МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский технологический университет “МИСиС”»

НИТУ МИСИС

Институт информационных технологий

Кафедра инфокоммуникационных технологий

ОТЧЕТ ПО ПРАКТИЧЕСКИМ РАБОТАМ ПО ДИСЦИПЛИНЕ «МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ» (OpenMP)

Выполнил студент: Маковецкий Иван

Группа: МИВТ-22-5

Москва — 2023

Оглавление

Стр.

Стр.

Стр.

Задание 1. Создание проекта в среде MS Visual Studio с поддержкой OpenMP

1. Создадим на рабочем столе папку «Маковецкий».
2. Запустим Microsoft Visual Studio 2010.
3. Создадим проект. Для этого выберем пункт в меню File -> New -> Project, или нажмем Ctrl+Shift+N.
4. В окне New Project в раскрывающемся списке Visual C++ выберем Win32. В подокне в середине выберем Win32 Console Application. Внизу введем имя проекта Name и место расположения проекта Location и нажмем кнопку OK.
5. В открывшемся окне Win32 Application Wizard — example1 нажмем кнопку Next, и затем в Additional options поставим галочку напротив Empty project. Нажмем кнопку Finish.
6. Теперь создадим файл с кодом приложения. Выберите пункт в меню Project -> Add New Item, или нажмите Ctrl+Shift+A. В категории Visual C++ выберем подкатегорию Code. В подокне в середине установим C++ File (.cpp). Введите имя файла, например, source, и нажмите кнопку Add.
7. В открывшемся окне source.cpp введем следующий код на языке C:

```
int main() {
    return 0;
}
```

   Сохраним файл, выбрав пункт меню File -> Save source.cpp, или нажав Ctrl+S.
8. Для компиляции приложения выберем пункт меню Debug -> Build Solution, или нажмем F7.
9. Для запуска приложения выберем пункт меню Debug -> Start Without Debugging, или нажмем Ctrl+F5.
10. Для включения поддержки OpenMP установим дополнительные параметры компиляции проекта:
    - В главном меню выберем Project-> Имя проекта Properties.

   - В открывшемся окне выберем Configuration Properties / C/C++ / Language. Установим для опции OpenMP Support значение Yes (/openmp).

11. Для компиляции приложения нажмем F7.
12. Для запуска приложения нажмем Ctrl+F5.

Задание 2. Многопоточная программа «Hello World!»

Напишите Open-MP программу, в которой создается 4 нити и каждая нить выводит на экран строку «Hello World!».

2.1 Входные данные

Нет.

2.2 Выходные данные

4 строки «Hello World!».

2.3 Пример входных и выходных данных

Входные данные	Выходные данные
	Hello World!
	Hello World!
	Hello World!
	Hello World!

2.4 Исходный код программы

```
#include <stdio.h>
#include <omp.h>
```

```
int main() {
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        printf("Hello World!\n");
    }

    return 0;
}
```

2.5 Результат выполнения программы

```
Hello World!
Hello World!
Hello World!
Hello World!
```

Задание 3. Программа «I am!»

Напишите программу, в которой создается k нитей, и каждая нить выводит на экран свой номер и общее количество нитей в параллельной области в формате:

```
I am <Номер нити> thread from <Количество нитей> threads!
```

3.1 Входные данные

k – количество нитей в параллельной области.

3.2 Выходные данные

k строк вида «I am <Номер нити> thread from <Количество нитей> threads!».

3.3 Пример входных и выходных данных

Входные данные	Выходные данные
3	I am 0 thread from 3 threads!
3	I am 1 thread from 3 threads!
3	I am 2 thread from 3 threads!

3.4 Исходный код программы

```
#include <stdio.h>
```

```
#include <omp.h>

int main() {
    int k = 3; // Количество нитей
    #pragma omp parallel num_threads(k)
    {
        int thread_num = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
        printf("I am %d thread from %d threads!\n", thread_num,
   num_threads);
    }

    return 0;
}
```

3.5 Результат выполнения программы

```
I am 0 thread from 3 threads!
I am 1 thread from 3 threads!
I am 2 thread from 3 threads!
```

3.6 Модифицированная программа для вывода только четных нитей

```
#include <stdio.h>
#include <omp.h>

int main() {
    int k = 3; // Количество нитей
    #pragma omp parallel num_threads(k)
    {
        int thread_num = omp_get_thread_num();
        if (thread_num % 2 == 0) {
            int num_threads = omp_get_num_threads();
            printf("I am %d thread from %d threads!\n",
   thread_num, num_threads);
```

```
        }
    }
    return 0;
}
```

3.7 Результат выполнения модифицированной программы

```
I am 0 thread from 3 threads!
I am 2 thread from 3 threads!
```

Задание 4. Общие и частные переменные в OpenMP: программа «Скрытая ошибка»

Изучите конструкции для управления работой с данными shared и private. Напишите программу, в которой создается k нитей, и каждая нить выводит на экран свой номер через переменную rank следующим образом:

```
rank = omp_get_thread_num();
printf("I am %d thread.\n", rank);
```

Экспериментами определите, общей или частной должна быть переменная rank.

```
I am <Номер нити> thread from <Количество нитей> threads!
```

4.1 Входные данные

Целое число k – количество нитей в параллельной области.

4.2 Выходные данные

k строк вида «I am <Номер нити>.».

4.3 Пример входных и выходных данных

Входные данные	Выходные данные
3	I am 0 thread.
3	I am 1 thread.
3	I am 2 thread.

4.4 Выполнение работы

Последовательно выполним каждый из пунктов и проанализируем результаты.

4.5 Определение переменной rank как общей

```
#include <stdio.h>
#include <omp.h>

int rank; // Объявление переменной rank как общей

int main() {
    int k = 3; // Количество нитей

    #pragma omp parallel num_threads(k)
    {
        rank = omp_get_thread_num();
        printf("I am %d thread.\n", rank);
    }

    return 0;
}
```

Результат: Эта программа не будет работать правильно. Переменная rank объявлена общей, поэтому все нити будут пытаться одновременно записывать и читать ее значение. Это может привести к непредсказуемым результатам, таким как перепутывание номеров нитей в выводе.

4.6 Добавление задержки (имитации длительных вычислений)

```
#include <stdio.h>
#include <omp.h>
#include <windows.h> // Для Sleep()
```

```
int main() {
    int k = 3; // Количество нитей

    #pragma omp parallel num_threads(k)
    {
        int rank = omp_get_thread_num();
        Sleep(100); // Имитация длительных вычислений
        printf("I am %d thread.\n", rank);
    }

    return 0;
}
```

Результат: Эта программа будет имитировать длительные вычисления с помощью функции Sleep(100) в каждой нити. В этом случае, каждая нить будет исполняться параллельно, результаты вывода в случайном порядке из-за того, что разные нити завершают свои задержки в разное время.

4.7 Переопределение переменной rank как частной

```
#include <stdio.h>
#include <omp.h>

int main() {
    int k = 3; // Количество нитей

    #pragma omp parallel private(rank) num_threads(k)
    {
        int rank = omp_get_thread_num();
        printf("I am %d thread.\n", rank);
    }

    return 0;
}
```

4.8 Результат работы программы

```
I am 0 thread.
I am 1 thread.
I am 2 thread.
```

Результат: При объявлении rank как частной, каждая нить будет иметь свою собственную копию переменной rank. В результате получим корректные номера нитей в выводе без перепутывания.

Общий вывод: В OpenMP важно правильно управлять доступом к переменным внутри параллельных областей. Общие переменные могут вызывать конфликты, если не синхронизировать доступ к ним, в то время как частные переменные гарантируют, что каждая нить имеет свою собственную копию переменной.

Задание 5. Общие и частные переменные в OpenMP: программа reduction

Напишите программу, в которой две нити параллельно вычисляют сумму чисел от 1 до N. Распределите работу по нитям с помощью оператора if языка C. Для сложения результатов вычисления нитей воспользуйтесь OpenMP-параметром reduction.

5.1 Входные данные

Целое число N — количество чисел,

5.2 Выходные данные

Каждая нить выводит свою частичную сумму в формате «[Номер нити]: Sum = <частичная сумма>», один раз выводится общая сумма в формате «Sum = <сумма>».

5.3 Пример входных и выходных данных

Входные данные	Выходные данные
4	[0]: Sum = 3
	[1]: Sum = 7
	Sum = 10

5.4 Выполнение работы

Используем OpenMP для параллельного вычисления суммы чисел от 1 до N. Воспользуемся директивой #pragma omp parallel for для распределения работы между нитями и параметром reduction для вычисления общей суммы

```
#include <stdio.h>
#include <omp.h>

int main() {
    int N = 4; // Количество чисел
    int sum = 0;

    #pragma omp parallel for reduction(+:sum) num_threads(2)
    for (int i = 1; i <= N; i++) {
        int thread_num = omp_get_thread_num();
        sum += i;
        printf("[%d]: Sum = %d\n", thread_num, sum);
    }

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

    return 0;
}
```

В этой программе:

- Мы используем директиву #pragma omp parallel for для создания параллельной области и распределения итераций цикла между нитями.
- С параметром reduction(+:sum) мы говорим OpenMP, что переменная sum должна считаться как общая сумма, а не как отдельные суммы для каждой нити.
- Мы выводим частичные суммы для каждой нити и общую сумму после завершения цикла.

5.5 Результат выполнения программы

Входные данные	Выходные данные
4	[0]: Sum = 3
	[1]: Sum = 7
	Sum = 10

Задание 6. 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.

6.1 Входные данные

Целое число k — количество нитей, целое число N — количество чисел.

6.2 Выходные данные

Каждая нить выводит свою частичную сумму в формате «[Номер нити]: Sum = <частичная сумма>», один раз выводится общая сумма в формате «Sum = <сумма>».

6.3 Пример входных и выходных данных

Входные данные	Выходные данные
2	[0]: Sum = 3
4	[1]: Sum = 7
	Sum = 10

6.4 Выполнение работы

Для решения этой задачи мы будем использовать OpenMP, библиотеку для параллельного программирования. В частности, мы будем использовать директиву `#pragma omp parallel for`, которая позволяет выполнять цикл for параллельно на нескольких потоках.

Сначала нам нужно установить количество потоков, которые будут использоваться для параллельного выполнения цикла. Это можно сделать с помощью директивы `#pragma omp parallel num_threads(k)`, где `k` — это количество потоков.

Мы используем директиву `#pragma omp for reduction(+:sum)`, чтобы распределить работу по потокам и суммировать результаты каждого потока в общем значении `sum`. Директива `reduction(+:sum)` гарантирует, что каждый поток будет иметь свою частную сумму, а затем все частные значения будут суммированы вместе, чтобы получить окончательное значение 1.

В конце мы выводим результаты каждого потока и общее значение суммы.

В этом коде `omp_get_thread_num()` используется для получения номера текущего потока.

```
#include <omp.h>
#include <iostream>

int main() {
    int k, N;
    std::cin >> k >> N;

    int sum = 0;
    #pragma omp parallel num_threads(k) reduction(+:sum)
    {
        int thread_sum = 0;
        #pragma omp for
        for (int i = 1; i <= N; i++) {
            thread_sum += i;
        }
        std::cout << "[" << omp_get_thread_num() <<
        "]: Sum = " << thread_sum << std::endl;
        sum += thread_sum;
    }
    std::cout << "Sum = " << sum << std::endl;
```

```
    return 0;
}
```

6.5 Результат выполнения программы

Входные данные	Выходные данные
2	[0]: Sum = 3
4	[1]: Sum = 7
	Sum = 10
2	[0]: Sum = 1
2	[1]: Sum = 2
	Sum = 3
3	[0]: Sum = 1
2	[1]: Sum = 2
	[2]: Sum = 0
	Sum = 3

Задание 7. Распараллеливание циклов в OpenMP: параметр schedule

Изучите параметр schedule директивы for. Модифицируйте программу «Сумма чисел» из задания 6 таким образом, чтобы дополнительно выводилось на экран сообщение о том, какая нить, какую итерацию цикла выполня- ет: [<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.

7.1 Выполнение работы

Наш код уже содержит параллельную область с директивой `for`. Для добавления вывода сообщения `"[<Номер нити>]: calculation of the iteration number <Номер итерации>."`, мы будем использовать функцию `omp_get_thread_num()`, которая возвращает номер текущей нити, и переменную `i`, которая представляет номер итерации.

```
#include <omp.h>
#include <iostream>

int main() {
  int k, N;
  std::cin >> k >> N;

  int sum = 0;
  #pragma omp parallel num_threads(k) reduction(+:sum)
  {
      int thread_sum = 0;
      #pragma omp for
      for (int i = 1; i <= N; i++) {
          std::cout << "[" << omp_get_thread_num() <<
"]: calculation of the iteration number " << i << ".\n";
          thread_sum += i;
      }
      std::cout << "[" << omp_get_thread_num() <<
"]: Sum = " << thread_sum << std::endl;
      sum += thread_sum;
```

```
    }
    std::cout << "Sum = " << sum << std::endl;

    return 0;
}
```

Чтобы добавить параметр schedule к директиве for, мы можем использовать следующий синтаксис: schedule(type[, chunk]). Здесь type может быть static, dynamic, guided или auto, а chunk — это необязательный параметр, который указывает размер блока итераций, который будет распределен между нитями.

Пример использования параметра schedule с типом static:

```
#pragma omp parallel for schedule(static)
```

7.2 Результат выполнения программы

Номер итерации	Значение параметра schedule						
	static	static, 1	static, 2	dynamic	dynamic, 2	guided	guided, 2
1	0	0	0	1	3	1	2
2	0	1	0	2	3	1	2
3	0	2	1	0	1	1	2
4	1	3	1	3	1	0	0
5	1	0	2	1	2	0	0
6	1	1	2	1	2	2	3
7	2	2	3	1	0	2	3
8	2	3	3	2	0	3	1
9	3	0	0	0	1	0	1
10	3	1	0	0	1	0	2

Задание 8. Распараллеливание циклов в OpenMP: программа «Число π»

Напишите OpenMP-программу, которая вычисляет число π с точностью до N знаков после запятой.

$$\pi = (\frac{4}{1+{x_0}^2}+\frac{4}{1+{x_1}^2}+\ldots+\frac{4}{1+{x_{N-1}}^2})\times\frac{1}{N},\ \text{где}\ x_i = (i+0.5)\times\frac{1}{N},\ i = \overline{0, N-1} \tag{8.1}$$

Распределите работу по нитям с помощью OpenMP-директивы for.

8.1 Выполнение работы

```
#include<stdio.h>
#include <omp.h>

int main()
{
   int i, N;
   double x, pi, step, sum = 0.0;

   printf("Enter the precision: ");
   scanf("%d", &N);

   int MAX_N =  N;
   int NUM_THREADS = 4;
   step = 1.0 / (double)MAX_N;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel for reduction(+:sum) private(x)
   for (i = 1; i <= MAX_N; i++) {
       x = (i - 0.5) * step;
       sum = sum + 4.0 / (1.0 + x * x);
   }

   pi = step * sum;
   printf("pi=%.100lf\n", pi);
   return 0;
```

```
25 }
```

8.2 Результат выполнения программы

Входные данные	Выходные данные
1000000000	3.14159265

Задание 9. Распараллеливание циклов в OpenMP: программа «Матрица»

Напишите OpenMP-программу, которая вычисляет произведение двух квадратных матриц $A \times B = C$ размера $n \times n$. Используйте следующую формулу:

$$A = \begin{pmatrix} a_{11} & a_{22} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ . & . & . & \dots & . \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix}, \; B = \begin{pmatrix} b_{11} & b_{22} & b_{13} & \dots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2n} \\ . & . & . & \dots & . \\ b_{n1} & b_{n2} & b_{n3} & \dots & b_{nn} \end{pmatrix} \tag{9.1}$$

$$c_{im} = \sum_{j=1}^{n} a_{ij} \cdot b_{jm}; \; i = 1,2,\dots,n; \; m = 1,2,\dots,n \tag{9.2}$$

$$C = \begin{pmatrix} \sum_{j=1}^{n} a_{1j} \cdot b_{j1} & \sum_{j=1}^{n} a_{1j} \cdot b_{j2} & \sum_{j=1}^{n} a_{1j} \cdot b_{j3} & \dots & \sum_{j=1}^{n} a_{1j} \cdot b_{jn} \\ \sum_{j=1}^{n} a_{2j} \cdot b_{j1} & \sum_{j=1}^{n} a_{2j} \cdot b_{j2} & \sum_{j=1}^{n} a_{2j} \cdot b_{j3} & \dots & \sum_{j=1}^{n} a_{2j} \cdot b_{jn} \\ \dots & \dots & \dots & \dots & \dots \\ \sum_{j=1}^{n} a_{nj} \cdot b_{j1} & \sum_{j=1}^{n} a_{nj} \cdot b_{j2} & \sum_{j=1}^{n} a_{nj} \cdot b_{j3} & \dots & \sum_{j=1}^{n} a_{nj} \cdot b_{n1} \end{pmatrix} \tag{9.3}$$

9.1 Входные данные

Целое число $n, 1 \leqslant n \leqslant 10$, n^2 вещественных элементов матрицы A и n^2 вещественных элементов матрицы B.

9.2 Выходные данные

n^2 вещественных элементов матрицы C.

9.3 Выполнение задания

В этой программе сначала считывается размер матрицы n. Затем считываются две матрицы размера $n \times n$. Затем эти две матрицы умножаются с использованием OpenMP, и результат выводится на экран.

```
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;

vector<vector<double>> readMatrix(int n) {
    vector<vector<double>> matrix(n, vector<double>(n));
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            cin >> matrix[i][j];
        }
    }
    return matrix;
}

void printMatrix(const vector<vector<double>>& matrix) {
    for(const auto& row : matrix) {
        for(double val : row) {
            cout << val << " ";
        }
        cout << "\n";
    }
}

vector<vector<double>> multiplyMatrix(const vector<vector<double
   >>& A,
const vector<vector<double>>& B, int n) {
    vector<vector<double>> C(n, vector<double>(n, 0));

    #pragma omp parallel for
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            for(int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
```

```
            }
        }
        return C;
}

int main() {
    int n;
    cin >> n;

    vector<vector<double>> A = readMatrix(n);
    vector<vector<double>> B = readMatrix(n);

    vector<vector<double>> C = multiplyMatrix(A, B, n);

    printMatrix(C);

    return 0;
}
```

9.4 Результат выполнения программы

Входные данные	Выходные данные
2	14 4
1 3	44 16
4 8	
5 4	
3 0	

Задание 10. Параллельные секции в OpenMP: программа «I'm here!»

Изучите OpenMP-директивы создания параллельных секций sections и section. Напишите программу, содержащую 3 параллельные секции, внутри каждой из которых должно выводиться сообщение: `<Номер нити>]: came in section <Номер секции>`.

Вне секций внутри параллельной области должно выводиться следую- щее сообщение: `[<Номер нити>]: parallel region`.

Запустите приложение на 2-х, 3-х, 4-х нитях. Проследите, как нити распределяются по параллельным секциям.

Входные данные: k — количество нитей в параллельной области. Выходные данные: k-строк вида `«[<Номер нити>]: came in section <Номер секции>»`, k-строк вида `«[<Номер нити>]: parallel region»`.

10.1 Выполнение задания

В этом коде, `#pragma omp parallel sections num_threads(num_threads)` создает параллельные секции, каждая из которых выполняется отдельной нитью. Внутри каждой секции используется `#pragma omp section` для обозначения начала новой секции. Функция `omp_get_thread_num()` используется для получения номера текущей нити.

Затем, `#pragma omp parallel num_threads(num_threads)` создает параллельную область, в которой каждая нить выводит сообщение `"[<Номер нити>]: parallel region"`.

```
#include <omp.h>
#include <stdio.h>

int main() {
    int num_threads = 3; // Замените на желаемое количество нитей

    #pragma omp parallel sections num_threads(num_threads)
    {
```

```
        #pragma omp section
        {
            printf("[%d]: came in section 1\n",
    omp_get_thread_num());
        }

        #pragma omp section
        {
            printf("[%d]: came in section 2\n",
    omp_get_thread_num());
        }

        #pragma omp section
        {
            printf("[%d]: came in section 3\n",
    omp_get_thread_num());
        }
    }

    #pragma omp parallel num_threads(num_threads)
    {
        printf("[%d]: parallel region\n", omp_get_thread_num());
    }

    return 0;
}
```

10.2 Результат выполнения программы для 3, 2 и 4 параллельных секций

```
ivan@W520:~$ g++ -fopenmp 10.cpp -o 10ivan@W520:~$ ./10
[0]: came in section 2
[1]: came in section 1
[2]: came in section 3
[1]: parallel region
[0]: parallel region
[2]: parallel region
ivan@W520:~$ g++ -fopenmp 10.cpp -o 10
ivan@W520:~$ ./10
```

```
[0]: came in section 1
[0]: came in section 3
[1]: came in section 2
[1]: parallel region
[0]: parallel region
ivan@W520:~$ g++ -fopenmp 10.cpp -o 10
ivan@W520:~$ ./10
[1]: came in section 2
[2]: came in section 1
[0]: came in section 3
[1]: parallel region
[0]: parallel region
[3]: parallel region
[2]: parallel region
```

Задание 11. Исследование масштабируемости OpenMP-программ

Проведите серию экспериментов на персональном компьютере по исследованию масштабируемости OpenMP-программ. Заполните следующие таблицы.

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

11.1 Программа «Матрица»

11.1.1 Модификация программы «Матрица»

Добавим функцию генерации случайных значений матрицы, функцию подсчета времени выполнения и функцию вызова аргументов: размер матрицы, число тредов, путь файла, в который будет записываться результат.

```
#include <iostream>
#include <vector>
#include <omp.h>
#include <chrono>
#include <fstream>
#include <random>
using namespace std;
using namespace chrono;

// Function to generate a random matrix of size n x n
vector<vector<double>> generateRandomMatrix(int n) {
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<double> dis(1.0, 10.0);

    vector<vector<double>> matrix(n, vector<double>(n, 0.0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = dis(gen);
```

```
        }
    }

    return matrix;
}

void printMatrix(const vector<vector<double>>& matrix) {
    for(const auto& row : matrix) {
        for(double val : row) {
            cout << val << " ";
        }
        cout << "\n";
    }
}

vector<vector<double>> multiplyMatrix(const vector<vector<double
   >>& A,
const vector<vector<double>>& B, int n) {
    vector<vector<double>> C(n, vector<double>(n, 0));

    #pragma omp parallel for
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            for(int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C;
}

int main(int argc, char *argv[]) {
    if (argc != 4) {
        cerr << "Usage: " << argv[0] << " <Matrix_Size> <
   Num_Threads> <Runtime_Output>" << endl;
        return 1;
    }

    int n = stoi(argv[1]);
    int num_threads = stoi(argv[2]);
    string runtime_output = argv[3];

    // Generate random matrices
```

```
    vector<vector<double>> A = generateRandomMatrix(n);
    vector<vector<double>> B = generateRandomMatrix(n);

    // Set the number of threads
    omp_set_num_threads(num_threads);

    // Measure the runtime
    auto start_time = high_resolution_clock::now();
    vector<vector<double>> C = multiplyMatrix(A, B, n);
    auto end_time = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end_time -
start_time);

    // Print the runtime to the specified output file
    ofstream runtime_file(runtime_output);
    runtime_file << duration.count() / 1000.0 << " seconds" <<
endl;
    runtime_file.close();

    return 0;
}
```

Оценить время умножения матрицы размером 10 000x10 000 с использованием одного потока на основе времени, затраченного на матрицу размером 100x100, можно пропорционально. Если время умножения матрицы размером 100x100 составляет 0.022 секунды, мы можем использовать эту информацию для оценки времени умножения матрицы размером 10 000x10 000.

Давайте предположим, что сложность по времени приблизительно кубическая ($O(n^3)$), и мы будем использовать линейную пропорцию для оценки времени — умножение матрицы 10000x10000 займет примерно 6 часов.

Number	Programm	Parameter N	Number of threads	Runtime (sec.)
1	Matrix	100	1	0.022
2	Matrix	10,000	1	∼22000
3	Matrix	100	2	0.012
4	Matrix	10,000	2	∼12000
5	Matrix	100	4	0.006
6	Matrix	10,000	4	∼6000
7	Matrix	100	6	0.007
8	Matrix	10,000	6	∼7000
9	Matrix	100	8	0.006
10	Matrix	10,000	8	∼6000
11	Matrix	100	10	0.006
12	Matrix	10,000	10	∼6000
13	Matrix	100	12	0.007
14	Matrix	10,000	12	∼7000

11.1.2 Результат для программы «Матрица»

11.1.3 Вычисление ускорения

1) Минимальная конфигурация: $k = 1$ 2) Время для минимальной конфигурации (T_k): $T_1 = 0.022$ сек 3) Вычисление времени на минимальной конфигурации: $k \times T_k = T_1 = 0.022$ сек 4) Ускорение:

- Ускорение для 2 ядер: $\frac{0.022}{0.012} \approx 1.83$
- Ускорение для 4 ядер: $\frac{0.022}{0.006} \approx 3.67$
- Ускорение для 6 ядер: $\frac{0.022}{0.007} \approx 3.14$
- Ускорение для 8 ядер: $\frac{0.022}{0.006} \approx 3.67$
- Ускорение для 10 ядер: $\frac{0.022}{0.006} \approx 3.67$
- Ускорение для 12 ядер: $\frac{0.022}{0.007} \approx 3.14$

11.2 Программа «Число пи»

11.2.1 Модификация программы «Число пи»

Добавим функцию ввода количества тредов.

```
#include <stdio.h>
#include <omp.h>

int main()
{
    int i, N, NUM_THREADS;
    double x, pi, step, sum = 0.0;

    printf("Enter the precision: ");
    scanf("%d", &N);

    printf("Enter the number of threads: ");
    scanf("%d", &NUM_THREADS);

    int MAX_N = N;
    step = 1.0 / (double)MAX_N;
    omp_set_num_threads(NUM_THREADS);

    double start_time = omp_get_wtime(); // Record start
time

    #pragma omp parallel for reduction(+:sum) private(x)
    for (i = 1; i <= MAX_N; i++)
    {
        x = (i - 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }

    double end_time = omp_get_wtime(); // Record end time

    pi = step * sum;
    printf("pi=%.100lf\n", pi);
    printf("Runtime: %lf seconds\n", end_time - start_time);
    printf("Number of threads: %d\n", NUM_THREADS);
```

```
35          return 0;
    }
```

11.2.2 Результат для программы «Пи»

#	Program	Parameter N	Number of Threads	Runtime (sec.)
1	Number pi	100	1	0.000022
2		10,000,000	1	0.098166
3		400,000,000	1	2.505343
4		100	2	0.000269
5		10,000,000	2	0.039858
6		400,000,000	2	1.27925
7		100	4	0.000245
8		10,000,000	4	0.021424
9		400,000,000	4	0.654643
10		100	6	0.000281
11		10,000,000	6	0.027002
12		400,000,000	6	0.69255
13		100	8	0.003926
14		10,000,000	8	0.018002
15		400,000,000	8	0.655944
16		100	10	0.000716
17		10,000,000	10	0.022722
18		400,000,000	10	0.669349
19		100	12	0.000829
20		10,000,000	12	0.018299
21		400,000,000	12	0.674665

Давайте вычислим ускорение для заданных значений времени выполнения программы на различном количестве ядер.

По формуле ускорения:

$$\text{Ускорение} = \frac{T_1}{T_N}$$

Где: - T_1 - время выполнения на минимальной конфигурации (одном ядре). - T_N - время выполнения на конфигурации с N ядрами.

Теперь посчитаем ускорения для различных конфигураций:

$$\text{Ускорение для 2 ядер} = \frac{0.000022}{0.000269} \approx 0.0817$$
$$\text{Ускорение для 4 ядер} = \frac{0.000022}{0.000245} \approx 0.0898$$
$$\text{Ускорение для 6 ядер} = \frac{0.000022}{0.000281} \approx 0.0783$$
$$\text{Ускорение для 8 ядер} = \frac{0.000022}{0.003926} \approx 0.0056$$
$$\text{Ускорение для 10 ядер} = \frac{0.000022}{0.000716} \approx 0.0307$$
$$\text{Ускорение для 12 ядер} = \frac{0.000022}{0.000829} \approx 0.0265$$

Значения ускорений показывают, насколько быстрее программа выполняется на конфигурации с N ядрами по сравнению с минимальной конфигурацией (1 ядро).