

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники

ОТЧЁТ
по лабораторной работе №3
по дисциплине «Операционные системы»
Тема: Процессы и потоки

Студент гр. 2307

Подберёзский А.Д.

Преподаватель

Тимофеев А.В.

Санкт-Петербург

2024

1. Введение

Тема работы: Процессы и потоки.

Цель работы: исследовать механизмы создания и управления процессами и потоками в ОС Windows.

Указания к выполнению

Задание 3.1. Реализация многопоточного приложения с использованием функций Win32 API.

1. Создайте приложение, которое вычисляет число π с точностью N знаков после запятой по следующей формуле

$$\pi = \left(\frac{4}{1+x_0^2} + \frac{4}{1+x_1^2} + \dots + \frac{4}{1+x_{N-1}^2} \right) \times \frac{1}{N}, \text{ где } x_i = (i+0.5) \times \frac{1}{N}, i = \overline{0, N-1}$$

, где $N = 100000000$.

– Используйте распределение итераций блоками (размер блока = $10 * N_{\text{студбилета}}$) по потокам. Сначала каждый поток по очереди получает свой блок итераций, затем тот поток, который заканчивает выполнение своего блока, получает следующий свободный блок итераций. Освободившиеся потоки получают новые блоки итераций до тех пор, пока все блоки не будут исчерпаны.

– Создание потоков выполняйте с помощью функции Win32 API CreateThread.

– Для реализации механизма распределения блоков итераций необходимо сразу в начале программы создать необходимое количество потоков в приостановленном состоянии, для освобождения потока из приостановленного состояния используйте функцию Win32 API ResumeThread.

– По окончании обработки текущего блока итераций поток не должен завершаться, а должен быть, например, приостановлен с

помощью функции Win32 API SuspendThread. Затем потоку должен быть предоставлен следующий свободный блок итераций, и поток должен быть освобождён, например, с помощью функции Win32 API ResumeThread.

2. Произведите замеры времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12, 16). По результатам измерений постройте график и определите число потоков, при котором достигается наибольшая скорость выполнения. Запротоколируйте результаты в отчёт.

3. Подготовьте итоговый отчёт с развёрнутыми выводами по заданию.

Задание 3.2. Реализация многопоточного приложения с использованием технологии OpenMP.

Указания к выполнению.

1. Создайте приложение, которое вычисляет число π с точностью N знаков после запятой по следующей формуле

$$\pi = \left(\frac{4}{1+x_0^2} + \frac{4}{1+x_1^2} + \dots + \frac{4}{1+x_{N-1}^2} \right) \times \frac{1}{N}, \text{ где } x_i = (i+0.5) \times \frac{1}{N}, i = \overline{0, N-1}$$

, где $N = 100000000$.

– Распределите работу по потокам с помощью OpenMP-директивы for.

– Используйте динамическое планирование блоками итераций (размер блока = $10 * N_{\text{студбилета}}$).

2. Произведите замеры времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12, 16). По результатам измерений постройте график и определите число потоков, при котором достигается наибольшая скорость выполнения. Запротоколируйте результаты в отчёт, сравните с результатами прошлой работы.

3. Подготовьте итоговый отчёт с развёрнутыми выводами по заданию.

2. Реализация многопоточного приложения с использованием функций Win32 API

2.1. Замеры времени выполнения приложения для разного числа потоков

Многопоточное приложение выводит результаты выполнения работы – время вычисления числа пи и само число пи – в терминал. В работе было сделано 5 замеров.

```
Multiprocessing: Win32 API.  
Threads number: 1 Time: 844 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 2 Time: 469 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 4 Time: 266 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 8 Time: 172 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 12 Time: 172 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 16 Time: 141 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
```

Рисунок 1: Замер времени выполнения 1 с использованием технологии Win 32 API

```
Threads number: 1 Time: 860 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 2 Time: 484 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 4 Time: 265 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 8 Time: 187 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 12 Time: 172 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 16 Time: 188 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
```

Рисунок 2: Замер времени выполнения 2 с использованием технологии Win 32 API

```
Threads number: 1 Time: 844 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 2 Time: 484 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 4 Time: 266 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 8 Time: 172 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125  
Threads number: 12 Time: 156 ms %pi: 3.1415926335897886752246410679845212143845856189727783203125  
Threads number: 16 Time: 156 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
```

Рисунок 3: Замер времени выполнения 3 с использованием технологии Win 32 API

```
Threads number: 1 Time: 828 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
Threads number: 2 Time: 500 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
Threads number: 4 Time: 297 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
Threads number: 8 Time: 203 ms %pi: 3.1415926335897898674133499330451968489796854555606842041015625
Threads number: 12 Time: 188 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
Threads number: 16 Time: 172 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
```

Рисунок 4: Замер времени выполнения 4 с использованием
технологии Win 32 API

```
Threads number: 1 Time: 875 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
Threads number: 2 Time: 500 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
Threads number: 4 Time: 281 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
Threads number: 8 Time: 188 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
Threads number: 12 Time: 156 ms %pi: 3.14159263358978927131899550051485903168213553726673126220703125
Threads number: 16 Time: 156 ms %pi: 3.1415926335897886752246410679845212143845856189727783203125
```

Рисунок 5: Замер времени выполнения 5 с использованием
технологии Win 32 API

2.2. Вывод по реализации многопоточного приложения с использованием технологии Win32 API



Рисунок 6: График зависимости времени выполнения задачи от количества задействованных потоков

В многопоточном приложении с использованием технологии Win32 API по графику среднего значения по 5 замерам видно, что спад времени выполнения задачи (вычисления числа π) прекращается с 8 потоков. Это связано с тем, что разделение задач между четырьмя, а затем и восемью потоками существенно оптимизирует работу, равномерно распределяя её по нескольким потокам, где она будет выполняться быстрее и примерно одинаковое время, вместо одного, где она будет выполняться долго. Наибольшая скорость выполнения была достигнута при 16 потоках (среднее – 162,6 мс), но стабилизация скорости выполнения была достигнута на 8 потоках (среднее – 184,4 мс).

Общая производительность (в данном случае – величина времени на графике) с увеличением числа задействованных потоков (в частности, на графике – с восьми потоков) растёт намного

медленнее, чем в начале, а также может упасть (в частности, на графике – на двенадцати потоках). Это объясняется тем, что наибольшая производительность достигается при количестве потоков, равным количеству процессов. В данном случае, замеры проводились на компьютере с 6 физическими ядрами, но 12 логическими ядрами, что позволило эффективно распределить 8 потока. При большом количестве потоков необходимо вытеснять одни потоки другими, чтобы у них была возможность выполнять задачи. Поэтому время тратится на ожидание других потоков, на операции планирования и на другое.

2.3. Исходный код программы

```
#include <iostream>
#include <windows.h>
#include <string>
#include <iomanip>
#include <list>
#include <numeric>

using namespace std;

// ----- VARIABLES INITIALIZATION -----

DWORD startTime = 0; // starting counting pi-number point
DWORD finishTime = 0; // ending counting pi-number point
DWORD allTime = -1; // milliseconds, which will take the pi counting
size_t blocksIterator; // current block
size_t blocksNumber; // number of all blocks
HANDLE synchIteration; // synchronizing iteration mutex
HANDLE synchSummary; // synchronizing summary mutex
const size_t BLOCKSIZE = 10 * 230727; // iteration distribution for threads
const size_t N = 100000000; // N iterations (not signs after comma)

list<long double> list1; // list of the all parts to summary
long double summaryResult = 0.0; // final pi result for each number of threads

// ----- FUNCTION DECLARATION -----

DWORD WINAPI countingPI (LPVOID localInThreads);
void preparingPI (int localNumberOfThreads);

// ----- MAIN -----

int main (int argc, char **argv)
```

```

{
    int numberOfThreads[] = {1, 2, 4, 8, 12, 16}; // number of threads
    int arraySize = sizeof(numberOfThreads)/sizeof(numberOfThreads[0]); // counting
an array size

    long double piNumber; // the %pi number
    cout << "\nMultiprocessing: Win32 API.\n";

    for (int i = 0; i < arraySize; i++)
    {
        list1.clear();
        summaryResult = 0.0;
        preparingPI(numberOfThreads[i]);
        cout << "\nThreads number: " << numberOfThreads[i] << " Time: " << allTime
<< " ms" << setprecision(N) << " %pi: " << summaryResult << "\n";
    }
    return 0;
}

```

// ----- FUNCTION'S BODY -----

DWORD WINAPI countingPI(LPVOID localInThreads)

```

{
    int i; // iterator
    int localIndicator = 1; // end indicator
    size_t startPoint = 2;
    size_t endPoint = 1;
    long double localResult = 0.0; // local result summary

    while (localIndicator != 0)
    {
        // SYNCHRONIZING ITERATIONS -- START

        DWORD waitError = WaitForSingleObject (synchIteration, INFINITE); // while
isn't released, i can't quit
    }
}

```

```

if (waitError != WAIT_OBJECT_0)
{
    cout << "Sorry, you have error w/ synchIteration (" << waitError << "). Last
error number: " << GetLastError() << "\n";
}

```

```

if (blocksIterator < blocksNumber)
{
    startPoint = blocksIterator * BLOCKSIZE; // blocksize number start
(iteration*number_of_items_in_block)
    endPoint = (blocksIterator + 1) * BLOCKSIZE - 1; // blocksize number end
(iteration*number) // HERE CHANGED FORMULA
    if (endPoint > N - 1) // checking for out of range error
    {
        endPoint = N - 1;
    }
    blocksIterator = blocksIterator + 1; // increasing iteration number
}
else
{
    startPoint = 2;
        endPoint = 1;
}

```

```

ReleaseMutex (synchIteration);

```

```

// SYNCHRONIZING ITERATIONS -- END

```

```

if (startPoint <= endPoint)
{
    localResult = 0.0;

    for (i = startPoint; i++ <= endPoint; ) // formula counting

```

```

    {
        localResult = localResult + (4 / (1 + (((long double)i + 0.5) / (long
double)N)*(((long double)i + 0.5) / (long double)N)));
    }

// SYNCRONIZING SUMMARY -- BEGIN

waitError = WaitForSingleObject (synchSummary, INFINITE);

if (waitError != WAIT_OBJECT_0)
{
    cout << "Sorry, you have problem w/ synchSummary (" << waitError << ").
Last error number: " << GetLastError() << endl;
}

list1.push_back(localResult); // adding the result to the list

ReleaseMutex (synchSummary);

// SYNCHRONIZING SUMMARY -- END
}
else
{
    localIndicator = 0;
}
}

return 0;
}

void preparingPI (int localNumberOfThreads)
{
    // 1 -- PREPARING AND INITIALIZING

```

```

// initilaizing objects and variables

blocksIterator = 0; // setting to null block iterator
blocksNumber = N % BLOCKSIZE == 0 ? (N / BLOCKSIZE) : (N / BLOCKSIZE
+ 1); // if div is full or not
int i = 0; // iterator
HANDLE *threadsArray = new HANDLE[localNumberOfThreads];
synchIteration = CreateMutex (NULL, FALSE, NULL); // synchronizing object for
selected iterations
synchSummary = CreateMutex (NULL, FALSE, NULL); // synchronizing object
for summary counting

// 2 -- CHECKING THREADS AND CREATING THREADS

// creating threads for counting pi-number (just creating and setting threads here)

for (i = 0; i < localNumberOfThreads; i++)
{
    threadsArray[i] = CreateThread (NULL, 0, countingPI, NULL,
CREATE_SUSPENDED, NULL);
}

// 3 -- COUNTING PI-NUMBER

// starting the timer

startTime = GetTickCount();

// starting threads for counting pi-number (just starting here)

for (unsigned i = 0; i < localNumberOfThreads; i++)
{
    ResumeThread (threadsArray[i]);
}

```

```

// waiting until all threads will be released

DWORD    waitError    =    WaitForMultipleObjects(localNumberOfThreads,
threadsArray, true, INFINITE);

// making the final result

summaryResult = std::accumulate(std::begin(list1), std::end(list1), 0.0);
summaryResult = summaryResult / N;

// ending the timer

finishTime = GetTickCount();

// counting final time

allTime = finishTime - startTime;

// 4 -- ENDING AND CLEANING

// "cleaning": closing handles and cleaning memory

for (i = 0; i < localNumberOfThreads; ++i)
{
    CloseHandle(threadsArray[i]);
}
CloseHandle(synchIteration);
CloseHandle(synchSummary);
delete threadsArray;
}

```

2.4. Вывод

В ходе выполнения первой части («Реализация многопоточного приложения с использованием технологии Win32 API») лабораторной работы 3 «Процессы и потоки» была освоена технология распараллеливания на основе Win32API. В частности, с помощью технологии распараллеливания Win32 API в программе было вычислено число пи с размером блока итерации для каждого потока $10 \cdot 230727$ и общим количеством итераций 100000000. В работе были проведены замеры для 1, 2, 4, 8, 12 и 16 потоков соответственно, а также составлены графики зависимости времени от количества потоков на данную задачу. Была выявлена зависимость времени числа задействованных потоков от числа логических процессов. Таким образом и было реализовано многопоточное приложение с использованием технологии Win32 API.

3. Реализация многопоточного приложения с использованием технологии OpenMP

3.1. Замеры времени выполнения приложения для разного числа потоков

Многопоточное приложение выводит результаты выполнения работы – время вычисления числа пи и само число пи – в терминал. В работе было сделано 5 замеров.

```
Threads count: 1 Time: 750 ms pi: 3.14159265358979355509984332517348093460896052420139312744140625
Threads count: 2 Time: 375 ms pi: 3.14159265358979299630204362614449564716778695583343505859375
Threads count: 4 Time: 235 ms pi: 3.14159265358979323244127679348736137399100698530673980712890625
Threads count: 8 Time: 156 ms pi: 3.14159265358979315589660341601074833306483924388885498046875
Threads count: 12 Time: 141 ms pi: 3.1415926535897931936268390185063026365241967141628265380859375
Threads count: 16 Time: 141 ms pi: 3.14159265358979323851280895940618620443274267017841339111328125
```

Рисунок 7: Замер времени выполнения 1 с использованием технологии OpenMP

```
Threads count: 1 Time: 735 ms pi: 3.14159265358979355509984332517348093460896052420139312744140625
Threads count: 2 Time: 422 ms pi: 3.141592653589792995434681888156092099961824715137481689453125
Threads count: 4 Time: 250 ms pi: 3.14159265358979331223855668842048771693953312933444976806640625
Threads count: 8 Time: 172 ms pi: 3.1415926535897932578116076296481651297654025256633758544921875
Threads count: 12 Time: 157 ms pi: 3.1415926535897932205150528961468125999090261757373809814453125
Threads count: 16 Time: 172 ms pi: 3.14159265358979320468570117785844786340021528303623199462890625
```

Рисунок 8: Замер времени выполнения 2 с использованием технологии OpenMP


```
Threads count: 1 Time: 750 ms pi: 3.14159265358979355509984332517348093460896052420139312744140625
Threads count: 2 Time: 422 ms pi: 3.1415926535897929663780636655445732685620896518230438232421875
Threads count: 4 Time: 266 ms pi: 3.14159265358979331440696103339149658495443873107433319091796875
Threads count: 8 Time: 156 ms pi: 3.1415926535897932005657329224135310141718946397304534912109375
Threads count: 12 Time: 125 ms pi: 3.14159265358979315871552906447305986148421652615070343017578125
Threads count: 16 Time: 156 ms pi: 3.14159265358979323200759592449315960038802586495876312255859375
```

Рисунок 9: Замер времени выполнения 3 с использованием
технологии OpenMP

```
Threads count: 1 Time: 750 ms pi: 3.14159265358979355509984332517348093460896052420139312744140625
Threads count: 2 Time: 421 ms pi: 3.14159265358979298611054320478075396749773062765598297119140625
Threads count: 4 Time: 234 ms pi: 3.1415926535897932166119250751989966374821960926055908203125
Threads count: 8 Time: 172 ms pi: 3.1415926535897932300560320140192516191746108233928680419921875
Threads count: 12 Time: 157 ms pi: 3.14159265358979319514472205998600884413463063538074493408203125
Threads count: 16 Time: 172 ms pi: 3.14159265358979318994055163205558756089885719120502471923828125
```

Рисунок 10: Замер времени выполнения 4 с использованием
технологии OpenMP

```
Threads count: 1 Time: 750 ms pi: 3.14159265358979355509984332517348093460896052420139312744140625
Threads count: 2 Time: 437 ms pi: 3.14159265358979299630204362614449564716778695583343505859375
Threads count: 4 Time: 250 ms pi: 3.14159265358979328426614063829447331954725086688995361328125
Threads count: 8 Time: 156 ms pi: 3.1415926535897932265865850620656374303507618606090545654296875
Threads count: 12 Time: 157 ms pi: 3.14159265358979319037423250104978933450183831155300140380859375
Threads count: 16 Time: 156 ms pi: 3.14159265358979324458434112532501103487447835505008697509765625
```

Рисунок 11: Замер времени выполнения 5 с использованием
технологии OpenMP

3.2. Вывод по реализации многопоточного приложения с использованием технологии OpenMP

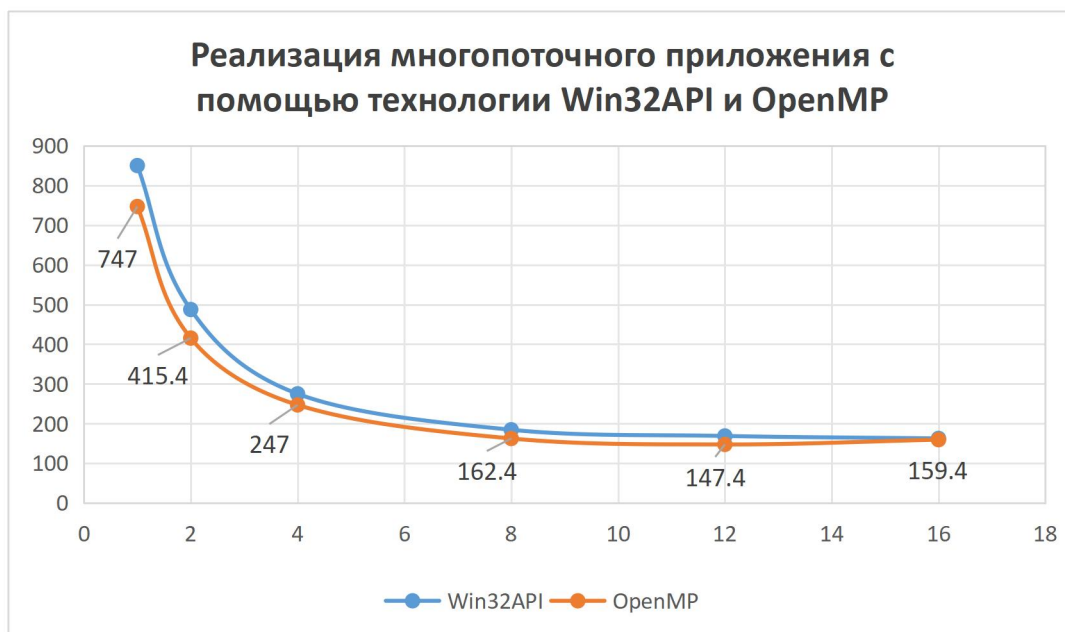


Рисунок 12: График зависимости времени выполнения задачи от количества задействованных потоков

В многопоточном приложении с использованием технологии OpenMP по графику среднего значения по 5 замерам видно, что спад времени выполнения задачи (вычисления числа пи) прекращается с 4 потоков. Данная тенденция аналогична графику многопоточного приложения с использованием технологии Win32 API. Наибольшая скорость выполнения была достигнута при 12 потоках (среднее – 147,4 мс), но стабилизация скорости выполнения была достигнута на 8 потоках (среднее – 162,4 мс).

На графике также видно, что использование технологии OpenMP при выполнении задачи занимает немного меньше времени, чем использование технологии Win32 API. Скорее всего, это связано с тем, что распределение крупной вычислительной задачи (в данном

случае – вычисление числа π) согласно данным Microsoft хорошо подходит к выполнению на стандарте OpenMP.

3.3. Исходный код программы

```
#include <iostream>
#include <windows.h>
#include <string>
#include <iomanip>
#include <omp.h>

using namespace std;

// ----- VARIABLES INITIALIZATION -----

DWORD startTime = 0; // starting counting pi-number point
DWORD finishTime = 0; // ending counting pi-number point
DWORD allTime = -1; // milliseconds, which will take the pi counting
const size_t BLOCKSIZE = 10 * 230727; // iteration distribution for threads
const size_t N = 100000000; // N iterations (not signs after comma)

// ----- FUNCTION DECLARATION -----

long double countingPI (size_t localIterations, size_t localBlocksize, int
localNumberOfThreads);

// ----- MAIN -----

int main (int argc, char **argv)
{
    int numberOfThreads[] = {1, 2, 4, 8, 12, 16}; // number of threads
    int arraySize = sizeof(numberOfThreads)/sizeof(numberOfThreads[0]); // counting
an array size
    long double piNumber; // the %pi number
    cout << "\nMultiprocessing: Open MP.\n";

    for (int i = 0; i < arraySize; i++)
```

```

{
    startTime = 0;
    finishTime = 0;
    allTime = 0;
    piNumber = countingPI(N, BLOCKSIZE, numberOfThreads[i]);
    cout << "\nThreads number: " << numberOfThreads[i] << " Time: " << allTime
    << " ms" << setprecision(N) << " %pi: " << piNumber << "\n";
}
return 0;
}

```

// ----- FUNCTION'S BODY -----

```

long double countingPI (size_t localIterations, size_t localBlocksize, int
localNumberOfThreads)
{
    // 1 -- COUNTING PI-NUMBER

    int i = 0; // iterator

    // starting the timer

    startTime = GetTickCount();

    long double summaryResult = 0.0;
    #pragma omp parallel shared(startTime, finishTime, allTime) reduction (+:
summaryResult) num_threads(localNumberOfThreads)
    {
        #pragma omp for schedule(dynamic, localBlocksize) nowait
        for (i = 0; i < localIterations; i++)
        {
            summaryResult = summaryResult + (4 / (1 + (((long double)i + 0.5) / (long
double)localIterations)*(((long double)i + 0.5) / (long double)localIterations)));
        }
    }
}

```

```
}

// making the final result

summaryResult = summaryResult / localIterations;

// ending the timer

finishTime = GetTickCount();

// counting final time

allTime = finishTime - startTime;

return summaryResult;
}
```

3.4. Вывод

В ходе выполнения второй части («Реализация многопоточного приложения с использованием технологии OpenMP») лабораторной работы 3 «Процессы и потоки» была освоена технология OpenMP, позволяющая на программном уровне осуществить распараллеливание приложения. В частности, с помощью технологии OpenMP в программе было вычислено число π с размером блока итерации для каждого потока $10 \cdot 230727$ и общим количеством итераций 100000000. В работе были проведены замеры для 1, 2, 4, 8, 12 и 16 потоков соответственно, а также составлены графики зависимости времени от количества потоков на данную задачу. Было осуществлено сравнение замеров с замерами технологии Win32 API, в результате чего была замечена более быстрая работа технологии OpenMP. Таким образом и было реализовано многопоточное приложение с использованием технологии OpenMP.

