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 6408

Студент Д.О. Абрамов

(*подпись*)

Преподаватель,

к.ф.-м.н. Е.С. Козлова

(*подпись*)

Самара 2022

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 | double |
| N | 8200000 |
| Количество процессов/потоков | [4, 8, 16] |
| Q | 12 |

ВВЕДЕНИЕ

Сейчас почти невозможно найти современную компьютерную систему без многоядерного процессора. Даже недорогие мобильные телефоны предлагают пару ядер под капотом. Идея многоядерных систем проста: это относительно эффективная технология для масштабирования потенциальной производительности процессора. Эта технология стала широкодоступной около двадцати лет назад, и теперь каждый современный разработчик способен создать приложение с параллельным выполнением для использования такой системы. Cложность параллельного программирования часто недооценивается [1].

Сейчас почти невозможно найти современную компьютерную систему без многоядерного процессора [2]. Развитие средств параллельного программирования на этом не останавливается [3].

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 и 2 представлены скрины запуска и работы программ без параметра Q.

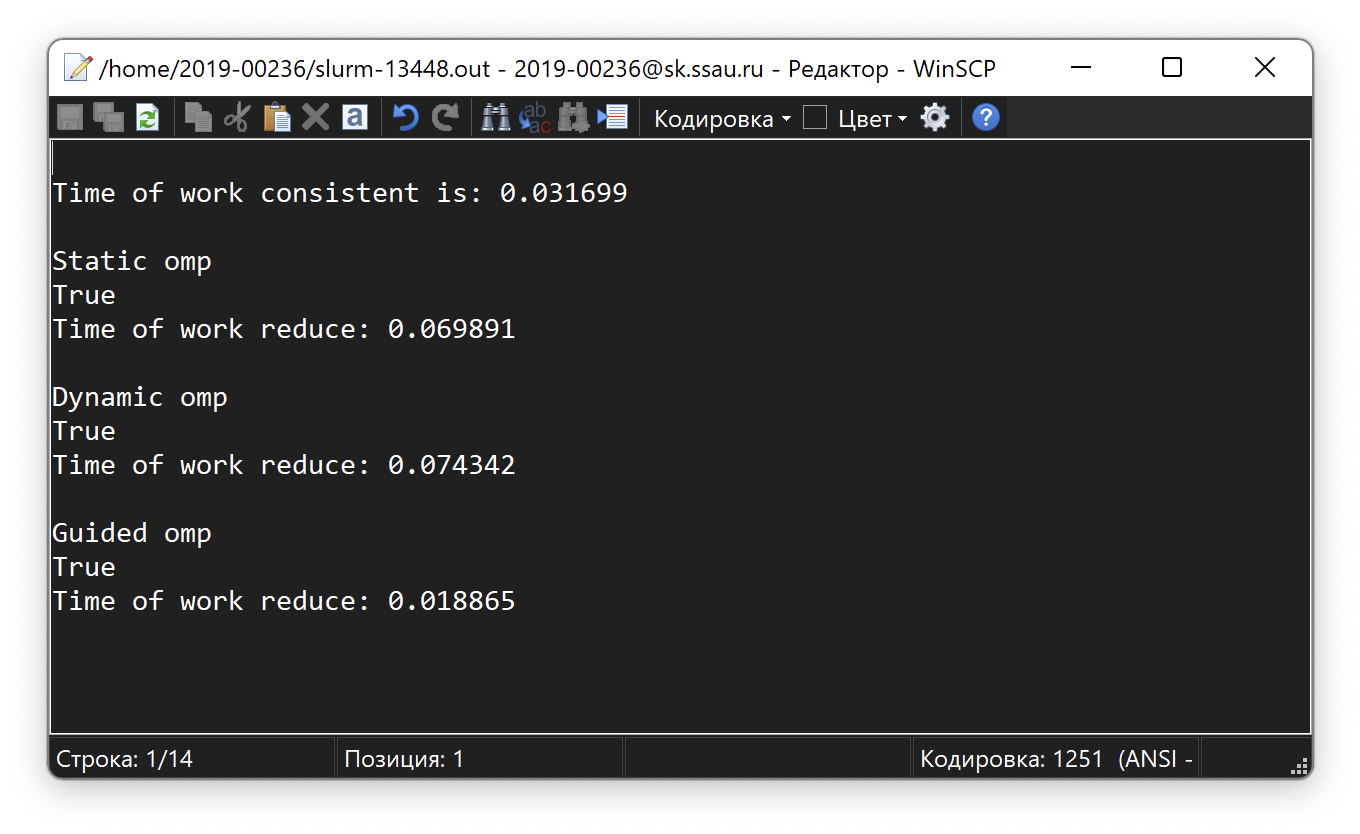


Рисунок 1 – Пример работы программы OpenMP на 4 процессах

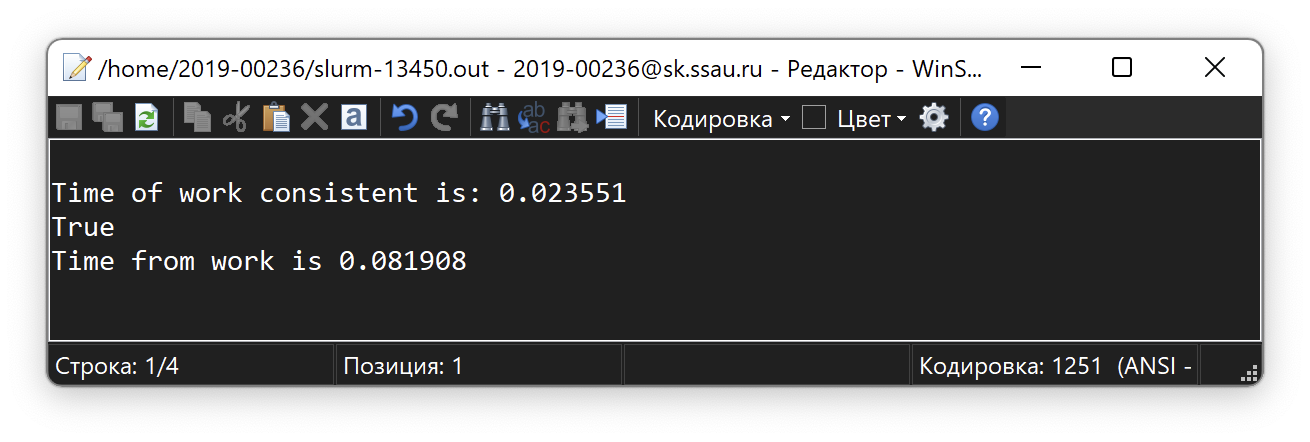


Рисунок 2 – Пример работы программы MPI на 4 процессах для кратных размерностей

Последовательная программа представляла собой программу для подсчёта суммы двух векторов одним процессом/потоком. Время работы последовательной программы составило 17577 мкс.

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, мкс | | | Время для MPI, мкс | |
| static | dynamic | quided | кратное | некратное |
| 4 | 68780 | 73231 | 21789 | 80897 | 107088 |
| 8 | 99936 | 65689 | 21789 | 101166 | 93607 |
| 16 | 119940 | 64178 | 15212 | 113663 | 117453 |

Таблица 4 – Ускорение параллельных программ без параметра Q

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во процессов, шт | Ускорение для OpenMP | | | Ускорение для MPI | |
| static | dynamic | quided | кратное | некратное |
| 4 | 0.25555 | 0.24002 | 0.08066 | 0.21727 | 0.16413 |
| 8 | 0.01758 | 0.26757 | 0.80669 | 0.17374 | 0.18777 |
| 16 | 0.14654 | 0.27387 | 1.15546 | 0.11564 | 0.14965 |

На рисунке 3 приведен график зависимости времени работы программ от количества процессов. На рисунке 4 приведен график зависимости ускорения программ от количества процессов.

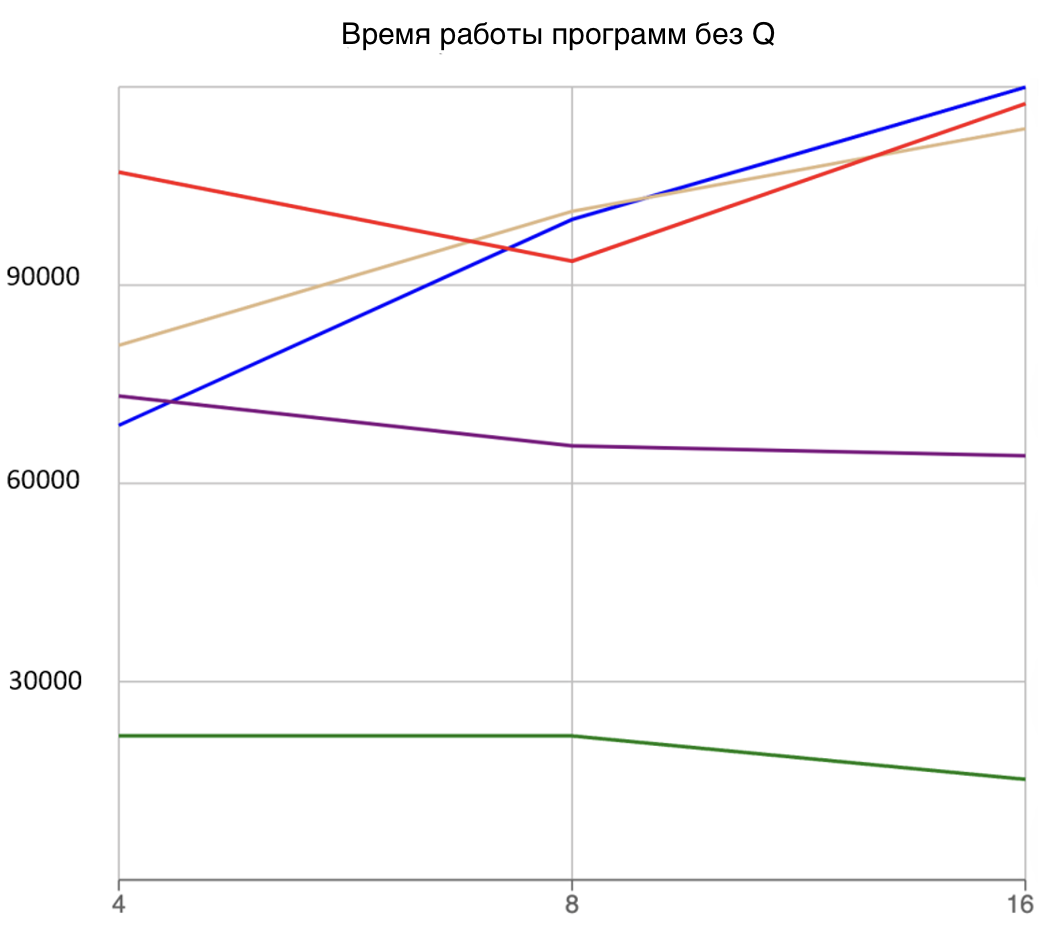


Рисунок 3 – Время работы программ без параметра Q

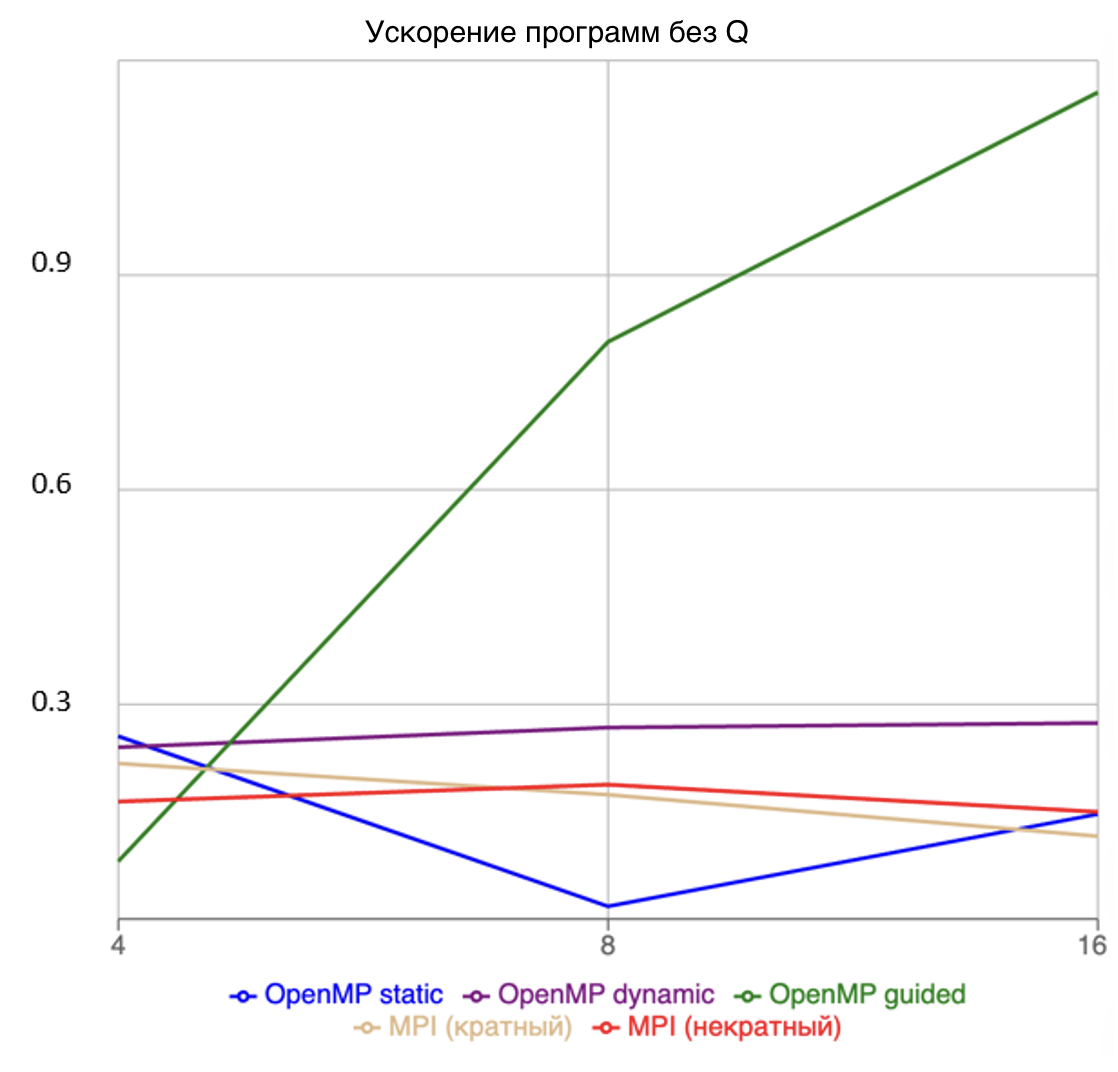


Рисунок 4 – Ускорение программ без параметра Q

2.3 Результаты работы программ с параметром Q

На рисунках 5 и 6 представлены скрины запуска и работы программ с параметром Q.

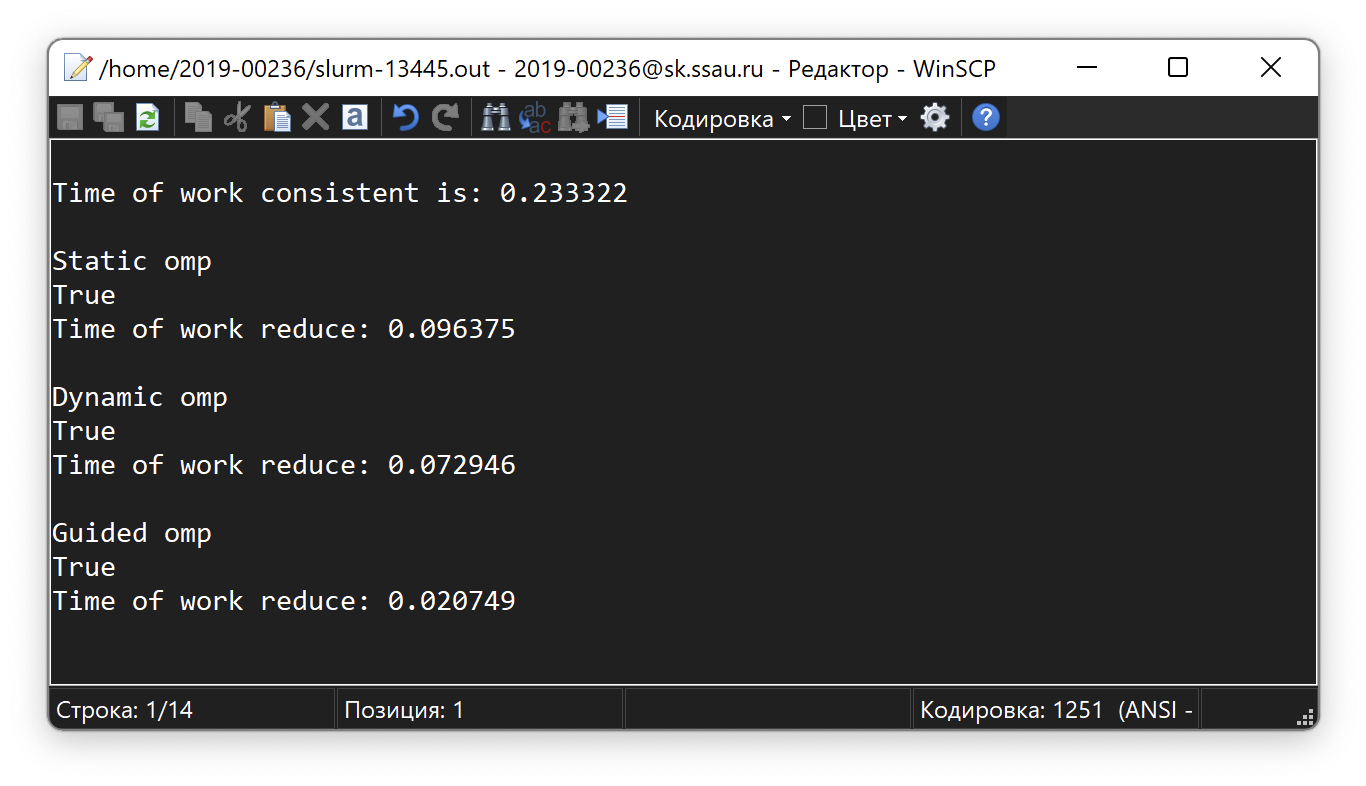


Рисунок 5 – Пример работы программы OpenMP на 4 процессах

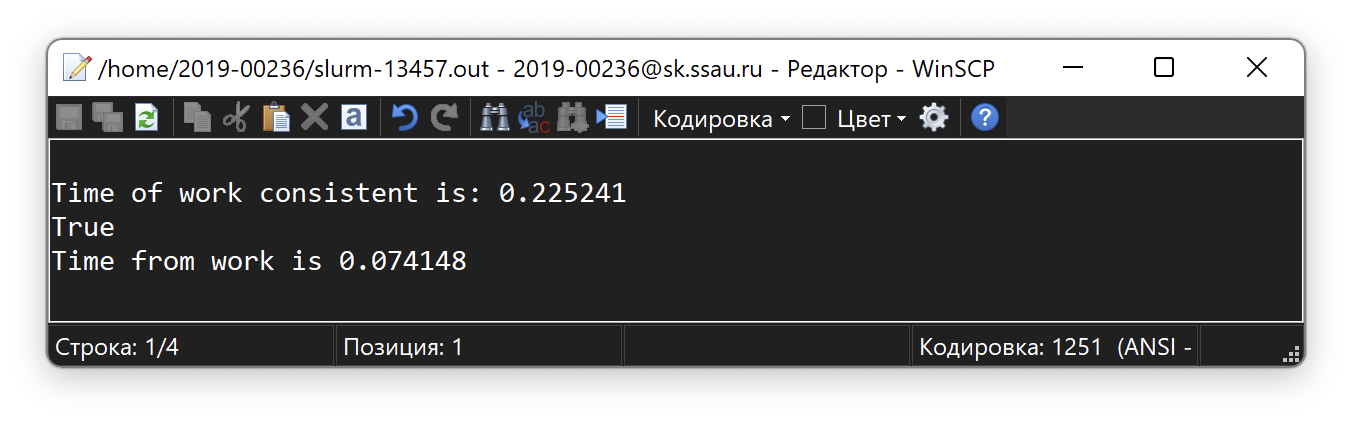


Рисунок 6 – Пример работы программы MPI на 4 процессах для кратных размерностей

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 204904 мкс.

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, мкс | | | Время для MPI, мкс | |
| static | dynamic | quided | кратное | некратное |
| 4 | 95264 | 73835 | 20638 | 73037 | 76952 |
| 8 | 77105 | 60002 | 16029 | 75653 | 92069 |
| 16 | 104962 | 71329 | 22639 | 112328 | 115101 |

Таблица 6 – Ускорение параллельных программ с параметром Q

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во процессов, шт | Ускорение для OpenMP | | | Ускорение для MPI | |
| static | dynamic | quided | кратное | некратное |
| 4 | 2.150907 | 2.775161 | 9.928481 | 2.805482 | 2.662751 |
| 8 | 2.657467 | 3.414953 | 12.78333 | 2.798472 | 2.225548 |
| 16 | 1.952173 | 2.972660 | 9.050930 | 1.824157 | 1.780210 |

На рисунке 7 приведен график зависимости времени работы программ от количества процессов. На рисунке 8 приведен график зависимости ускорения программ от количества процессов.

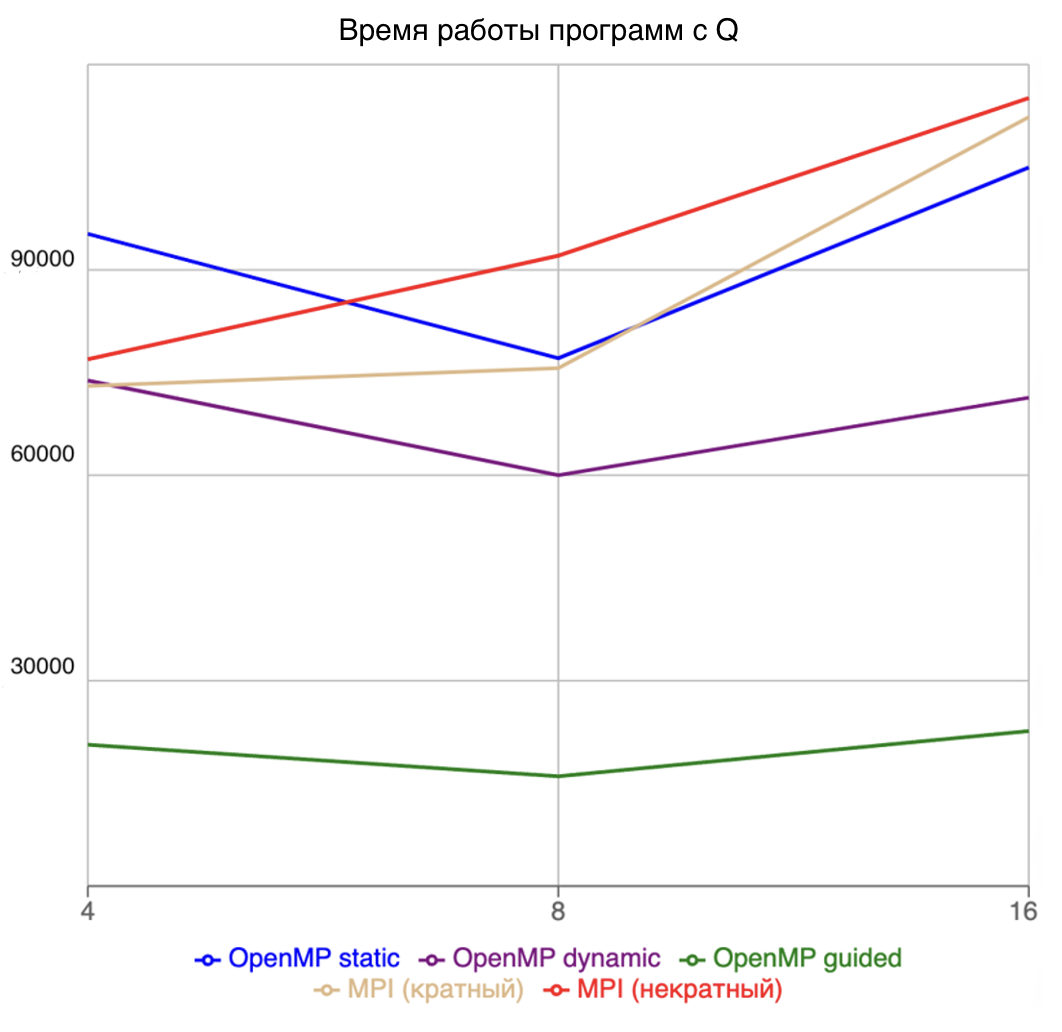


Рисунок 7 – Время работы программ с параметром Q

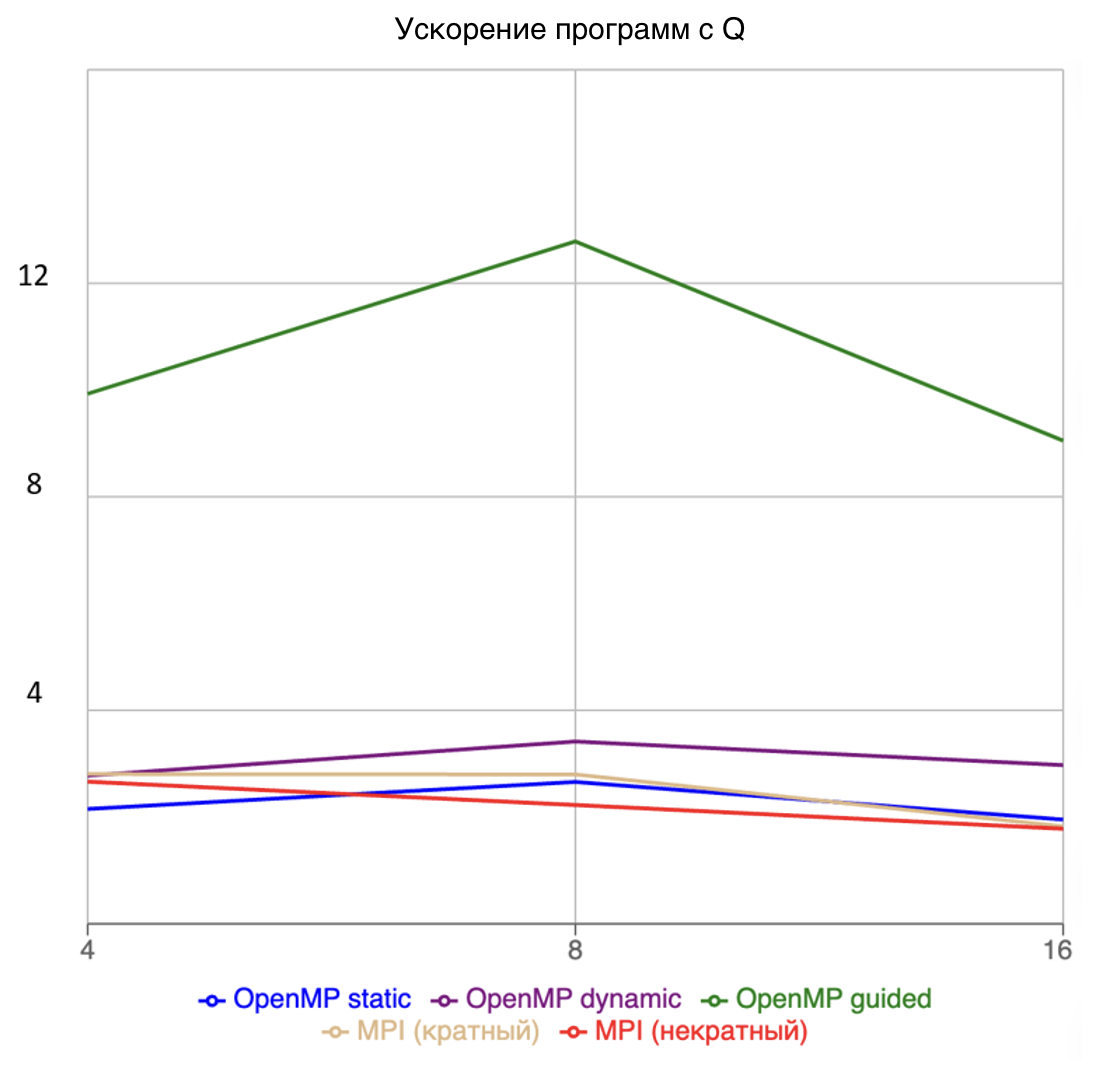


Рисунок 8 – Ускорение программ с параметром Q

**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, что можно увидеть при увеличении числа потоков до 16.
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 растёт 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, в виду того, что подсчёт частичных сумм, где каждый процесс/поток работает с частями двух векторов и всё это выполняется параллельно естественно должен работать быстрее. 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 основы MPI и OpenMP, приобрел навыки по написанию параллельных программ с использованием вышеперечисленных технологий. Наиболее сложной частью выполнения лабораторной работы было реализация программ MPI. Интерес вызвали больший темп роста ускорения программ MPI, чем у 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. Хабр, Введение в параллелизм [Электронный ресурс] / 2021. – URL: https://habr.com/ru/company/intel/blog/583286/ (дата обращения: 20.09.2022)
3. Райнер, Г. Параллельное программирование на современном С++ [Электронный ресурс] / Г. Райнер, 2022. — 618 с. – URL: https://www.rulit.me/data/programs/resources/pdf/Grimm\_Parallelnoe-programmirovanie-na-sovremennom-yazyke-C-\_RuLit\_Me\_705793.pdf (дата обращения: 20.09.2022)
4. Козлова, Е.С. Лабораторные работы по курсу «Параллельное программирование»: Методические указания [Текст] / Сост. Е.С. Козлова, А.С. Широканев − Самара, 2019. – 61 с.
5. оеводин, В. В. Параллельные вычисления [Текст] / В. В. Воеводин, Вл. В. Воеводин. — СПб.: БХВ-Петербург, 2002. — 608 с.
6. Богачёв К.Ю. Основы параллельного программирования: учебное пособие, 2-е изд. [Текст] / К. Ю. Богачёв ‑ М. : БИНОМ. Лаборатория знаний, 2013. ‑ 344 с.
7. Гергель, В. П. Теория и практика параллельных вычислений, 2-е изд. [Текст] / B. II. Гергель. — М.: Интуит. 2016. - 500 с.
8. Боресков А.В. Параллельные вычисления на GPU. Архитектура и программная модель CUDА Учеб. пособие [Текст] / А.В. Боресков ‑ М.: Издательство Московского университета, 2012. - 336 с.
9. 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 MPI Scatter/Gather

#include **"mpi.h"**#include **<cstdio>**#include **<cstdlib>**#include **<ctime>  
  
  
void** sum(**const double**\* a, **const double**\* b, **double**\* res, **int** n){  
 **for** (**int** i = 0; i < n; ++i) **for** (**int** j = 0; j < 12; ++j) res[i] = a[i] + b[i];  
}  
  
**double**\* consistent(**const double**\* vec1, **const double**\* vec2, **int** n){  
 **double** \* sum = (**double**\*)calloc(**sizeof**(**double**), n);  
 **for**(**int** i = 0; i < n; ++i) **for** (**int** j = 0; j < 12; ++j) sum[i] = vec1[i] + vec2[i];  
 **return** sum;  
}  
  
**void** test(**const double**\* vec1, **const double**\* vec2, **int** n){  
 **bool** flag = **true**;  
 **for** (**int** i = 0; i < n && flag; ++i) flag = vec1[i] == vec2[i];  
 **if** (flag) printf(**"\nTrue"**);  
 **else** printf(**"\nFalse"**);  
}  
  
**int** main(**int** argc, **char**\* argv[]){  
 srand(time(**NULL**));  
 **int** p, nu, N = 8200 \* 1000;  
 **double** \*a, \*b, \*c, \*real\_sum;  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &nu);  
 **if** (nu == 0){  
 a = (**double**\*)calloc(**sizeof**(**double**), N);  
 b = (**double**\*)calloc(**sizeof**(**double**), N);  
 c = (**double**\*)calloc(**sizeof**(**double**), N);  
 real\_sum = (**double** \*)calloc(**sizeof**(**double**), N);  
 **for** (**int** i = 0; i < N; ++i){  
 a[i] = 100 + 10 \* rand() % 100;  
 b[i] = 100 + 10 \* rand() % 100;  
 }  
 }  
 **double** st\_time;  
 **int** offset = N / p;  
 **double**\* a\_loc = (**double** \*)calloc(**sizeof**(**double**), offset);  
 **double**\* b\_loc = (**double** \*)calloc(**sizeof**(**double**), offset);  
 **double**\* c\_loc = (**double** \*)calloc(**sizeof**(**double**), offset);  
 MPI\_Scatter(a, offset, MPI\_DOUBLE, a\_loc, offset, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
 MPI\_Scatter(b, offset, MPI\_DOUBLE, b\_loc, offset, MPI\_DOUBLE, 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, offset);  
 MPI\_Gather(c\_loc, offset, MPI\_DOUBLE, c, offset, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
 **if** (nu == 0){  
 test(c, real\_sum, N);  
 printf(**"\nTime from work is %f\n"**, MPI\_Wtime() - st\_time);  
 free(real\_sum);  
 free(c);  
 }  
 free(a\_loc);  
 free(b\_loc);  
 free(c\_loc);  
 MPI\_Finalize();  
 **return** 0;  
}

ПРИЛОЖЕНИЕ Б

Код программы MPI Scatterv/Gatherv на языке C++

#include **"mpi.h"**#include **<cstdio>**#include **<cstdlib>**#include **<ctime>  
  
void** sum(**const double**\* a, **const double**\* b, **double**\* res, **int** n){  
 **for** (**int** i = 0; i < n; ++i) **for** (**int** j = 0; j < 12; ++j) res[i] = a[i] + b[i];  
}  
  
**double**\* consistent(**const double** \* vec1, **const double**\* vec2, **int** n){  
 **double**\* sum = (**double**\*)calloc(**sizeof**(**double**), n);  
 **for**(**int** i = 0; i < n; ++i) **for** (**int** j = 0; j < 12; ++j) sum[i] = vec1[i] + vec2[i];  
 **return** sum;  
}  
  
**void** test(**const double**\* vec1, **const double**\* vec2, **double** n){  
 **bool** flag = **true**;  
 **for** (**int** i = 0; i < n && flag; ++i) flag = vec1[i] == vec2[i];  
 **if** (flag) printf(**"\nTrue"**);  
 **else** printf(**"\nFalse"**);  
}  
  
**int** main(**int** argc, **char**\* argv[]){  
 srand(time(**NULL**));  
 **int** p, nu, N = 8200 \* 1000, \*sendCounts, \*displs;  
 **double** \*a, \* b, \*c, \*real\_sum;  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &nu);  
 **if** (nu == 0){  
 a = (**double**\*)calloc(**sizeof**(doubl), N);  
 b = (**double**\*)calloc(**sizeof**(**double**), N);  
 c = (**double**\*)calloc(**sizeof**(**double**), N);  
 real\_sum = (**double**\*)calloc(**sizeof**(**double**), 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;  
 **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];  
 **double**\* a\_loc = (**double**\*)calloc(**sizeof**(**double**), sendCounts[nu]);  
 **double**\* b\_loc = (**double**\*)calloc(**sizeof**(**double**), sendCounts[nu]);  
 **double**\* c\_loc = (**double**\*)calloc(**sizeof**(**double**), sendCounts[nu]);  
 MPI\_Scatterv(a, sendCounts, displs, MPI\_DOUBLE, a\_loc, sendCounts[nu], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
 MPI\_Scatterv(b, sendCounts, displs, MPI\_DOUBLE, b\_loc, sendCounts[nu], MPI\_DOUBLE, 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\_DOUBLE, c, sendCounts, displs, MPI\_DOUBLE, 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 В

Код с технологией OpenMP

#include **<omp.h>**#include **<cstdio>**#define **THREAD\_COUNT** 4  
#define **Q** 12  
  
**double** \*consistent(**const double** \*vec1, **const double** \*vec2, **int** n) {  
 **double** \*sum = (**double** \*) calloc(**sizeof**(**double**), n);  
 **for** (**int** i = 0; i < n; ++i) **for** (**int** j = 0; j < **Q**; ++j) sum[i] = vec1[i] + vec2[i];  
 **return** sum;  
}  
  
**double** \*omp\_static(**const double** \*vec1, **const double** \*vec2, **int** n) {  
 **double** \*sum = (**double** \*) calloc(**sizeof**(**double**), n);  
#pragma omp parallel **for** shared(vec1, vec2, sum) schedule(**static**, 15)  
 **for** (**int** i = 0; i < n; ++i) **for** (**int** j = 0; j < **Q**; ++j)sum[i] = vec1[i] + vec2[i];  
 **return** sum;  
}  
  
**double** \*omp\_dynamic(**const double** \*vec1, **const double** \*vec2, **int** n) {  
 **double** \*sum = (**double** \*) calloc(**sizeof**(**double**), n);  
#pragma omp parallel **for** shared(vec1, vec2, sum) schedule(dynamic, 15)  
 **for** (**int** i = 0; i < n; ++i) **for** (**int** j = 0; j < **Q**; ++j) sum[i] = vec1[i] + vec2[i];  
 **return** sum;  
}  
  
**double** \*omp\_guided(**const double** \*vec1, **const double** \*vec2, **int** n) {  
 **double** \*sum = (**double** \*) calloc(**sizeof**(**double**), n);  
#pragma omp parallel **for** shared(vec1, vec2, sum) schedule(guided, 15)  
 **for** (**int** i = 0; i < n; ++i) **for** (**int** j = 0; j < **Q**; ++j) sum[i] = vec1[i] + vec2[i];  
 **return** sum;  
}  
  
**void** test(**const double** \*vec1, **const double** \*vec2, **int** n) {  
 **bool** flag = **true**;  
 **for** (**int** i = 0; i < n && flag; ++i) flag = vec1[i] == vec2[i];  
 **if** (flag) printf(**"\nTrue"**);  
 **else** printf(**"\nFalse"**);  
}  
  
**int** main(**int** argc, **char** \*argv[]) {  
 omp\_set\_num\_threads(**THREAD\_COUNT**);  
 **int** n = 8200 \* 1000;  
 **double** st\_time, total\_time;  
 **double** \*a = (**double** \*) calloc(**sizeof**(**double**), n);  
 **double** \*b = (**double** \*) calloc(**sizeof**(**double**), n);  
 **double** \*sum\_vec;  
 **double** \*referenced\_vec;  
 **for** (**int** 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\n"**, total\_time);  
 free(a);  
 free(b);  
 free(referenced\_vec);  
 **return** 0;  
}

ПРИЛОЖЕНИЕ Г

Код последовательной программы

#include **<cstdio>**#include **<ctime>  
int** main(**int** argc, **char**\* argv[]) {  
 **int** Q = 12;  
 **int** n = 8200000;  
 **double** \*a = **new double** [n], \*sum = **new double** [n], \*b = **new double** [n];  
 *// инициализация массивов* **for** (**int** j = 0; j < n; j++) {  
 a[j] = 1;  
 b[j] = 1;  
 sum[j] = 0;  
 }  
 *//суммирование векторов* **double** st\_time, end\_time = 0;  
 *//последовательная программа* **for** (**int** timeCount = 0; timeCount < 12; timeCount++) {  
 st\_time = clock();  
 **for** (**int** k = 0; k < n; k++) {  
 **for** (**int** 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;  
}