклов в OpenMP: параметр schedule**

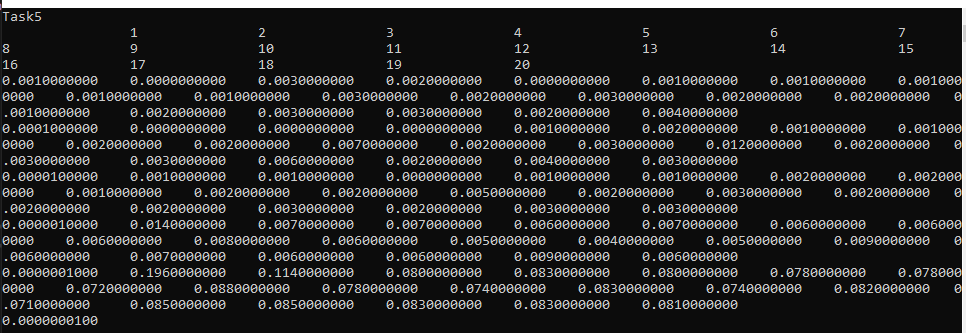
Реализовать вычисление числа π с требуемой точностью ε по формуле

Для вычисления интеграла использовать метод левых прямоугольников.

Исследовать зависимость времени работы алгоритма от числа потоков {k=1,2,3,…,20} и желаемой точности (ε=0.001, 0.0001, 0.00001, …, 0.000000001, … - остановка, когда время последнего расчёта превысило 2 мин). Полученную зависимость показать на графике.

Для достижения необходимой точности на каждом шаге увеличивать количество разбиений в 2 раза, пока уточнение результата не станет изменяться меньше, чем на ε.

**Скриншоты результатов.**



**Код.**

printf("Task5\n");

std::ofstream OutputData;

OutputData.open("OutputData.csv");

long long n = 2;

int i = 0;

double eps = 0.001, time;

printf("\t");

OutputData << ';';

for (int k = 1; k <= 20; ++k) {

printf("\t%d\t", k);

OutputData <<';' <<k;

}

do {

printf("\n%.10f\t", eps);

OutputData << '\n' << eps << ';';

for (int k = 1; k <= 20;) {

start = clock();

double myPi = Task5(n, k);

end = clock();

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

if (fabs(M\_PI - myPi) > eps)n \*= 2;

else {

printf("%.10f\t", time);

OutputData << time << ';';

if (time > 120)break;

++k;

}

}

eps \*= 0.1;

} while (time <= 120);

**Таблица.**



**График.**

**Выводы.**

Незначительная разница скорости вычислений при превышении количества потоков логических процессоров.