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 № 2

по курсу   
Параллельное программирование

Группа 6408

Студент Д.О. Абрамов

(*подпись*)

Преподаватель,

к.ф.-м.н. Е.С. Козлова

(*подпись*)

(*подпись*)

Самара 2022

ЗАДАНИЕ

**Произвести запуск программ** для поиска суммы элементов**, которые используют технологии MPI и OpenMP, на различном количестве процессоров (потоков). Для упрощения процесса написания программы допускается использование размерности вектора, кратной количеству процессоров.**

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

**Для дальнейшего анализа технологий необходимо искусственно увеличить соотношение вычислений и операций по обеспечению параллелизма путем повторения функции поиска суммы элементов Q раз. В ходе анализа работы программ 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 № 2

|  |  |
| --- | --- |
| 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 Характеристики ПК

В таблице 2 представлены характеристики ПК.

Таблица 2 – Характеристики ПК

|  |  |
| --- | --- |
| Процессор | Apple M1 |
| Количество ядер | 8 |
| Количество потоков | 8 |
| Оперативная память | 16 Гб |
| Тип системы | ARMv8.5-A |

2.2 Результаты работы программ без параметра Q

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

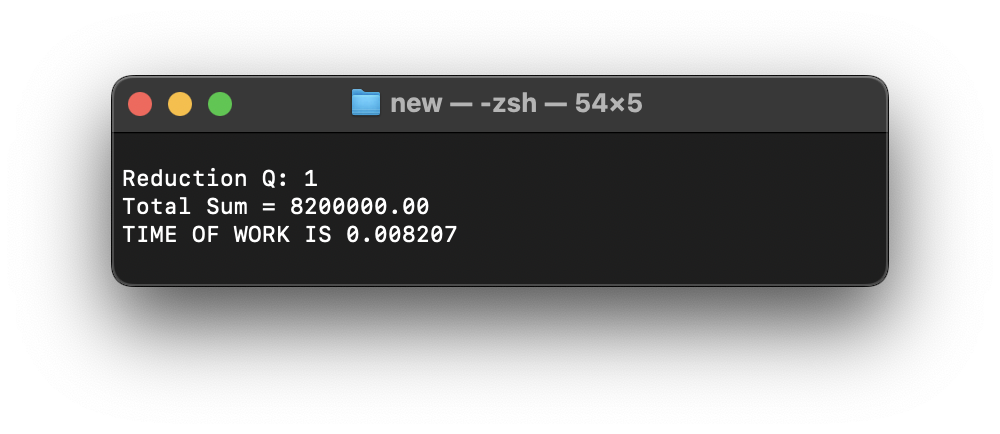


Рисунок 1 – Пример работы программы OpenMP на 4 процессах для reduction

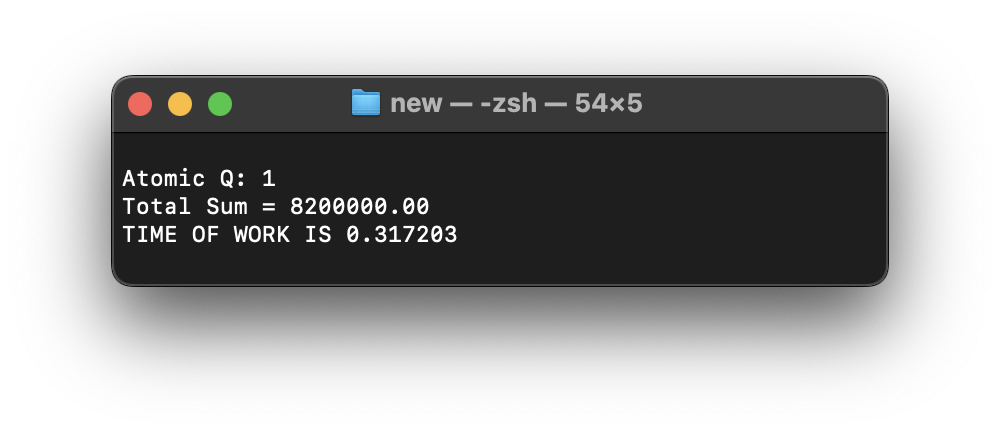


Рисунок 2 – Пример работы программы OpenMP на 4 процессах для atomic

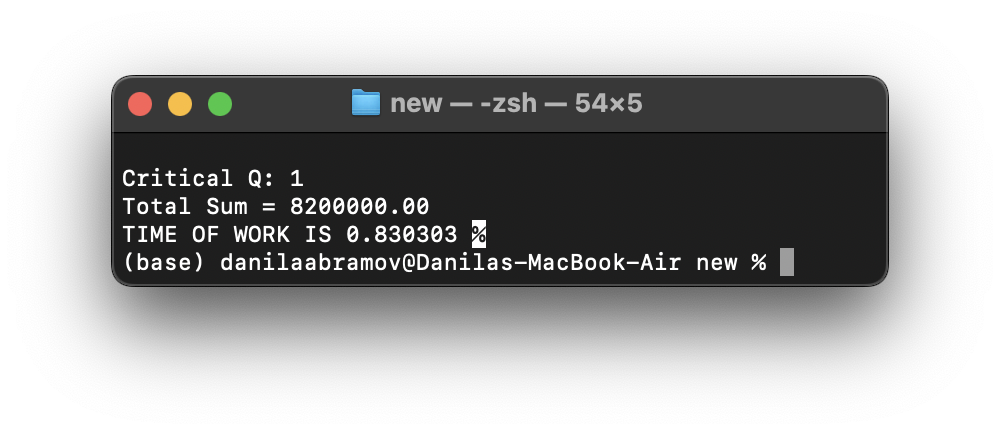


Рисунок 3 – Пример работы программы OpenMP на 4 процессах для critical

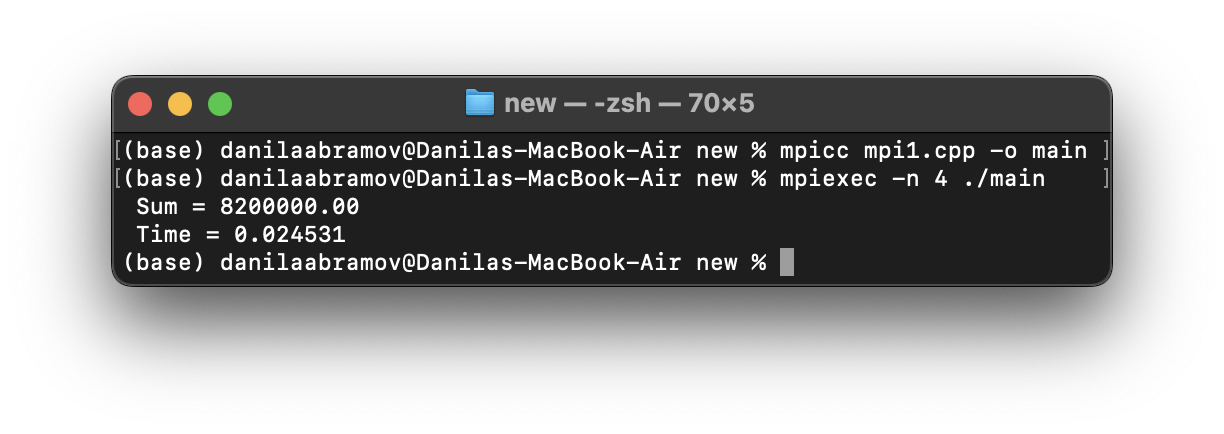


Рисунок 4 – Пример работы программы MPI на 4 процессах для операции «точка-точка»

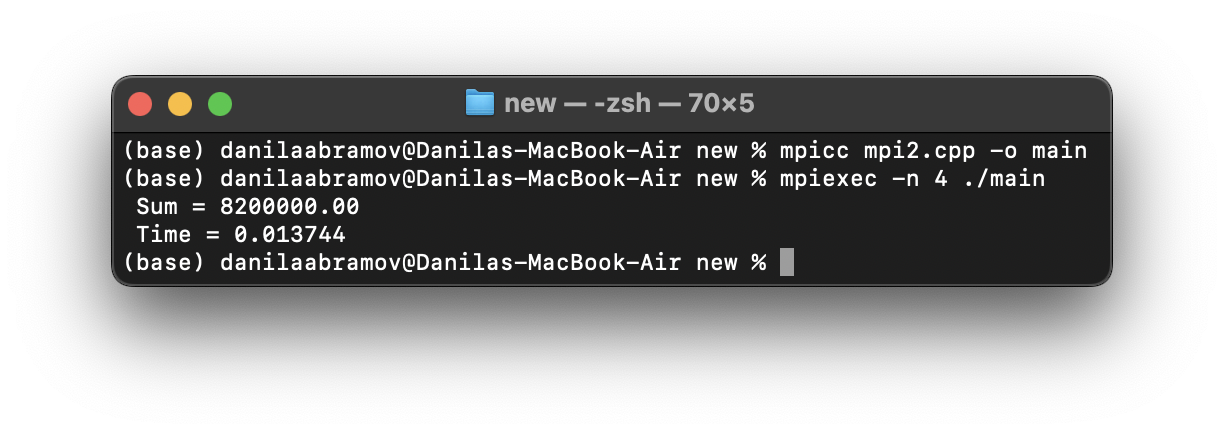


Рисунок 5 – Пример работы программы MPI на 4 процессах для коллективной операции

Последовательная программа представляла собой программу подсчёта суммы элементов вектора и вывод этой суммы в консоль одним процессом/потоком. Время работы последовательной программы составило без параметра Q – 29644 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 | |
| critical | atomic | reduction | «точка-точка» | Коллективная |
| 4 | 830303 | 317203 | 8207 | 24432 | 13643 |
| 8 | 1241308 | 661084 | 8381 | 12230 | 15739 |
| 16 | 1102347 | 709124 | 9412 | 143220 | 79112 |

Таблица 4 – Ускорение параллельных программ без параметра Q

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во процессов, шт | Ускорение для OpenMP | | | Ускорение для MPI | |
| critical | atomic | reduction | «точка-точка» | Коллективная |
| 4 | 0,03570 | 0,09345 | 3,61204 | 1,21333 | 2,17284 |
| 8 | 0,02388 | 0,04484 | 3,53071 | 2,42388 | 1,88347 |
| 16 | 0,02689 | 0,04184 | 3,14960 | 0,20699 | 0,37471 |

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

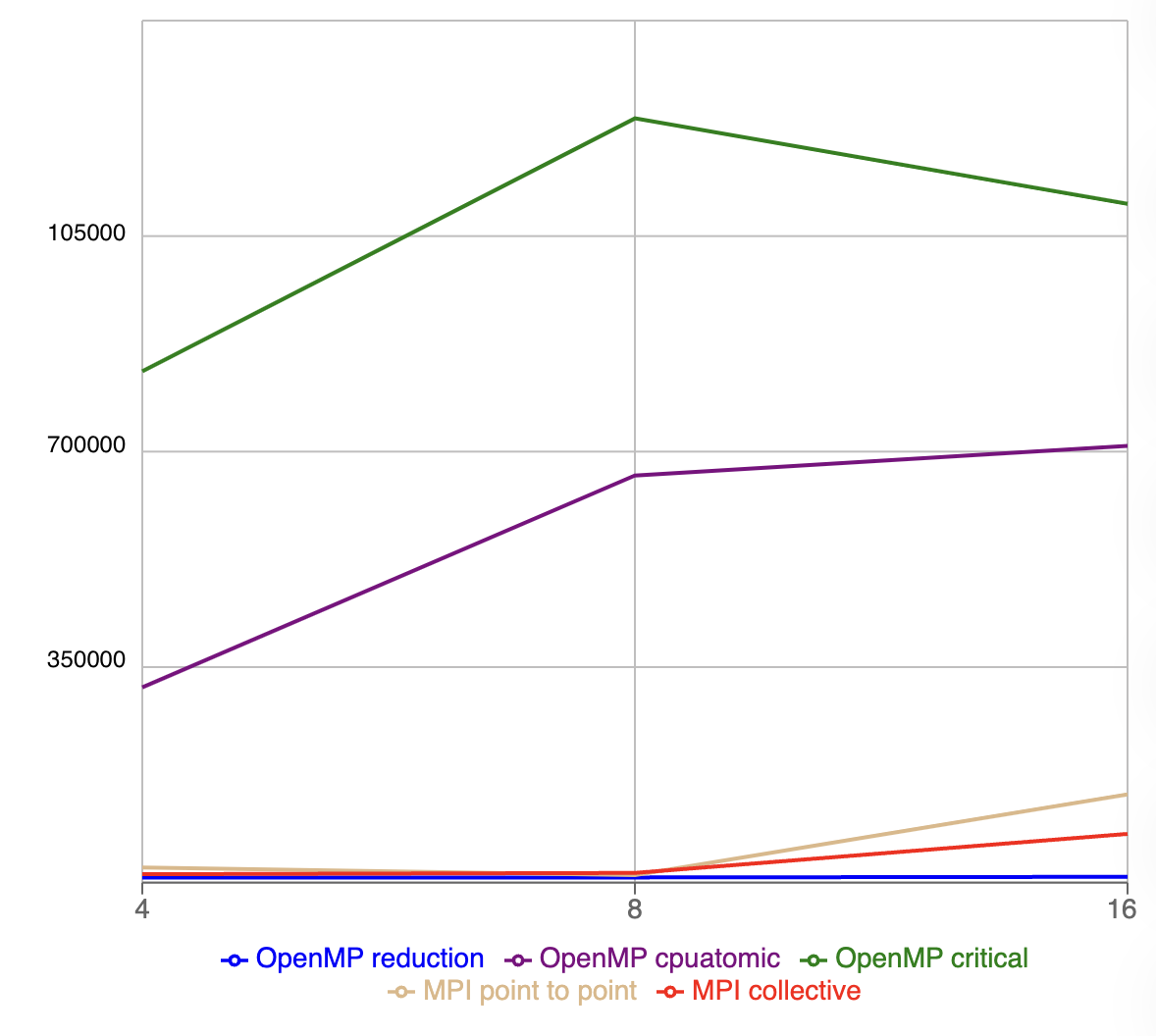


Рисунок 6 – Время работы программ без параметра Q

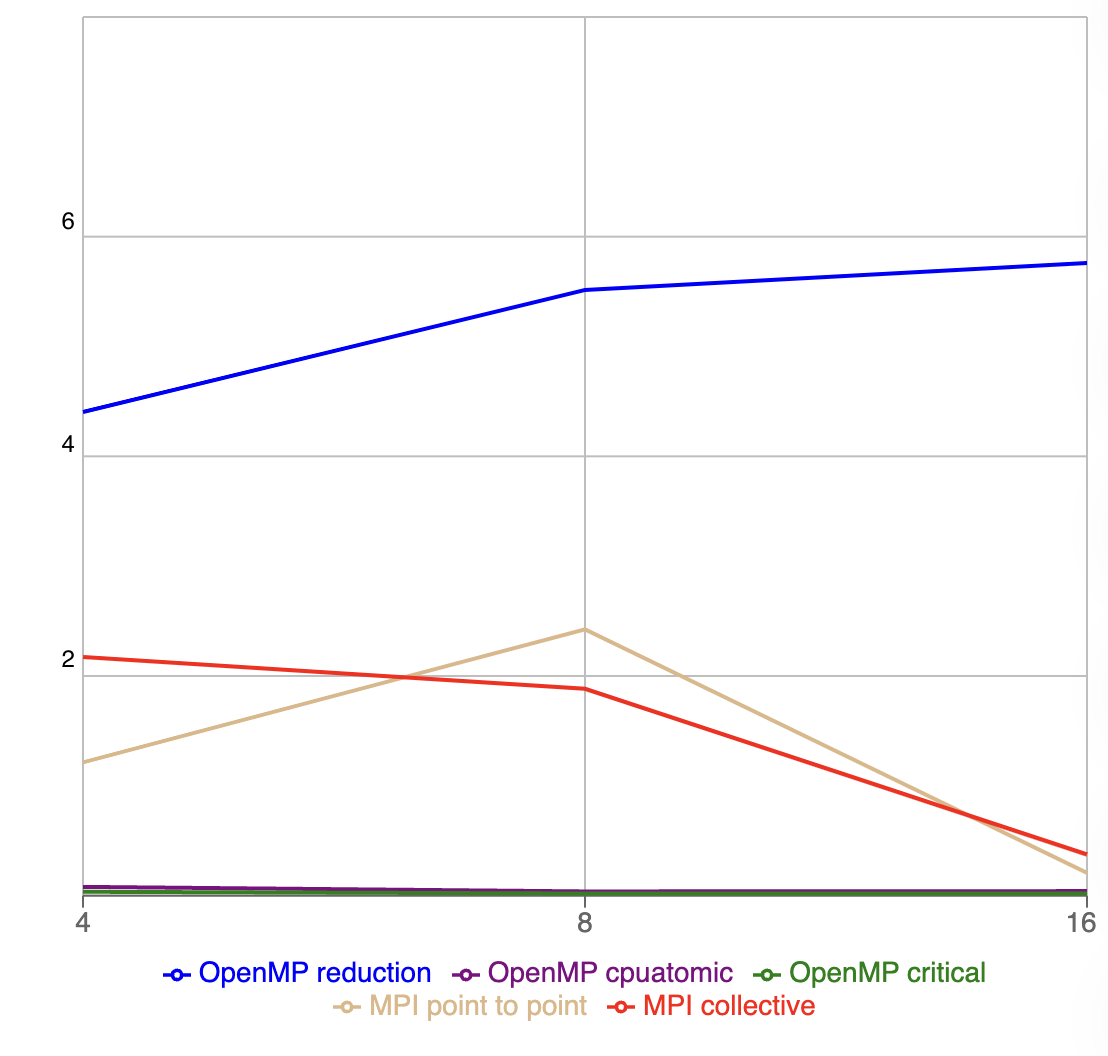


Рисунок 7– Ускорение программ без параметра Q

2.3 Результаты работы программ с параметром Q

На рисунках 8-12 представлены скрины запуска и работы программ с параметром Q.

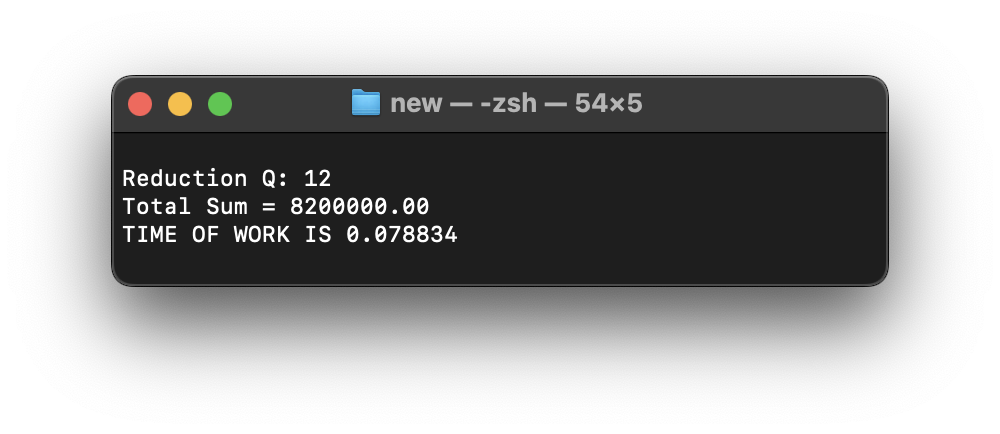


Рисунок 8 – Пример работы программы OpenMP на 4 процессах для reduction

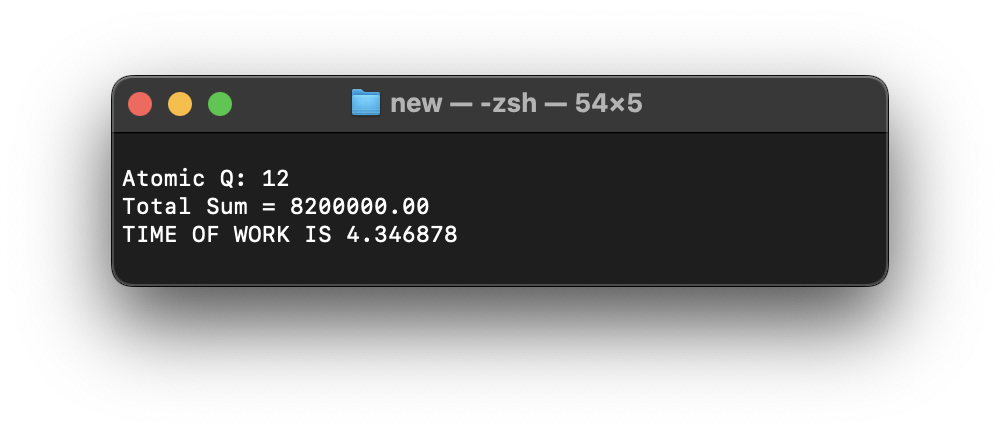


Рисунок 9 – Пример работы программы OpenMP на 4 процессах для atomic

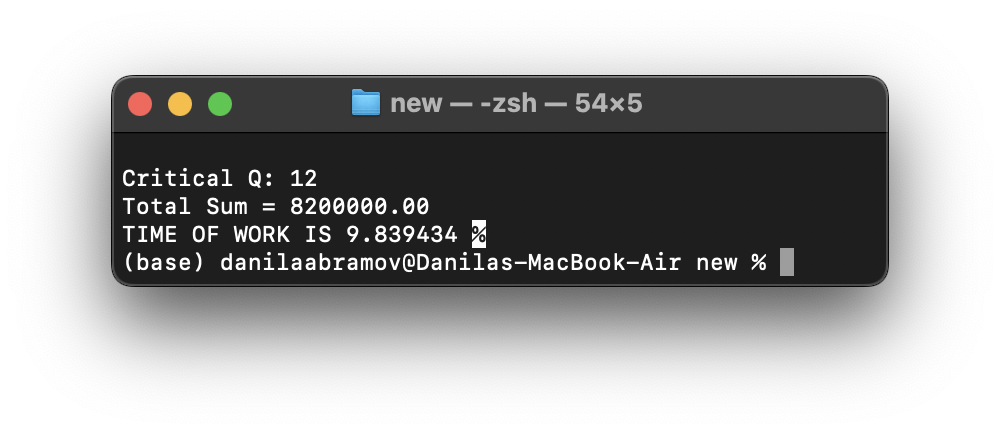


Рисунок 10 – Пример работы программы OpenMP на 4 процессах для critical

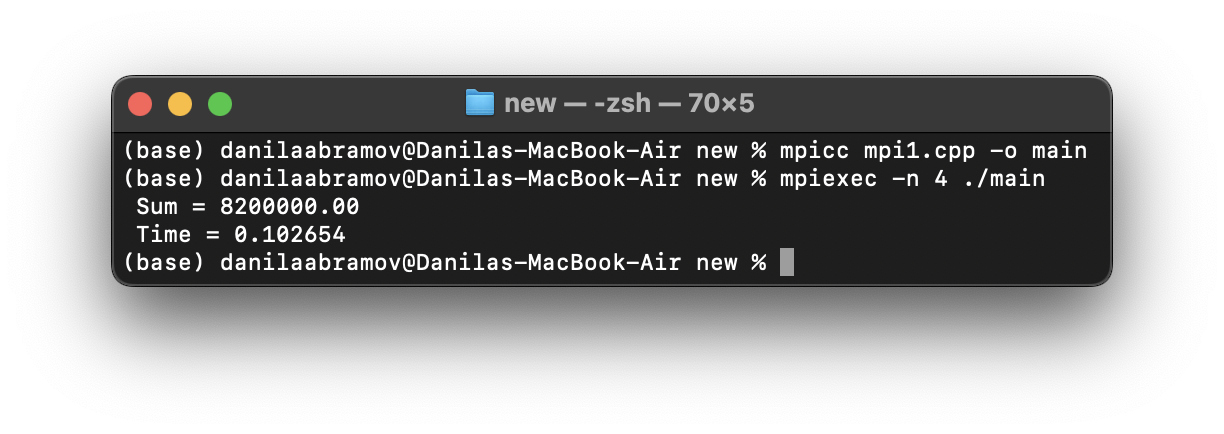


Рисунок 11 – Пример работы программы MPI на 4 процессах для операции «точка-точка»

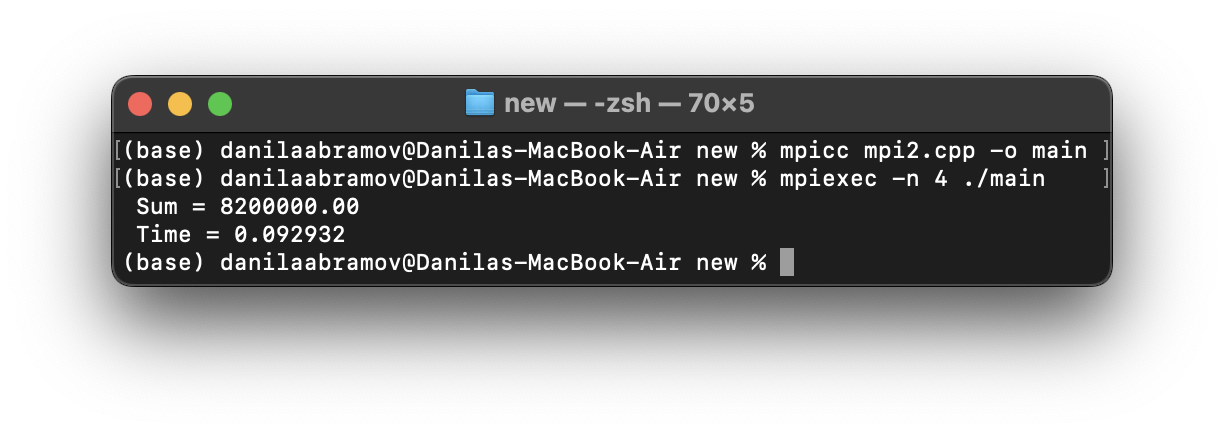


Рисунок 12 – Пример работы программы MPI на 4 процессах для коллективной операции

Усложненная последовательная программа представляла собой программу подсчёта суммы элементов вектора и вывод этой суммы в консоль одним процессом/потоком с параметром пересчёта Q в цикле. Время работы последовательной программы с параметром Q – 347132 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 | |
| critical | atomic | reduction | «точка-точка» | Коллективная |
| 4 | 9839434 | 4346878 | 78834 | 102753 | 92833 |
| 8 | 17887753 | 9007996 | 62939 | 87828 | 67194 |
| 16 | 18566444 | 8377027 | 60260 | 111755 | 141122 |

Таблица 6 – Ускорение параллельных программ с параметром Q

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во процессов, шт | Ускорение для OpenMP | | | Ускорение для MPI | |
| critical | atomic | reduction | «точка-точка» | Коллективная |
| 4 | 0,03527 | 0,07986 | 4,40333 | 3,37831 | 3,73932 |
| 8 | 0,01941 | 0,03854 | 5,51537 | 3,95241 | 5,16612 |
| 16 | 0,01869 | 0,04144 | 5,76057 | 3,10619 | 2,45980 |

На рисунке 13 приведен график зависимости времени работы программ от количества процессов. На рисунке 14 приведен график зависимости ускорения программ от количества процессов.

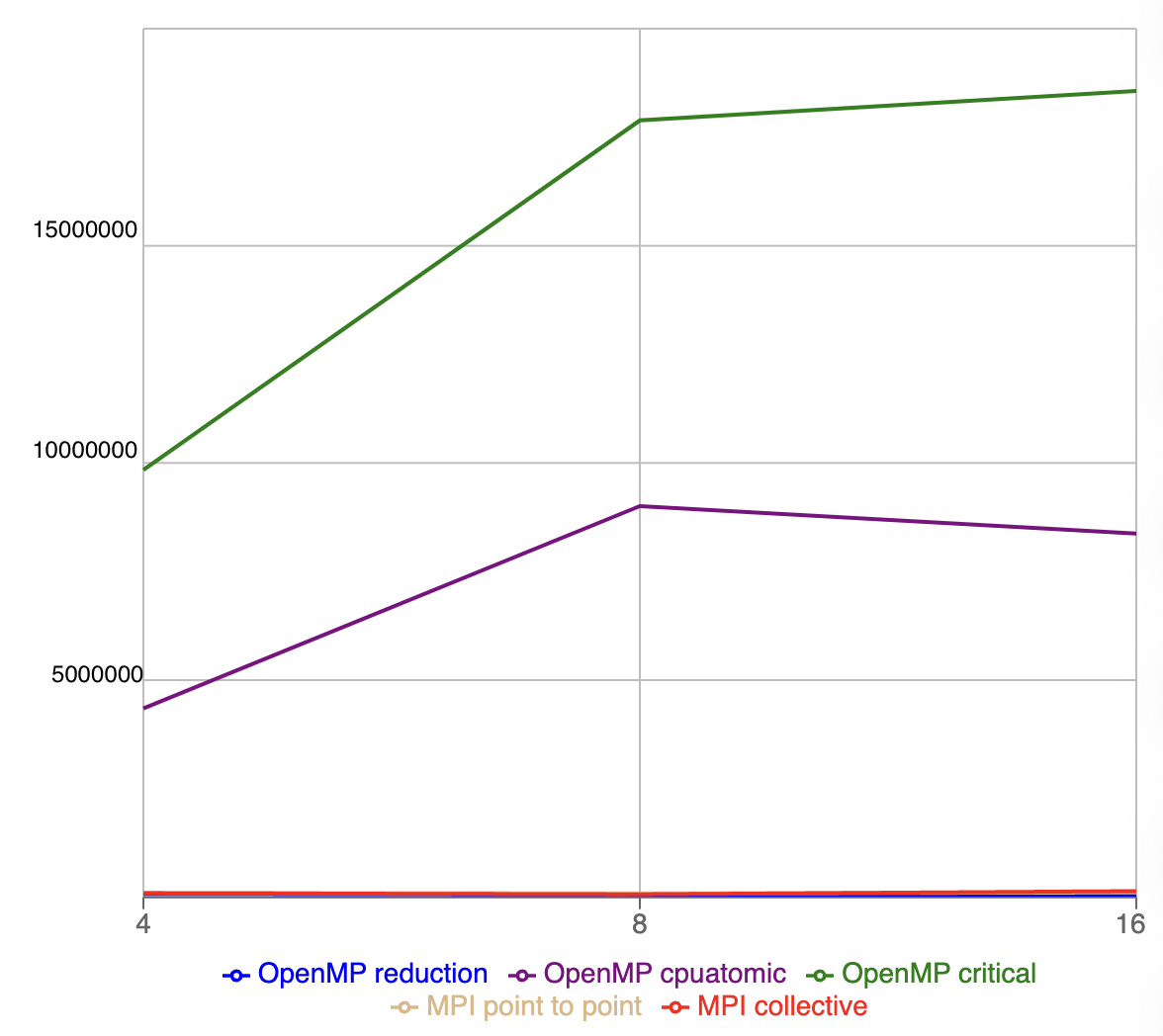


Рисунок 13 – Время работы программ с параметром Q

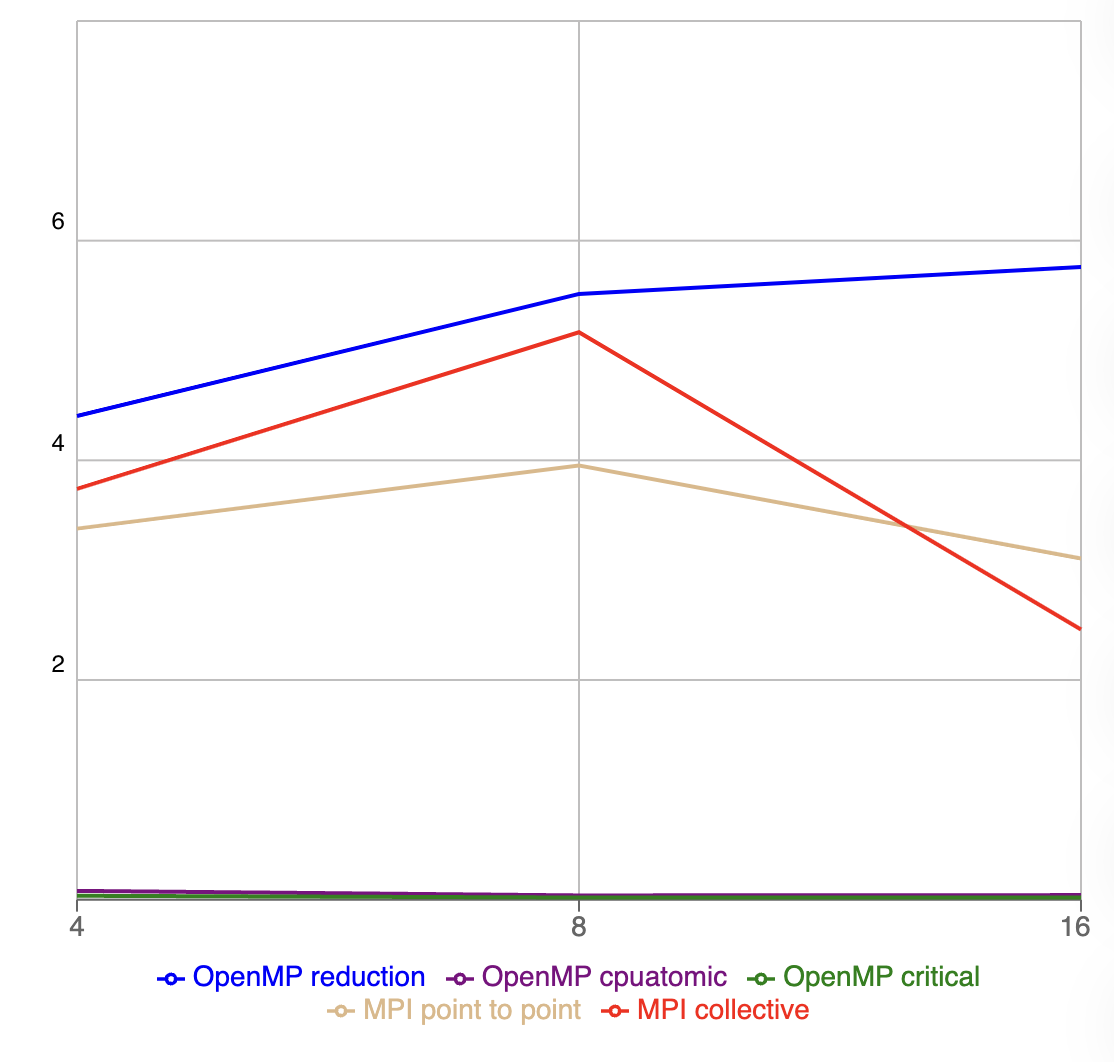


Рисунок 14 – Ускорение программ с параметром Q

**ВЫВОДЫ:**

Из полученных результатов видно, что:

1. Для программ OpenMP варианты Atomic, Critical приводят к увеличению времени выполнения программы, Reduction показал хорошие результаты и ускорение. Это связано с тем, что Atomic блокирует доступ к переменной всем запущенным на данный момент потокам, директива Critical используется для создания критической секции, которая выполняется только одним потоком. В случае с Reduction для каждого потока создаются локальные копии переменных, директива позволяет собрать в главном потоке результаты вычисления частичных сумм каждым потоком. Для программ MPI получены хорошие результаты, лучше справились коллективные операции, чем операции «точка-точка», работающие с блокировкой, что приводит к замедлению выполнения программы.
2. Максимальное ускорение для OpenMP достигает Reduction. Максимальное ускорение для MPI достигается с использованием коллективных операций. С использованием параметра Q, увеличивается время выполнения параллельной программы более быстрыми темпами, чем последовательной, ускорение достигает меньших результатов.
3. С увеличением числа процессов время выполнения программ OpenMP в случае с Reduction уменьшается, в случае с Critical и Atomic время выполнения увеличивается, использование параметра Q увеличивает время выполнения для всех вариантов. С увеличением числа процессов время работы программы MPI для операций «точка-точка» увеличивается, с параметром Q также увеличивается. С увеличением числа процессов до 8 время выполнения программ MPI в случае коллективных операций показывает уменьшение, при 16 процессах уже происходит увеличение, что связано с излишними для данной задачи затратами на распараллеливание.
4. С использованием параметра Q время выполнения для обеих технологий увеличивается, происходит больше вычислений, следовательно, время работы программ увеличивается. Для OpenMP Critical и Atomic показали схожие результаты, что связано с их принципом работы, Reduction же приблизился ко времени работы программ MPI, как с параметром Q, так и без.

ЗАКЛЮЧЕНИЕ

Цель лабораторной работы – написать параллельные программы поиска суммы элементов вектора 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 требовалось обработать большой массив данных. Лучше всего себя показала технологии MPI коллективные операции и OpenMP с использованием директивы Reduction? Это можно объяснить тем, что коллективные операции в MPI работают быстрее операций «точка-точка», которые используют блокировки. Директива Reduction в OpenMP позволяет собрать в главном потоке результаты вычисления частичных сумм каждым потоком, что также целесообразно в данной задаче.

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 зависимость ускорения от количества процессов и вида технологии.

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 MPI (point to point)

#include **<cstdio>**#include **<cstdlib>**#include **"mpi.h"  
  
int** main(**int** argc, **char** \*argv[]) {  
 **double** \*a, \*b, TotalSum, ProcSum = 0.0;  
 **int** ProcRank, ProcNum, N = 8200000, i, j, Q = 12, k;  
 MPI\_Status Status;  
 **double** st\_time, end\_time;  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_size(**MPI\_COMM\_WORLD**, &ProcNum);  
 MPI\_Comm\_rank(**MPI\_COMM\_WORLD**, &ProcRank);  
 **if** (ProcRank == 0) {  
 a = **static\_cast**<**double** \*>(malloc(N \* **sizeof**(**double**)));  
 **for** (i = 0; i < N; ++i) a[i] = 1;  
 }  
 b = **static\_cast**<**double** \*>(malloc((N / ProcNum) \* **sizeof**(**double**)));  
 MPI\_Scatter(a, N / ProcNum, **MPI\_DOUBLE**, b, N / ProcNum, **MPI\_DOUBLE**, 0, **MPI\_COMM\_WORLD**);  
 st\_time = MPI\_Wtime();  
 k = N / ProcNum;  
 **for** (i = 0; i < k; ++i) **for** (j = 0; j < Q; ++j) ProcSum = ProcSum + b[i];  
 ProcSum /= (**double**) Q;  
 **if** (ProcRank == 0)  
 **for** (i = 1, TotalSum = ProcSum; i < ProcNum; ++i) {  
 MPI\_Recv(&ProcSum, 1, **MPI\_DOUBLE**, i, 0, **MPI\_COMM\_WORLD**, &Status);  
 TotalSum = TotalSum + ProcSum;  
 }  
 **else** MPI\_Send(&ProcSum, 1, **MPI\_DOUBLE**, 0, 0, **MPI\_COMM\_WORLD**);  
 MPI\_Barrier(**MPI\_COMM\_WORLD**);  
 end\_time = MPI\_Wtime();  
 end\_time = end\_time - st\_time;  
 **if** (ProcRank == 0) {  
 printf(**" Sum = %10.2f"**, TotalSum);  
 printf(**"\n Time = %f \n"**, end\_time);  
 free(a);  
 }  
 free(b);  
 MPI\_Finalize();  
 **return** 0;  
}

# **ПРИЛОЖЕНИЕ Б**

# Код MPI (collective)

#include <cstdio>  
#include **<cstdlib>**#include **"mpi.h"  
  
int** main(**int** argc, **char** \*argv[]) {  
 **double** \*a, \*b, TotalSum, ProcSum = 0.0;  
 **int** ProcRank, ProcNum, N = 8200000, i, j, Q = 12, k;  
 **double** st\_time, end\_time;  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_size(**MPI\_COMM\_WORLD**, &ProcNum);  
 MPI\_Comm\_rank(**MPI\_COMM\_WORLD**, &ProcRank);  
 **if** (ProcRank == 0) {  
 a = **static\_cast**<**double** \*>(malloc(N \* **sizeof**(**double**)));  
 **for** (i = 0; i < N; ++i) a[i] = 1;  
 }  
 b = **static\_cast**<**double** \*>(malloc((N / ProcNum) \* **sizeof**(**double**)));  
 MPI\_Scatter(a, N / ProcNum, **MPI\_DOUBLE**, b, N / ProcNum, **MPI\_DOUBLE**, 0, **MPI\_COMM\_WORLD**);  
 **if** (ProcRank == 0) free(a);  
 st\_time = MPI\_Wtime();  
 k = N / ProcNum;  
 **for** (i = 0; i < k; ++i) **for** (j = 0; j < Q; ++j) ProcSum = ProcSum + b[i];  
 ProcSum /= (**double**) Q;  
 MPI\_Reduce(&ProcSum, &TotalSum, 1, **MPI\_DOUBLE**, **MPI\_SUM**, 0, **MPI\_COMM\_WORLD**);  
 end\_time = MPI\_Wtime();  
 end\_time = end\_time - st\_time;  
 **if** (ProcRank == 0) {  
 printf(**" Sum = %10.2f"**, TotalSum);  
 printf(**"\n Time = %f \n"**, end\_time);  
 }  
 free(b);  
 MPI\_Finalize();  
 **return** 0;  
}

# **ПРИЛОЖЕНИЕ В**

Код с технологией OpenMP

#include **<omp.h>**  
#include **"stdio.h"**

#define **NMAX** 8200000  
 **int** main(**int** argc, **char** \*argv[]) {  
 omp\_set\_num\_threads(12);  
 **int** i, j, Q = 12, count, t;  
 **double** sum, full\_time = 0;  
 **double** \*a = **new double**[**NMAX**];  
 **for** (i = 0; i < **NMAX**; ++i) {  
 a[i] = 1;  
 }  
 **double** st\_time, end\_time;  
 full\_time = 0;  
 *//reduction* **for** (t = 0; t < 12; ++t) {  
 st\_time = omp\_get\_wtime();  
#pragma omp parallel **for** shared(a) **private**(i, j) reduction(+: sum)  
 **for** (i = 0; i < **NMAX**; i++) {  
 **for** (j = 0; j < Q; ++j) {  
 sum += a[i];  
 }  
 }  
 end\_time = omp\_get\_wtime();  
 end\_time = end\_time - st\_time;  
 full\_time += end\_time;  
 }  
 sum = sum / (12 \* Q);  
 printf(**"\n\nReduction Q: %d"**, Q);  
 printf(**"\nTotal Sum = %10.2f"**, sum);  
 printf(**"\nTIME OF WORK IS %f "**, full\_time / 12);  
  
 sum = 0;  
 full\_time = 0;  
 *//atomic* **for** (t = 0; t < 12; ++t) {  
 st\_time = omp\_get\_wtime();  
#pragma omp parallel **for** shared(a) **private**(i, j)  
 **for** (i = 0; i < **NMAX**; i++) {  
  
 **for** (j = 0; j < Q; ++j) {  
#pragma omp atomic  
 sum += a[i];  
 }  
 }  
 end\_time = omp\_get\_wtime();  
 end\_time = end\_time - st\_time;  
 full\_time += end\_time;  
 }  
 sum = sum / (12 \* Q);  
 printf(**"\n\nAtomic Q: %d"**, Q);  
 printf(**"\nTotal Sum = %10.2f"**, sum);  
 printf(**"\nTIME OF WORK IS %f "**, full\_time / 12);  
 sum = 0;  
 full\_time = 0;  
  
 *//critical* **for** (t = 0; t < 12; ++t) {  
 st\_time = omp\_get\_wtime();  
#pragma omp parallel **for** shared(a) **private**(i, j)  
 **for** (i = 0; i < **NMAX**; i++) {  
 **for** (j = 0; j < Q; ++j) {  
#pragma omp critical  
 sum += a[i];  
 }  
 }  
 end\_time = omp\_get\_wtime();  
 end\_time = end\_time - st\_time;  
 full\_time += end\_time;  
 }  
 sum = sum / (12 \* Q);  
 printf(**"\n\nCritical Q: %d"**, Q);  
 printf(**"\nTotal Sum = %10.2f"**, sum);  
 printf(**"\nTIME OF WORK IS %f "**, full\_time / 12);  
 **delete** a;  
 **return** 0;  
}

ПРИЛОЖЕНИЕ Г

Код последовательной программы

*// Подключение необходимых библиотек*#include **<stdio.h>**#include **<stdlib.h>**#include **<time.h>**#define **VECTOR\_SIZE** 8200000  
  
**int** main(**int** argc, **char** \*argv[]) {  
 **int** i, j, num\_iterations = 12, count, q = 12;  
 **double** sum, total\_time = 0;  
 **double** \*vector\_a = **new double**[**VECTOR\_SIZE**];  
 *// Инициализация вектора a* **for** (i = 0; i < **VECTOR\_SIZE**; ++i) {  
 vector\_a[i] = 1;  
 }  
  
 **double** start\_time, end\_time;  
  
 sum = 0;  
*// Последовательное суммирование* **for** (count = 0; count < num\_iterations; count++) {  
 start\_time = clock();  
 **for** (i = 0; i < **VECTOR\_SIZE**; i++) {  
 **for** (j = 0; j < q; ++j) {  
 sum += vector\_a[i];  
 }  
 }  
 end\_time = clock();  
 end\_time = end\_time - start\_time;  
 total\_time += end\_time;  
 }  
  
 sum = sum / (num\_iterations \* q);  
 printf(**"\nConsistent Q: %d"**, q);  
 printf(**"\nTotal Sum = %10.2f"**, sum);  
 printf(**"\nTIME OF WORK IS %f "**, total\_time / num\_iterations);  
  
 sum = 0;  
 total\_time = 0;  
  
 **return** 0;  
}