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 6409

Студент П.А. Маштаков

(*подпись*)

Преподаватель

В.Д. Зайцев

(*подпись*)

Самара 2023

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 | float |
| N | 6300000 |
| Количество процессов/потоков | 3, 9, 12 |
| Q | 26 |

ВВЕДЕНИЕ

Использование параллельных вычислительных систем становится все более актуальным направлением в современном мире. Это обусловлено наличием задач, требующих больших вычислительных ресурсов, для которых существующей вычислительной техники недостаточно. Одними из наиболее популярных технологий параллельного программирования являются технологии MPI и OpenMP [1].

В ходе выполнения лабораторной работы было рассчитано время работы программ, реализующих поиск суммы элементов вектора и использующих технологии MPI и OpenMP.

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 1-5 представлены скрины запуска и работы программ без параметра Q.

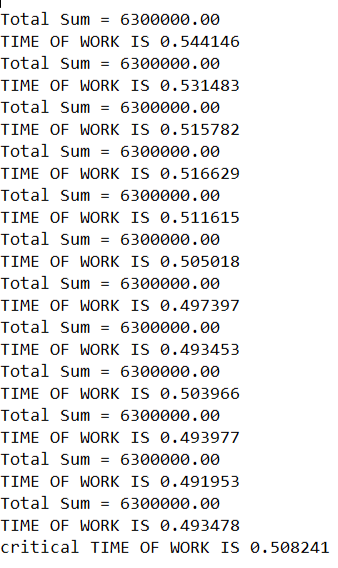


Рисунок 1 – Пример работы программы OpenMP на 3 процессах для critical

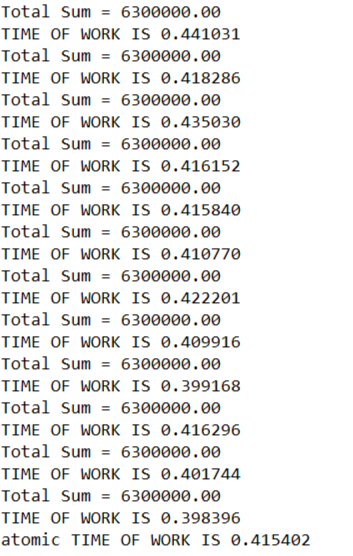


Рисунок 2 – Пример работы программы OpenMP на 3 процессах для atomic

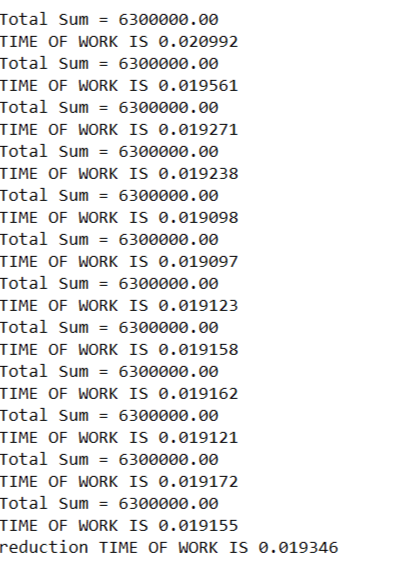


Рисунок 3 – Пример работы программы OpenMP на 3 процессах для reduction

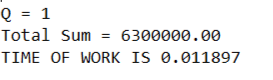


Рисунок 4 – Пример работы программы MPI на 3 процессах для операций «точка-точка»

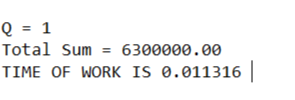


Рисунок 5 – Пример работы программы MPI на 3 процессах для коллективной операции

Последовательная программа представляла собой поиск суммы элементов вектора. Время работы последовательной программы составило 0,047500 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 | |
| critical | atomic | reduction | «точка-точка» | Коллективная |
| 3 | 0,680748 | 0,417672 | 0,019365 | 0,011873 | 0,011316 |
| 9 | 0,790728 | 0,490863 | 0,006442 | 0,003932 | 0,003813 |
| 12 | 1,348936 | 0,509947 | 0,005079 | 0,002964 | 0,002859 |

Таблица 4 – Ускорение параллельных программ без параметра Q

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во процессов, шт | Ускорение для OpenMP | | | Ускорение для MPI | |
| critical | atomic | reduction | «точка-точка» | Коллективная |
| 3 | 0,069776 | 0,113725 | 2,452879 | 4,000673 | 4,197596 |
| 9 | 0,060071 | 0,096768 | 7,373486 | 12,080366 | 12,457382 |
| 12 | 0,035212 | 0,093147 | 9,352234 | 16,025641 | 16,614200 |

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

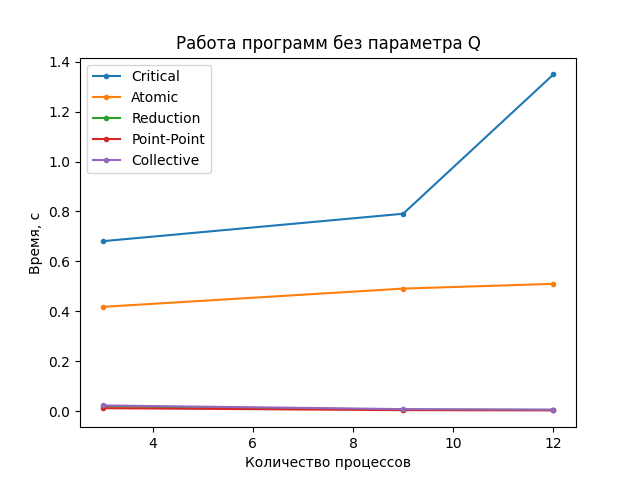


Рисунок 6 – Время работы программ без параметра Q

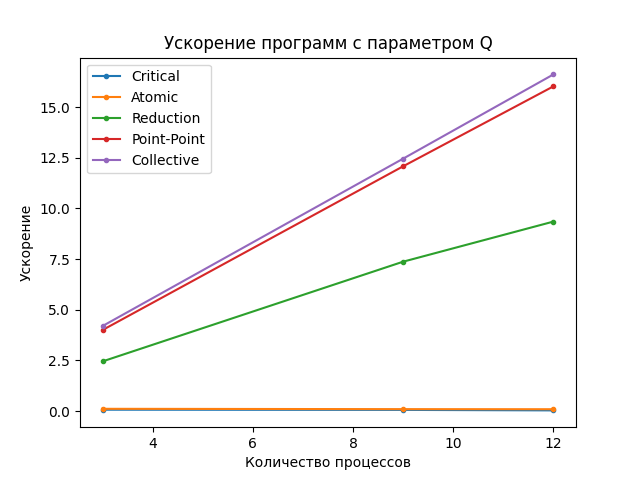


Рисунок 7 – Ускорение программ без параметра Q

2.2 Результаты работы программ с параметром Q

На рисунках 7-11 представлены скрины запуска и работы программ с параметром Q.

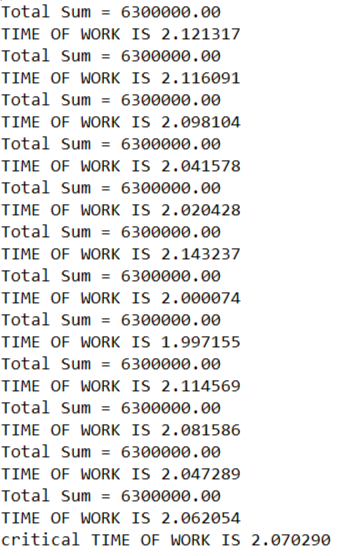


Рисунок 7 – Пример работы программы OpenMP на 3 процессах для critical

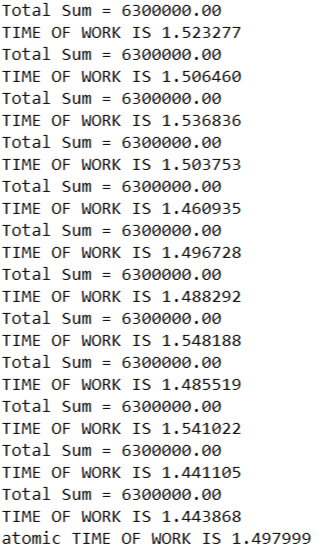


Рисунок 8 – Пример работы программы OpenMP на 3 процессах для atomic

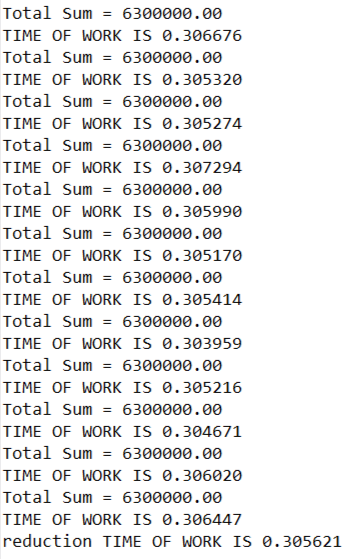


Рисунок 9 – Пример работы программы OpenMP на 3 процессах для reduction

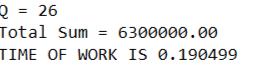


Рисунок 10 – Пример работы программы MPI на 3 процессах для операций «точка-точка»

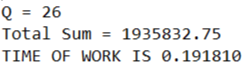


Рисунок 11 – Пример работы программы MPI на 3 процессах для коллективной операции

Усложненная последовательная программа представляла собой поиск суммы элементов вектора с параметром пересчёта Q в цикле. Время работы последовательной программы составило 0,777500 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 | |
| critical | atomic | reduction | «точка-точка» | Коллективная |
| 3 | 2,042489 | 1,657799 | 0,306437 | 0,190300 | 0,191642 |
| 9 | 2,947196 | 0,744916 | 0,101528 | 0,062777 | 0,062769 |
| 12 | 3,073283 | 1,333880 | 0,076629 | 0,047022 | 0,046961 |

Таблица 6 – Ускорение параллельных программ с параметром Q

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во процессов, шт | Ускорение для OpenMP | | | Ускорение для MPI | |
| critical | atomic | reduction | «точка-точка» | Коллективная |
| 3 | 0,380663 | 0,468995 | 2,537226 | 4,085654 | 4,057043 |
| 9 | 0,263810 | 1,043741 | 7,657985 | 12,385109 | 12,386687 |
| 12 | 0,252986 | 0,582886 | 10,14628 | 16,534813 | 16,556291 |

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

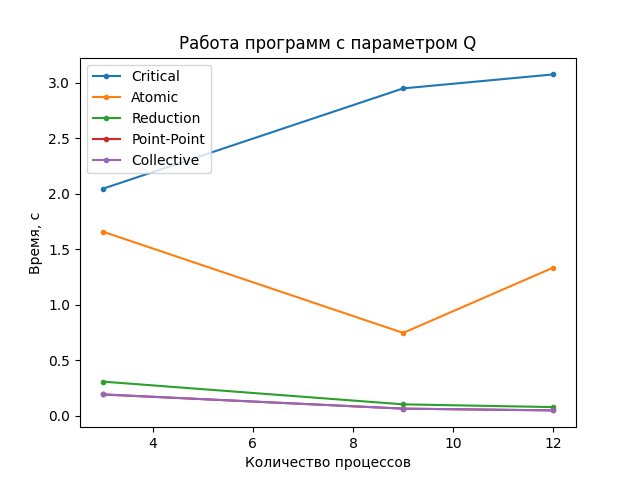


Рисунок 12 – Время работы программ с параметром Q

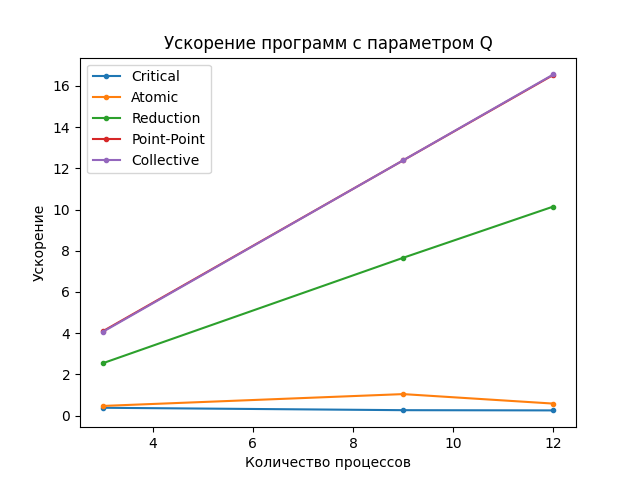


Рисунок 13 – Ускорение программ с параметром Q

**ВЫВОДЫ:**

Из полученных результатов видно, что:

1. Вариант Reduction технологии OpenMP показал лучший результат, чем Atomic и Critical. Reduction позволяет собрать вместе в главном потоке результаты вычислений частичных сумм, Atomic блокирует доступ к переменной всем запущенным в данный момент нитям, кроме нити, выполняющей операцию, Critical оформляет критическую секцию, в которой может находиться только одна нить. Оба варианта технологии MPI показали более лучшие результаты, чем OpenMP.
2. Максимальное ускорение среди программ без параметра Q и с параметром Q показал вариант коллективной MPI.
3. В случае, когда параметр Q не используется, с увеличением количества процессов увеличивается время выполнения программ с использованием вариантов Critical и Atomic технологии OpenMP, при использовании Reduction время уменьшается. При использовании параметра Q вариант Critical замедляется с увеличением количества процессов, вариант Atomic сначала замедляется, затем начинает работать быстрее, а вариант Reduction ускоряется с возрастанием количества процессов. Оба варианта MPI с увеличением количества процессов работают быстрее и показывают практически одинаковые результаты.
4. Использование параметра Q приводит к увеличению количества вычислений и, соответственно, времени выполнения программ. С увеличением параметра Q, увеличивается время выполнения программ обоих технологий.

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 необходимости обработки массива данных. Коллективные операции технологии MPI показали наилучшие результаты.

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 была организация работы на кластере.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Рыжков, Е. А. Параллельные заметки №1 – технология OpenMP / Е. А. Рыжков // Хабр : [сайт]. – Тула, 2010. – URL: https://habr.com/ru/companies/intel/articles/82486/ (дата обращения: 02.10.2023).
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. оеводин, 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] / B. II. Гергель. — М.: Интуит. 2016. - 500 с.
6. 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.
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 <math.h>

#include <stdio.h>

#include <stdlib.h>

#include "mpi.h"

int main(int argc, char\* argv[])

{

int N = 6300000;

float\* x = new float[N];

\*x = 0;

float TotalSum, ProcSum = 0.0;

int ProcRank, ProcNum, j, i;

MPI\_Status Status;

double st\_time, end\_time;

int Q = 1;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &ProcNum);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &ProcRank);

if (ProcRank == 0) {

delete[] x;

x = new float[N];

for (i = 0; i < N; i++) {

x[i] = 1.0;

}

}

float\* x\_loc = new float[N / ProcNum];

MPI\_Scatter(x, N / ProcNum, MPI\_FLOAT, x\_loc, N / ProcNum, MPI\_FLOAT, 0, MPI\_COMM\_WORLD);

st\_time = MPI\_Wtime();

for (i = 0; i < N / ProcNum; i++) {

for (j = 0; j < Q; j++) {

ProcSum = ProcSum + x\_loc[i];

}

}

ProcSum /= Q;

TotalSum = 0;

if (ProcRank == 0)

{

TotalSum = ProcSum;

for (i = 1; i < ProcNum; i++)

{

MPI\_Recv(&ProcSum, 1, MPI\_FLOAT, i, 0, MPI\_COMM\_WORLD, &Status);

TotalSum = TotalSum + ProcSum;

}

}

else

MPI\_Send(&ProcSum, 1, MPI\_FLOAT, 0, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

end\_time = MPI\_Wtime();

end\_time = end\_time - st\_time;

if (ProcRank == 0)

{

printf("\nQ = %d", Q);

printf("\nTotal Sum = %10.2f", TotalSum);

printf("\nTIME OF WORK IS %f ", end\_time);

}

MPI\_Finalize();

return 0;

}/\*#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include "mpi.h"

int main(int argc, char\* argv[])

{

int N = 6300000;

float\* x = new float[N];

\*x = 0;

float TotalSum, ProcSum = 0.0;

int ProcRank, ProcNum, j, i;

MPI\_Status Status;

double st\_time, end\_time;

int Q = 1;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &ProcNum);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &ProcRank);

if (ProcRank == 0) {

delete[] x;

x = new float[N];

for (i = 0; i < N; i++) {

x[i] = 1.0;

}

}

float\* x\_loc = new float[N / ProcNum];

MPI\_Scatter(x, N / ProcNum, MPI\_FLOAT, x\_loc, N / ProcNum, MPI\_FLOAT, 0, MPI\_COMM\_WORLD);

st\_time = MPI\_Wtime();

for (i = 0; i < N / ProcNum; i++) {

for (j = 0; j < Q; j++) {

ProcSum = ProcSum + x\_loc[i];

}

}

ProcSum /= Q;

TotalSum = 0;

MPI\_Reduce(&ProcSum, &TotalSum, 1, MPI\_FLOAT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

end\_time = MPI\_Wtime();

end\_time = end\_time - st\_time;

if (ProcRank == 0)

{

printf("\nQ = %d", Q);

printf("\nTotal Sum = %10.2f", TotalSum);

printf("\nTIME OF WORK IS %f ", end\_time);

}

MPI\_Finalize();

return 0;}\*/

ПРИЛОЖЕНИЕ Б

Код программы с технологией OpenMP

#include <stdlib.h>

#include <omp.h>

#include <stdio.h>

#define CHUNK 100

#define NMAX 6300000

#define NUM\_THREADS 3

float atomic(const int Q, float a[]) {

int i;

double sum;

double st\_time, end\_time;

st\_time = omp\_get\_wtime();

sum = 0;

#pragma omp parallel for shared(a, sum) private(i)

for (i = 0; i < NMAX; i++) {

for (int q = 0; q < Q; ++q) {

a[i] = (a[i] + a[i]) / 2;

}

#pragma omp atomic

sum += a[i];

}

end\_time = omp\_get\_wtime() - st\_time;

printf("\nTotal Sum = %10.2f", sum);

printf("\nTIME OF WORK IS %f ", end\_time);

return end\_time;

}

int main() {

omp\_set\_num\_threads(NUM\_THREADS);

float sum\_atomic = 0;

int Q = 1;

float\* a = new float[NMAX];

for (int i = 0; i < NMAX; ++i) {

a[i] = 1;

}

for (int i = 0; i < 12; ++i) {

sum\_atomic += atomic(Q, a);

}

printf("\natomic TIME OF WORK IS %f ", sum\_atomic / 12);

}

/\*#include <stdlib.h>

#include <omp.h>

#include <stdio.h>

#define CHUNK 100

#define NMAX 6300000

#define NUM\_THREADS 3

float critical(const int Q, float a[]) {

int i;

double sum;

double st\_time, end\_time;

st\_time = omp\_get\_wtime();

sum = 0;

#pragma omp parallel for shared(a, sum) private(i)

for (i = 0; i < NMAX; i++) {

for (int q = 0; q < Q; ++q) {

a[i] = (a[i] + a[i]) / 2;

}

#pragma omp critical

sum += a[i];

}

end\_time = omp\_get\_wtime();

end\_time = end\_time - st\_time;

printf("\nTotal Sum = %10.2f", sum);

printf("\nTIME OF WORK IS %f ", end\_time);

return end\_time;

}

int main() {

omp\_set\_num\_threads(NUM\_THREADS);

float sum\_critical = 0;

int Q = 1;

float\* a = new float[NMAX];

for (int i = 0; i < NMAX; ++i) {

a[i] = 1;

}

for (int i = 0; i < 12; ++i) {

sum\_critical += critical(Q, a);

}

printf("\ncritical TIME OF WORK IS %f ", sum\_critical / 12);

}

#include <stdlib.h>

#include <omp.h>

#include <stdio.h>

#define CHUNK 100

#define NMAX 6300000

#define NUM\_THREADS 3

float reduction(const int Q, float a[]) {

int i;

double sum;

double st\_time, end\_time;

st\_time = omp\_get\_wtime();

sum = 0;

#pragma omp parallel for shared(a) private(i) reduction(+: sum)

for (i = 0; i < NMAX; i++) {

for (int q = 0; q < Q; ++q) {

a[i] = (a[i] + a[i]) / 2;

}

sum += a[i];

}

end\_time = omp\_get\_wtime() - st\_time;

printf("\nTotal Sum = %10.2f", sum);

printf("\nTIME OF WORK IS %f ", end\_time);

return end\_time;

}

int main() {

omp\_set\_num\_threads(NUM\_THREADS);

float sum\_reduction = 0;

int Q = 1;

float\* a = new float[NMAX];

for (int i = 0; i < NMAX; ++i) {

a[i] = 1;

}

for (int i = 0; i < 12; ++i) {

sum\_reduction += reduction(Q, a);

}

printf("\nreduction TIME OF WORK IS %f ", sum\_reduction / 12);

}\*/

ПРИЛОЖЕНИЕ В

Код последовательной программы

#include "stdio.h"

#include <time.h>

int main(int argc, char\* argv[]) {

int Q = 1;

int i, j, N = 6300000;

float\* x = new float[N];

double sum = 0;

for (int i = 0; i < N; i++) {

x[i] = 1.0;

}

double start = clock();

for (int k = 0; k < 12; ++k) {

sum = 0;

for (i = 0; i < N; i++) {

for (j = 0; j < Q; j++) {

x[i] = (x[i] + x[i]) / 2;

}

sum += x[i];

}

}

double end = clock();

double t = (end - start) / CLOCKS\_PER\_SEC;

printf("%f\n", sum);

printf("%f", t / 12);

}