МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)

Институт информатики и кибернетики
Кафедра технической кибернетики

ЛАБОРАТОРНАЯ РАБОТА № 3

по курсу
Параллельное программирование

Группа 6409
Студент Д.К. Чернышова
(*подпись*)

Преподаватель,
к.ф.-м.н. Е.С. Козлова
(*подпись*)

Самара 2023

ЗАДАНИЕ

Произвести запуск программ для поиска суммы векторов, которые используют технологии MPI и OpenMP, на различном количестве процессоров (потоков).

В ходе анализа работы программы оценить время ее выполнения на различном количестве исполняющих нитей (процессов). Оценить влияние различных функций и директив(опций) на скорость работы приложений.

Для дальнейшего анализа технологий искусственно увеличить соотношение вычислений и операций по обеспечению параллелизма путем повторения функции сложения векторов Q раз. В ходе анализа работы программ также оценить время их выполнения на различном количестве исполняющих нитей (процессов). Оценить влияние различных функций и директив на скорость работы приложений.

Реализовать последовательный вариант программы. Оценить время ее выполнения для обычного и "усложненного" вариантов. Рассчитать ускорение параллельных программ относительно последовательного варианта.

Таблица 1 – Исходные данные на ЛР № 3

Тип	int
N	4 500 000(+11)
Количество процессов/потоков	5, 10, 15
Q	18

ВВЕДЕНИЕ

Сейчас почти невозможно найти современную компьютерную систему без многоядерного процессора. Идея многоядерных систем проста: это относительно эффективная технология для масштабирования потенциальной производительности процессора. Эта технология стала широко доступной около двадцати лет назад, и теперь каждый современный разработчик способен создать приложение с параллельным выполнением для использования такой системы. Сложность параллельного программирования часто недооценивается [1].

ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ

2.1 Результаты работы программ без параметра Q

В ходе исследования времени работы программ здесь и далее проводилось усреднение не менее чем по 12 запускам.

На рисунках 1-3 представлены скрины запуска и работы программ без параметра Q.

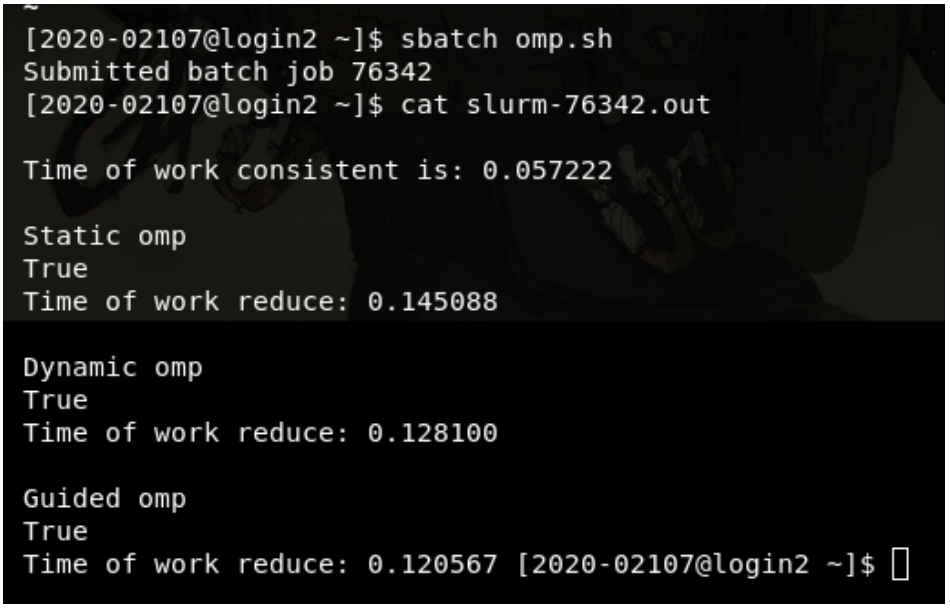

Рисунок 1 – Пример работы программы OpenMP на 15 процессах для static, dynamic, guided

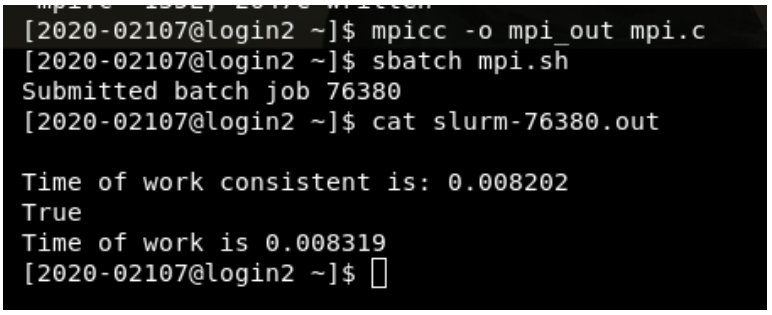

Рисунок 2 – Пример работы программы MPI на 5 процессах для кратных размерностей

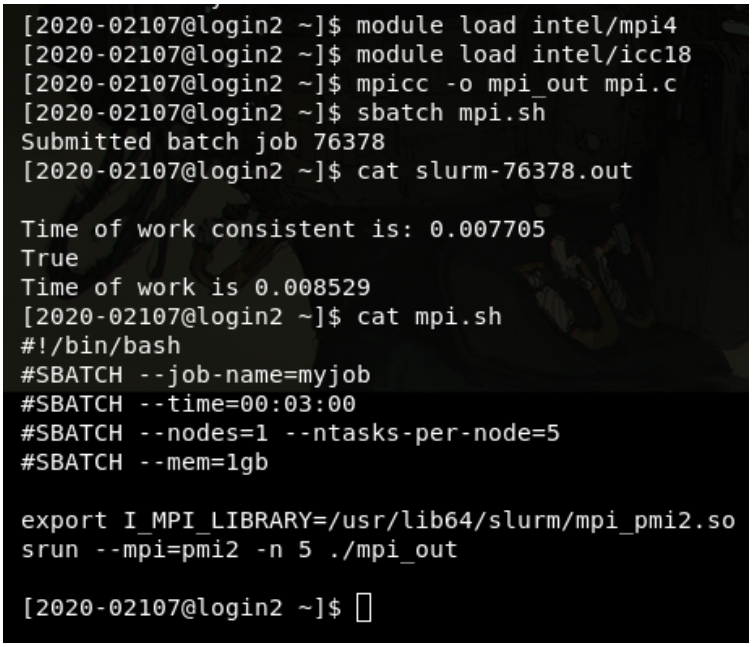

Рисунок 3 – Пример работы программы MPI на 5 процессах для некратных размерностей

Последовательная программа представляла собой программу для подсчёта суммы двух векторов одним процессом/потоком. Время работы последовательной программы составило 0,03 с.

В таблицах 3-4 представлено время выполнения параллельных программ и их ускорение по сравнению с последовательным вариантом.

Таблица 3 – Время работы параллельных программ без параметра Q

Кол-во процессов, шт	Время для OpenMP, с			Время для MPI, с	
	static	dynamic	guided	кратное	некратное
5	0,013005	0,014447	0,007852	0,008780	0,008319
10	0,011146	0,014716	0,006964	0,008854	0,008534
15	0,010959	0,012558	0,006415	0,009743	0,007453

Таблица 4 – Ускорение параллельных программ без параметра Q

Кол-во процессов, шт	Ускорение для OpenMP			Ускорение для MPI	
	static	dynamic	guided	кратное	некратное
5	2,306805	2,076555	3,82068	3,416856	3,606202
10	2,691548	2,038597	4,40140	3,38829	3,515350
15	2,737476	2,388915	4,62463	3,07913	4,048582

На рисунке 6 приведен график зависимости времени работы программ от количества процессов. На рисунке 7 приведен график зависимости ускорения программ от количества процессов.

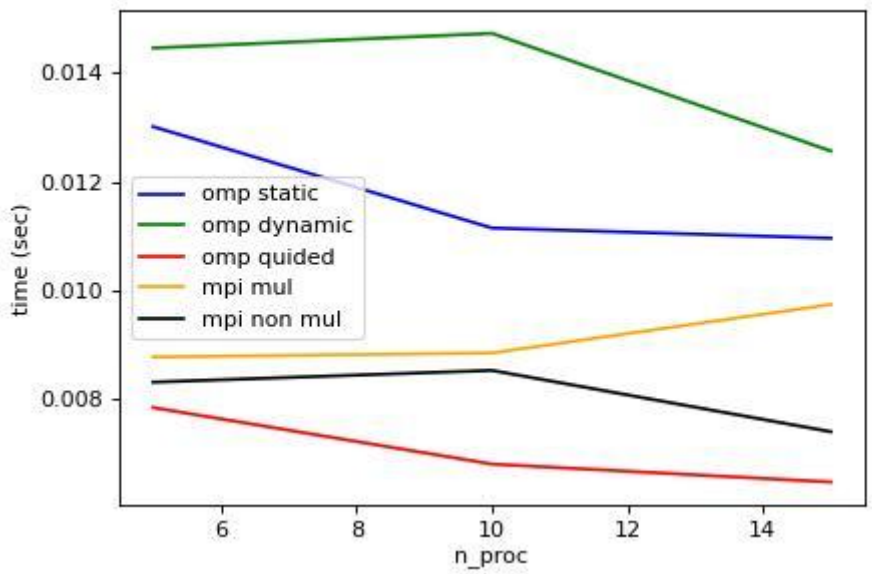

Рисунок 4 – Время работы программ без параметра Q, сек

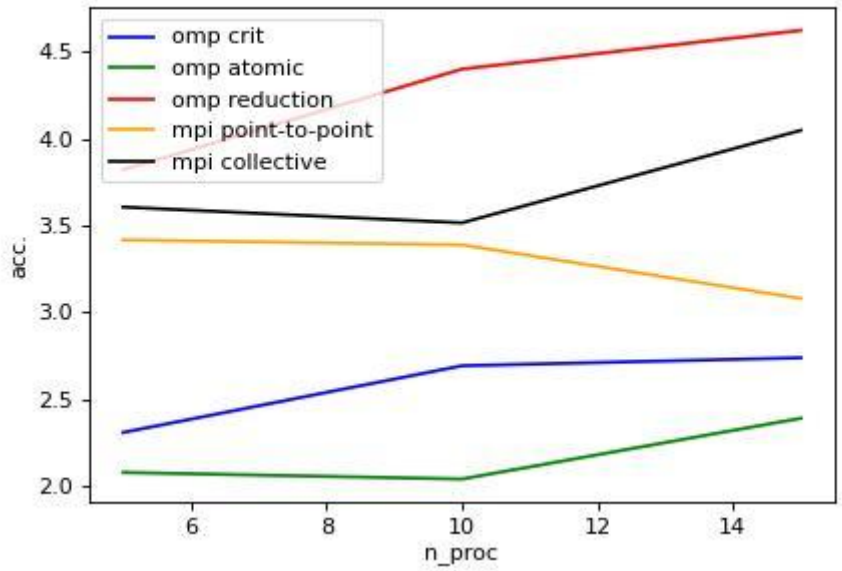

Рисунок 5 – Ускорение программ без параметра Q

2.3 Результаты работы программ с параметром Q

На рисунках 6-8 представлены скрины запуска и работы программ с параметром Q.

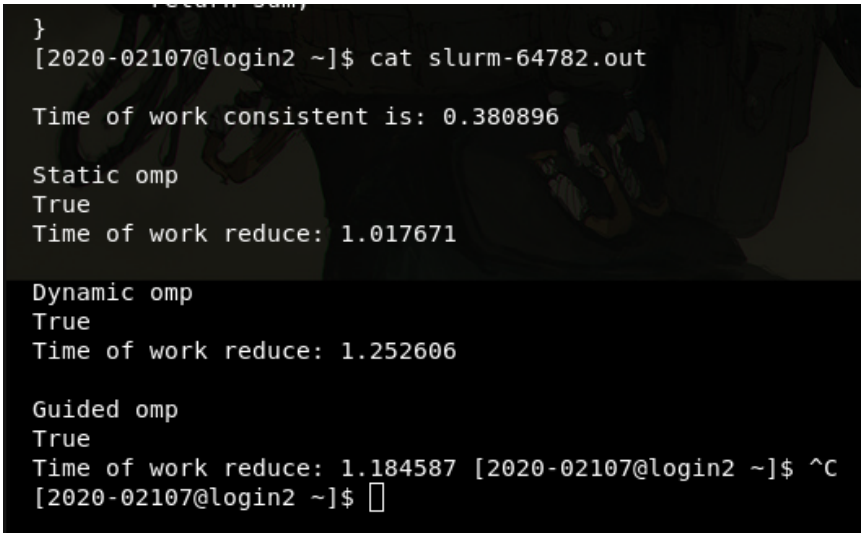

Рисунок 6 – Пример работы программы OpenMP на 15 процессах для static, dynamic, guided

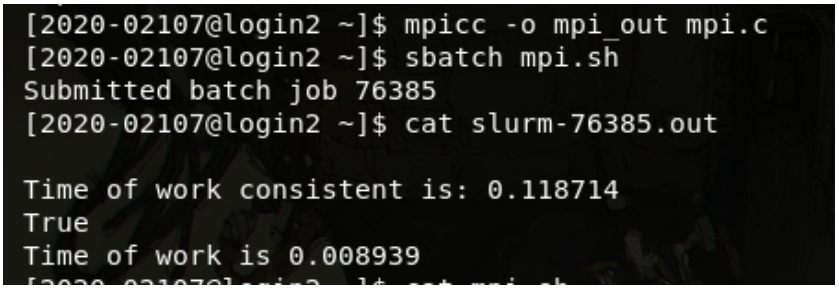

Рисунок 7 – Пример работы программы MPI на 5 процессах для кратных

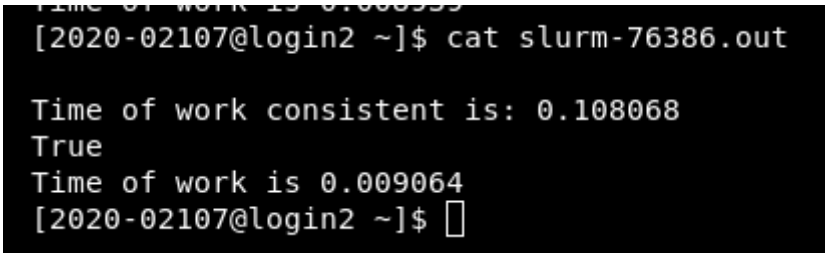

Рисунок 8 – Пример работы программы MPI на 5 процессах для некратных размерностей

Усложненная последовательная программа представляла собой программу для подсчёта суммы двух векторов одним процессом/потоком с параметром пересчёта Q в цикле. Время работы последовательной программы составило 5,89 с.

В таблицах 5-6 представлено время выполнения параллельных программ и их ускорение по сравнению с последовательным вариантом.

Таблица 5 – Время работы параллельных программ с параметром Q

Кол-во процессов, шт	Время для OpenMP, с			Время для MPI, с	
	static	dynamic	guided	кратное	некратное
5	0,090479	0,087457	0,08415	0,09764	0,10852
10	0,048601	0,049339	0,04359	0,09775	0,10764
15	0,053674	0,044055	0,03818	0,11074	0,10744

Таблица 6 – Ускорение параллельных программ с параметром Q

Кол-во процессов, шт	Ускорение для OpenMP			Ускорение для MPI	
	static	dynamic	guided	кратное	некратное
5	42,99340	44,4790	46,22697	39,8402	33,0669
10	80,0395	78,8518	89,2406	39,7933	28,23853
15	72,47456	88,2666	101,8858	35,1263	22,7827

На рисунке 12 приведен график зависимости времени работы программ от количества процессов. На рисунке 13 приведен график зависимости ускорения программ от количества процессов.

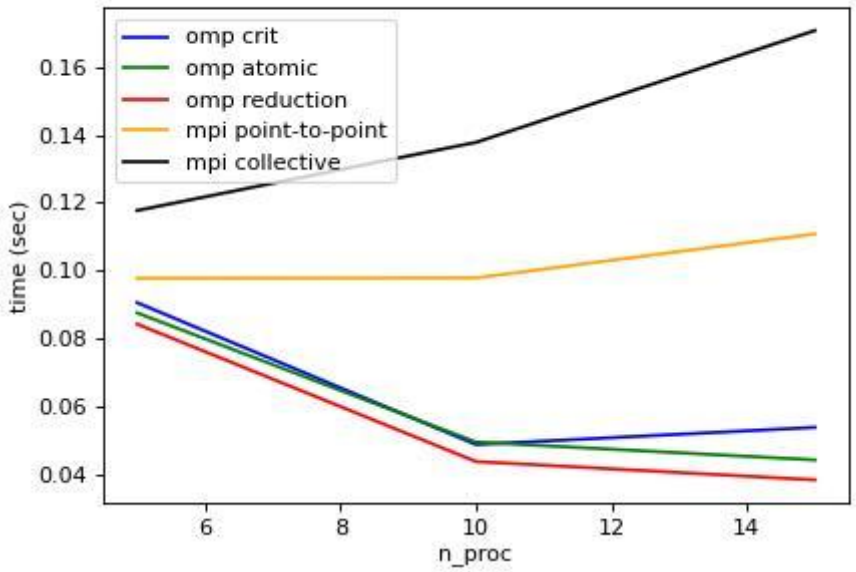

Рисунок 9 – Время работы программ с параметром Q, сек

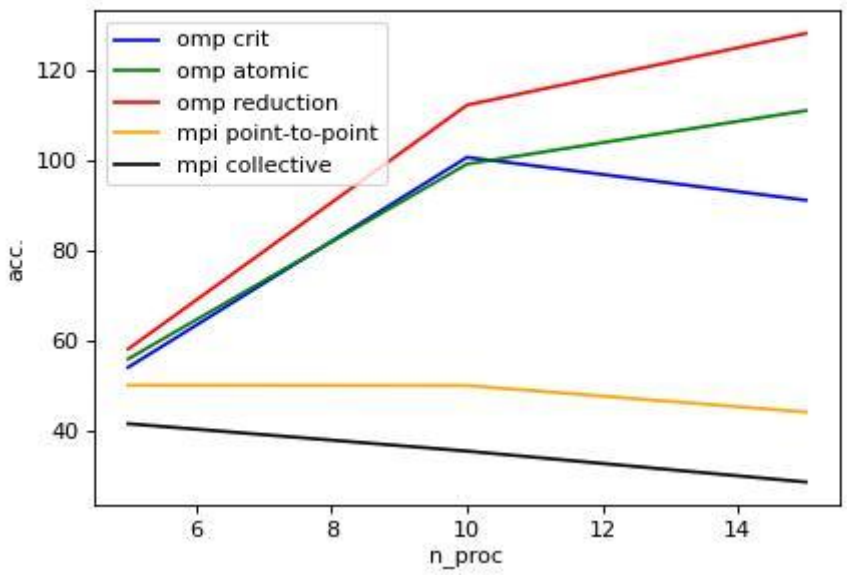

Рисунок 10 – Ускорение программ с параметром Q

ВЫВОДЫ:

Из полученных результатов видно, что:

1. Программы OpenMP и MPI показали с параметром Q результаты лучше, чем последовательные программы. Без Q параллельная программа показала результаты хуже. Варианты распараллеливания OpenMP распределяли вычисления между потоками, каждый поток обрабатывал порцию итераций, в силу параллельной работы потоков это приводит к ускорению программы. Варианты распараллеливания MPI также распределяли части вектора между процессорами, таким образом каждый процесс производил расчёт для частей векторов.
2. Максимальное ускорение для технологии OpenMP без параметра Q было получено с использованием guided, с параметром Q с использованием также guided. Для технологии MPI максимальное ускорение было получено для кратного количества элементов вектора без параметра Q и с параметром Q.
3. Увеличение числа процессов для обеих технологий приводило к уменьшению времени работы с параметром Q и без Q, но излишний параллелизм может привести к замедлению программы.

4. С ростом количества процессов растет и ускорение программ в случае использования параметра Q, в случае работы без параметра Q ускорение падает. С увеличением Q растет разрыв между временем последовательной и параллельной программой. Учет параметра Q означает Q-кратное увеличение объема вычислений на данном потоке/процессе.

ЗАКЛЮЧЕНИЕ

Цель лабораторной работы – написать параллельные программы суммы двух векторов с использованием технологий MPI и OpenMP и сравнить время выполнения с длительностью последовательной программы достигнута. Показано, что использование параллельных технологий для данного типа программ обосновано/не требуется, ввиду того, что подсчёт частичных сумм, где каждый процесс/поток работает с частями двух векторов, выполняется параллельно. Лучше всего себя показала технология OpenMP с использованием guided, но темпы роста ускорения больше у кратного MPI. Это можно объяснить тем, что создание новых нитей требует большего количества ресурсов, чем создание сети MPI для использования процессов.

В ходе выполнения лабораторной работы я изучил(а) основы MPI и OpenMP, приобрел(а) навыки по написанию параллельных программ с использованием вышеперечисленных технологий. Наиболее сложной частью выполнения лабораторной работы было составление отчета. Интерес вызвало изучение новых директив MPI и OpenMP.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Денисенко, А.А. Параллельное программирование [Электронный ресурс] / А. А. Денисенко. — Краснодар. 2019. — 38 с. // VIII международная научная конференция «Технические науки в России и за рубежом» URL: https://moluch.ru/conf/tech/archive/332/pdf/ (дата обращения: 20.09.2022).

2 Козлова, Е.С. Лабораторные работы по курсу «Параллельное программирование»: Методические указания [Текст] / Сост. Е.С. Козлова, А.С. Широканев − Самара, 2019. – 61 с.

3 Воеводин, В. В. Параллельные вычисления [Текст] / В. В. Воеводин, Вл. В. Воеводин. — СПб.: БХВ-Петербург, 2002. — 608 с.

4 Богачёв К.Ю. Основы параллельного программирования: учебное пособие, 2-е изд. [Текст] / К. Ю. Богачёв - М. : БИНОМ. Лаборатория знаний, 2013. - 344 с.

5 Гергель, В. П. Теория и практика параллельных вычислений, 2-е изд. [Текст] / В. П. Гергель. — М.: Интуит. 2016. - 500 с.

6 Боресков А.В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA Учеб. пособие [Текст] / А.В. Боресков - М.: Издательство Московского университета, 2012. - 336 с.

7 Библиографическое описание документа. Общие требования и правила составления [Электронный ресурс] / сост.: В.С. Крылова, С.М. Григорьевская, Е.Ю. Кичигина // Официальный интернет-сайт научной библиотеки Томского государственного университета. – Электрон. дан. – Томск, [2010]. – http://www.lib.tsu.ru/win/produkzija/metodichka/metodich.html (дата обращения: 10.09.2019).

ПРИЛОЖЕНИЕ А

Код программы с технологией MPI

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>


void sum(int* a, int* b, int* res, int n){
	int i,j;
	for ( i = 0; i < n; ++i){
		for ( j = 0; j < 5; ++j){
			res[i] = a[i] + b[i];
		}
	}
}

int* consistent(int* vec1, int* vec2, int n){
	int* sum = (int*)calloc(sizeof(int), n);
	int i, j;
	for(i = 0; i < n; ++i){
		for (j = 0; j < 5; ++j){
			sum[i] = vec1[i] + vec2[i];
		}
	}
	return sum;
}

void test(int* vec1, int* vec2, int n){
	bool flag = true;
	for (int i = 0; i < n && flag; ++i){
		flag = vec1[i] == vec1[i];
	}

	if (flag){
		printf("\nTrue");
	}
	else{
		printf("\nFalse");
	}
}

int main(int argc, char* argv[])
{
	srand(time(0));

	int p, nu;
	int N = 4500 * 1000;
	int* a = NULL;
	int* b = NULL;
	int* c = NULL;
	int* real_sum = NULL;

	int* sendCounts = NULL;
	int* displs = NULL;

	MPI_Init(&argc, &argv);
```

```
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        MPI_Comm_rank(MPI_COMM_WORLD, &nu);

        if (nu == 0)
        {
                a = (int*)calloc(sizeof(int), N);
                b = (int*)calloc(sizeof(int), N);
                c = (int*)calloc(sizeof(int), N);
                real_sum = (int*)calloc(sizeof(int), N);

                for (int i = 0; i < N; ++i){
                        a[i] = 100 + 10 * rand() % 100;
                        b[i] = 100 + 10 * rand() % 100;
                        real_sum[i] = a[i] + b[i];
                }
        }

        double st_time;
        MPI_Status status;
        int offset = N / p;

        sendCounts = (int*)calloc(sizeof(int), p);
        displs = (int*)calloc(sizeof(int), p);

        for (int i = 0; i < p; ++i){
                sendCounts[i] = offset;
        }
        for (int i = 0; i < N % p; ++i){
                sendCounts[i] += 1;
        }
        displs[0] = 0;
        for (int i = 1; i < p; ++i){
                displs[i] = displs[i-1] + sendCounts[i-1];
        }
        int* a_loc = (int*)calloc(sizeof(int), sendCounts[nu]);
        int* b_loc = (int*)calloc(sizeof(int), sendCounts[nu]);
        int* c_loc = (int*)calloc(sizeof(int), sendCounts[nu]);

        MPI_Scatterv(a, sendCounts, displs, MPI_INT, a_loc, sendCounts[nu], MPI_INT,
0, MPI_COMM_WORLD);
        MPI_Scatterv(b, sendCounts, displs, MPI_INT, b_loc, sendCounts[nu], MPI_INT,
0, MPI_COMM_WORLD);


        if (nu == 0) {
                st_time = MPI_Wtime();
                real_sum = consistent(a, b, N);
                double total_time = MPI_Wtime() - st_time;
                printf("\nTime of work consistent is: %f", total_time);

                free(a);
                free(b);
                st_time = MPI_Wtime();
        }

        sum(a_loc, b_loc, c_loc, sendCounts[nu]);

        MPI_Gatherv(c_loc, sendCounts[nu], MPI_INT, c, sendCounts, displs, MPI_INT,
0, MPI_COMM_WORLD);

        if (nu == 0)
```

```
        {
                test(c, real_sum, N);
                printf("\nTime of work is %f\n", MPI_Wtime() - st_time);
                free(real_sum);
                free(c);

        }

        free(a_loc);
        free(b_loc);
        free(c_loc);
        free(sendCounts);
        free(displs);

        MPI_Finalize();

        return 0;
}
```

ПРИЛОЖЕНИЕ Б

Код программы с технологией OpenMP

```
#include <omp.h>
#include "stdio.h"
#include <stdbool.h>
#define THREAD_COUNT 5
#define Q 18

int* consistent(int* vec1, int* vec2, int n){
    int* sum = (int*)calloc(sizeof(int), n);
    int i, j;
    for(i = 0; i < n; ++i){
        for(j = 0; j < Q; ++j){
               sum[i] = vec1[i] + vec2[i];
        }
    }
    return sum;
}
int* omp_static(int* vec1, int* vec2, int n){
    int* sum = (int*)calloc(sizeof(int), n);
    int i, j;
    #pragma omp parallel for private(i,j) shared(vec1, vec2, sum) schedule(static,
100)
    for(i = 0; i < n; ++i){
        for(j = 0; j < Q; ++j){
               sum[i] = vec1[i] + vec2[i];
        }
    }
    return sum;
}
int* omp_dynamic(int* vec1, int* vec2, int n){
    int* sum = (int*)calloc(sizeof(int), n);
    int i, j;
    #pragma omp parallel for private(i,j) shared(vec1, vec2, sum) schedule(dynamic,
100)
    for(i = 0; i < n; ++i){
        for(j = 0; j < Q; ++j){
               sum[i] = vec1[i] + vec2[i];
        }
    }
    return sum;
}
int* omp_guided(int* vec1, int* vec2, int n){
    int* sum = (int*)calloc(sizeof(int), n);
    int i, j;
    #pragma omp parallel for private(i,j) shared(vec1, vec2, sum) schedule(guided,
100)
    for(i = 0; i < n; ++i){
        for(j = 0; j < Q; ++j){
               sum[i] = vec1[i] + vec2[i];
        }
    }
    return sum;
}
void test(int* vec1, int* vec2, int n){
    bool flag = true;
    int i;
    for (i = 0; i < n && flag; ++i){
```

```
        flag = vec1[i] == vec1[i];
    }
    if (flag){
        printf("\nTrue");
    }
    else{
        printf("\nFalse");
    }
}
int main(int argc, char* argv[]) {
    omp_set_num_threads(THREAD_COUNT);
    srand(time(0));
    int n = 4500 * 1000;
    double st_time, total_time;

    int* a = (int*)calloc(sizeof(int), n);
    int* b = (int*)calloc(sizeof(int), n);
    int* sum_vec = NULL;
    int* referenced_vec = NULL;

    int i;
    for (i = 0; i < n; ++i){
        a[i] = i;
        b[i] = n - i;
    }

    st_time = omp_get_wtime();
    referenced_vec = consistent(a, b, n);
    total_time = omp_get_wtime() - st_time;
    printf("\nTime of work consistent is: %f", total_time);

    st_time = omp_get_wtime();
    sum_vec = omp_static(a, b, n);
    total_time = omp_get_wtime() - st_time;
    printf("\n\nStatic omp");
    test(sum_vec, referenced_vec, n);
    free(sum_vec);
    printf("\nTime of work reduce: %f ", total_time);

    st_time = omp_get_wtime();
    sum_vec = omp_dynamic(a, b, n);
    total_time = omp_get_wtime() - st_time;
    printf("\n\nDynamic omp");
    test(sum_vec, referenced_vec, n);
    free(sum_vec);
    printf("\nTime of work reduce: %f ", total_time);

    st_time = omp_get_wtime();
    sum_vec = omp_guided(a, b, n);
    total_time = omp_get_wtime() - st_time;
    printf("\n\nGuided omp");
    test(sum_vec, referenced_vec, n);
    free(sum_vec);
    printf("\nTime of work reduce: %f ", total_time);

    free(a);
    free(b);
    free(referenced_vec);

    return 0;
}
```

ПРИЛОЖЕНИЕ В

Код последовательной программы

```
#include <cstdio>
#include <time.h>
int main(int argc, char* argv[]) {
	int Q = 18;
	int n = 4500000;
int i, j, k, timeCount;
int a[n], b[n], sum[n];
	for ( j = 0; j < n; j++) {
	a[j] = 1;
	b[j] = 1;
	sum[j] = 0;
	}

	double st_time, end_time = 0;

	for (timeCount = 0; timeCount < 12; timeCount++) {
	st_time = clock();
	for ( k = 0; k < n; k++) {
		for ( j = 0; j < Q; j++) {
		sum[k] += a[k] + b[k];
		}
	}
	end_time += clock() - st_time;
	}
	printf("\nSEQUENCE: %f ", end_time / 12);
	printf("\nsum[0]=%f, a[0]=%f, b[0]=%f", sum[0], a[0], b[0]);
	printf("\nsum[1]=%f, a[1]=%f, b[1]=%f", sum[1], a[1], b[1]);
	printf("\nsum[2]=%f, a[2]=%f, b[2]=%f", sum[2], a[2], b[2]);
	return 0;
}
```