МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

**«Вятский государственный университет»**

**(ФГБОУ ВО «ВятГУ»)**

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

# Параллельное программирование

Многопоточная реализация вычислительно сложного алгоритма с применением библиотеки OpenMP

Вариант 7

Выполнил студент группы ИВТ-31 _____/Кудяшев Я.Ю./

Проверил преподаватель_____/Долженкова М.Л./

Киров 2022

## 1. Задание

Познакомиться со стандартом OpenMP, получить навыки реализации многопоточных SPMD-приложений с применением OpenMP.

Этапы работы:

1) Изучить основные принципы создания приложений с использованием библиотеки OpenMP, рассмотреть базовый набор директив компилятора
2) Выделить в полученной в ходе выполнения первой лабораторной работы реализации алгоритма фрагменты кода, выполнение которых может быть разнесено на несколько процессорных ядер
3) Реализовать многопоточную версию алгоритма с помощью языка C++ и библиотеки OpenMP, используя при этом необходимые примитивы синхронизации
4) Показать корректность полученной реализации путём осуществления на построенном в ходе первой лабораторной работы наборе тестов
5) Провести доказательную оценку эффективности OpenMP-реализации алгоритма

## 2. Метод распараллеливания алгоритма

В качестве областей участков для распараллеливания при помощи OpenMP были выбраны те же участки, что и при простом распараллеливании. Это было сделано для наиболее точного сравнения результатов тестов.

Из исследований алгоритма для перемножения полиномов с помощью быстрого преобразованья Фурье удалось выяснить, что время в большей степени зависит от количества входных векторов, нежели от размерности. Было принято решение переложить работу по умножению каждой пары векторов на потоки.

Таким образом, после перемножения пары векторов, каждое последующее умножение будет происходить уже с полученным в результате предыдущего умножения вектором. В случае, когда количество входных векторов равно 2, параллельно будет выполняться ДПФ для каждого входного вектора.

## 3. Программная реализация

Листинг программной OpenMP-реализации алгоритма приведен в приложении А.

## 4. Тестирование

Тестирование проводилось на ЭВМ под управлением 64-разрядной ОС Windows 10, с 8 ГБ оперативной памяти, с процессором Intel Core i5-8250U с частотой 1.80 ГГц (8 логических и 4 физических ядра).

Результаты тестирования и сравнения с параллельной и последовательной реализациями приведены в таблице 1.

| Исходные данные (кол-во полиномов, размерность) | Последовательная реализация, мс | Параллельная реализация, мс | OpenMP-реализация, мс | Ускорение (параллельная раелизация) | Ускорение (последовательная реализация) |
|---|---|---|---|---|---|
| 2, 100000 | 210 мс | 184 мс | 208 | 0,88 | 1,01 |
| 4, 100000 | 1183 мс | 581 мс | 590 | 0,98 | 2,01 |
| 6, 100000 | 5485 мс | 1046 мс | 1217 | 0,86 | 4,51 |
| 8, 100000 | 24990 мс | 4133 мс | 4308 | 0,96 | 5,80 |
| 2, 1000000 | 1493 мс | 1033 мс | 1175 | 0,88 | 1,27 |
| 4, 1000000 | 10734 мс | 4008 мс | 4101 | 0,98 | 2,62 |
| 6, 1000000 | 50241 мс | 9558 мс | 10171 | 0,94 | 4,94 |
| 8, 1000000 | 259036 мс | 40712 мс | 40150 | 1,01 | 6,45 |
| 2, 10000000 | 26759 мс | 19438 мс | 21103 | 0,92 | 1,27 |
| 8, 10000 | 2640 мс | 419 мс | 502 | 0,83 | 5,26 |
| | | | Среднее | 0,92 | 3,51 |
| | | | Максимальное | 1,01 | 6,45 |
| | | | Минимальное | 0,83 | 1,01 |

Таблица 1 – Результаты тестирования

Исходя из результатов тестирования можно сказать, что в реализации данного алгоритма OpenMP не дает существенного ускорения в скорости при сравнении с параллельной реализацией. Наоборот, большинство тестов имеют задержку, связанную с тем, что на создание потоков тратится больше времени. Значительное ускорение, что не удивительно, присутствует при сравнении с последовательной реализацией алгоритма.

5. Вывод

В ходе выполнения лабораторной работы была реализована OpenMP-версия алгоритма перемножения полиномов с помощью быстрого преобразованья Фурье на языке C++. OpenMP-версия алгоритма оказалась менее эффективной, чем версия с простым распараллеливанием процессов. Это связано со структурой алгоритма, т.к. OpenMP в значительной части оптимизирован вокруг модели однопоточного основного процесса и работы в массивных циклах, что не совсем подходит под данный алгоритм.

# Приложение А

(обязательное)

## Листинг программы

```
#include <iostream>
#include <omp.h>
#include <vector>
#include <complex>
#include <chrono>
#include <fstream>

using namespace std;

typedef complex<double> base;

vector<int> information[30];      //data vector
vector<int> result(10000000);     //result vector
int counter = 1;
int thread_counter = 8;
//std::thread threads[8];  //8 threads were created


int rev(int num, int lg_n) {            //begining of good realisation
	int res = 0;
	for (int i = 0; i < lg_n; ++i)
		if (num & (1 << i))
			res |= 1 << (lg_n - 1 - i);
	return res;
}

void good_realisation(vector<base>& a, bool invert) { //БПФ и обратное БПФ
	int n = (int)a.size();
	int lg_n = 0;
	while ((1 << lg_n) < n)  ++lg_n;

	for (int i = 0; i < n; ++i)
		if (i < rev(i, lg_n))
			swap(a[i], a[rev(i, lg_n)]);

	for (int len = 2; len <= n; len <<= 1) {
		double ang = 2 * 3.14 / len * (invert ? -1 : 1);
		base wlen(cos(ang), sin(ang));
		for (int i = 0; i < n; i += len) {
			base w(1);
			for (int j = 0; j < len / 2; ++j) {
				base u = a[i + j], v = a[i + j + len / 2] * w;
				a[i + j] = u + v;
				a[i + j + len / 2] = u - v;
				w *= wlen;
			}
		}
	}
	if (invert)
		for (int i = 0; i < n; ++i)
			a[i] /= n;
}

void good_multiplication(const vector<int>& a, const vector<int>& b, vector<int>& res,
int number) {       //multiplication of two vectors
	vector<base> fa(a.begin(), a.end()), fb(b.begin(), b.end());
	int n = 1;
	while (n < max(a.size(), b.size()))  n <<= 1;
```

```
	n <<= 1;
	fa.resize(n), fb.resize(n);
	if (number == 2) {
#pragma omp parallel sections
		{
	#pragma omp section
			{
		good_realisation(fa, false);
		}
	#pragma omp section
			{

	good_realisation(fb, false);
	}
		}
	}
	else {
		good_realisation(fa, false);

		good_realisation(fb, false);
	}

	for (int i = 0; i < n; ++i)
		fa[i] *= fb[i];
	good_realisation(fa, true);

	res.resize(n);
	for (int i = 0; i < n; ++i)
		res[i] = int(fa[i].real() + 0.5);
}

void fill_from_file(string path, int number_of_vectors, int size_of_vectors) {
	//reading data from the file
	ifstream vectorr("C:\\Programming\\Parallel programming\\Lab 1\\" + path);

	for (int i = 0; i < number_of_vectors; i++) {
		information[i].resize(size_of_vectors);
	}

	for (int i = 0; i < number_of_vectors; i++) {
		for (int j = 0; j < size_of_vectors; j++) {
			vectorr >> information[i].at(j);
		}
	}
	vectorr.close();
}

void enter(string first, int number, int size, string path) {		//input

	std::cout << "\n" + first + " test is running\n";
	fill_from_file(path, number, size);	//number, size

	std::cout << "Good algorithm: ";
	unsigned int start_time = clock();


	switch (number) {
	case 2:
		good_multiplication(information[0], information[1], information[0], 2);

		break;

	case 4:
#pragma omp parallel for
		for(int i=0;i<2;i++)
```

```
                        good_multiplication(information[i*2], information[i*2+1],
information[i*2], 1);

                good_multiplication(information[0], information[2], information[0], 2);

                break;

        case 6:
#pragma omp parallel for
                for (int i = 0; i < 2; i++)
                        good_multiplication(information[i * 2], information[i * 2 + 1],
information[i * 2], 1);

#pragma omp parallel sections
                {
#pragma omp section
                        {
                                good_multiplication(information[0], information[2],
information[0], 1);
                        }
#pragma omp section
                        {
                                good_multiplication(information[4], information[5],
information[4], 1);
                        }
                }

                good_multiplication(information[0], information[4], information[0], 2);

                break;

        case 8:
#pragma omp parallel for
                for (int i = 0; i < 4; i++)
                        good_multiplication(information[i * 2], information[i * 2 + 1],
information[i * 2], 1);

#pragma omp parallel sections
                {
#pragma omp section
                        {
                                good_multiplication(information[0], information[2],
information[0], 1);
                        }
#pragma omp section
                        {
                                good_multiplication(information[4], information[6],
information[4], 1);
                        }
                }

                good_multiplication(information[0], information[4], information[0], 2);

                break;

        default:
                std::cout << "Wrong number of vectors";
        }


        unsigned int end_time = clock();
        unsigned int search_time = end_time - start_time;
        std::cout << search_time << " mc\n";
        cout << '\n';
        //counter = 1;
```

```
}

int main()
{
	cout << "2 vectors of size 100000";
	enter("The first", 2, 100000, "int_0-100 2_100000.txt");

	cout << "4 vectors of size 100000";
	enter("The second", 4, 100000, "int_0-100 4_100000.txt");

	cout << "8 vectors of size 100000";
	enter("The third", 8, 100000, "int_0-100 8_100000.txt");

	cout << "2 vectors of size 1000000";
	enter("The fourth", 2, 1000000, "int_0-100 2_1000000.txt");

	cout << "4 vectors of size 1000000";
	enter("The fifth", 4, 1000000, "int_0-100 4_1000000.txt");

	cout << "8 vectors of size 1000000";
	enter("The sixth", 8, 1000000, "int_0-100 8_1000000.txt");

	cout << "6 vectors of size 100000";
	enter("The seventh", 6, 100000, "int_0-100 6_100000.txt");

	cout << "6 vectors of size 1000000";
	enter("The eighth", 6, 1000000, "int_0-100 6_1000000.txt");

	cout << "8 vectors of size 10000";
	enter("The nineth", 8, 10000, "int_0-100 8_10000.txt");

	cout << "2 vectors of size 10000000";
	enter("The tenth", 2, 10000000, "int_0-100 2_10000000.txt");

}
```