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 Н.В. Носов

(*подпись*)

Преподаватель,

к.ф.-м.н. Е.С. Козлова

(*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 | double |
| N | 3200000 |
| Количество процессов/потоков | [4, 8, 16] |
| Q | 28 |

ВВЕДЕНИЕ

Параллельные вычисления – вычисления, которые можно реализовать на многопроцессорных системах с использованием возможности одновременного выполнения многих действий, порождаемых процессом решения одной или многих задач.

Основной целью параллельных вычислений является уменьшение времени решения задачи. Многие необходимые для практики задачи необходимо решать в реальном времени или для их решения требуется очень большой объем вычислений. Но следует отметить, что увеличение процессоров не обязательно приводит к уменьшению времени решения задачи.

Таким образом, задачей параллельных вычислений является создание ресурса параллелизма в процессах решения задач и управление реализацией этого параллелизма с целью достижения наибольшей эффективности использования многопроцессорной вычислительной техники.

Получить параллельный алгоритм решения задачи можно путём распараллеливания имеющегося последовательного алгоритма или путём разработки нового параллельного алгоритма [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 1-5 представлены скрины запуска и работы программ без параметра Q.

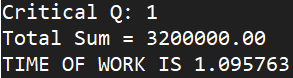


Рисунок 1 – Пример работы программы OpenMP на 4 процессах для critical

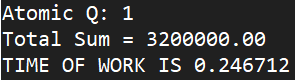


Рисунок 2 – Пример работы программы OpenMP на 4 процессах для atomic

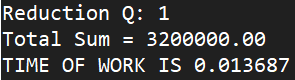


Рисунок 3 – Пример работы программы OpenMP на 4 процессах для reduction



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



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

Последовательная программа представляла собой программу подсчёта суммы элементов массива и вывод этой суммы в консоль одним процессом. Время работы последовательной программы составило 0,015833 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 |
| 4 | 1,096 | 0,247 | 0,014 | 0,013 | 0,023 |
| 8 | 1,218 | 0,171 | 0,005 | 0,016 | 0,015 |
| 16 | 1,212 | 0,251 | 0,024 | 0,014 | 0,014 |

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 | 0,01445 | 0,06410 | 1,13093 | 1,21792 | 0,68839 |
| 8 | 0,01300 | 0,09259 | 3,16660 | 0,98956 | 1,05553 |
| 16 | 0,01306 | 0,06308 | 0,65971 | 1,13093 | 1,13093 |

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.

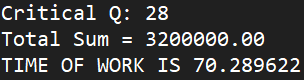


Рисунок 7 – Пример работы программы OpenMP на 4 процессах для critical

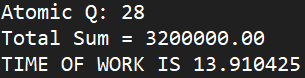


Рисунок 8 – Пример работы программы OpenMP на 4 процессах для atomic

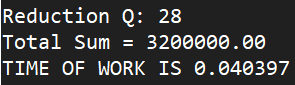


Рисунок 9 – Пример работы программы OpenMP на 4 процессах для reduction



Рисунок 10 – Пример работы программы MPI на 4 процессах для операций «точка-точка»



Рисунок 11 – Пример работы программы MPI на 4 процессах для коллективной операции

Усложненная последовательная программа представляла собой программу подсчёта суммы элементов массива и вывод этой суммы в консоль одним процессом с параметром пересчёта Q в цикле. Время работы последовательной программы с параметром Q составило 0,306667 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 |
| 4 | 70,289 | 13,910 | 0,040 | 0,0192 | 0,055 |
| 8 | 37,351 | 8,945 | 0,103 | 0,0520 | 0,050 |
| 16 | 480,141 | 18,657 | 0,044 | 0,0226 | 0,031 |

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 | 0,00436 | 0,02205 | 7,66667 | 15,97224 | 5,57576 |
| 8 | 0,00821 | 0,03428 | 2,97735 | 5,89744 | 6,13334 |
| 16 | 0,00064 | 0,01644 | 6,96970 | 13,56934 | 9,89248 |

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. Для программ с использованием технологии MPI время практически одинаковое, параллельные программы с параметром Q, при использовании этой технологии, выполняются быстрее, чем с технологией OpenMP примерно в 2 раза. При использовании технологии OpenMP, reduction показал лучший результат. Самым медленным в использовании оказался critical. Технология OpenMP тратит значительное время на создание нитей и синхронизацию с использованием соответствующих директив, следовательно, мы можем получить низкое ускорение или даже замедление.
2. Для технологии MPI максимальное ускорение без параметра Q получено для 4 процессов практически одинаковое для коллективной операции и «точка-точка» и равно примерно 1,22. С параметром Q был получен идентичный результат, максимальное ускорение равно примерно 16. Это можно объяснить архитектурой кластера: 8 процессоров находятся на одной материнской плате, а другие на другой, и их взаимодействие осуществляется через высокоскоростную коммуникационную сеть, которая все же медленнее оперативной памяти. Максимальное ускорение для OpenMP получено при использовании reduction на 8 процессах и равно примерно 3 для программы без Q и 7,7 для программы с параметром Q, а максимальное замедление при использовании critical и равно 0,013 на 8 процессах без параметра Q и 0,00064 на 16 процессах с параметром Q. Это можно объяснить тем, что reduction позволяет выполнять операции на процессах в одно время (то есть максимально параллельно), локализуя при этом данные (создание локальной копии для общей переменной суммы) для каждого потока, в то время как critical блокирует все потоки, то есть работать с данными может только один процесс, по факту программа становится последовательной. Использование atomic позволяет избежать гонки за данными, но все равно создает очередь процессов для обращений к общей переменной.
3. Без параметра и с параметром Q, при увеличении числа процессов, время и ускорение увеличиваются. Однако для технологии MPI, при переходе от 8 процессов к 16 время работы программы резко уменьшается, это объясняется в предыдущем пункте. Время увеличивается некратно из-за того, что ресурсы тратятся на создание среды и потоков.
4. Ускорение параллельных программ, использующих технологии MPI с параметром Q, превышает ускорение программ без параметра Q. Для программ с использованием технологии OpenMP с параметром Q ускорение ниже, чем без параметра Q, а с использованием atomic ускорение незначительно уменьшается, что можно объяснить погрешностью измерений. Для всех вариантов программы с использованием технологии OpenMP, кроме reduction, с увеличением числа процессов, ускорение, как правило, уменьшается, а если точнее, то происходит замедление. Для опции reduction без использования параметра Q, ускорение при 4 нитях равно 1,13, при 8 нитях равно 3,16, на 16 – 0,66; с параметром Q на 4 нитях ускорение равно 7,7, на 8 – 2,97, на 16 – 6,9). Операции MPI с параметром Q, усложняющим работу программ, дают наилучшее ускорение.

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 технология коллективных операций MPI. Это можно объяснить тем, что коллективные операции в MPI работают быстрее операций «точка-точка», которые используют блокировки.

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 стала реализация программ с использованием технологии MPI, а также многократный запуск программ на кластере. Интерес вызвали результаты вычислительных экспериментов, а именно зависимость ускорения от количества процессов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гафаров, Ф.М. Параллельные вычисления: учеб. Пособие / Ф.М Гафаров, А.Ф Галимянов. – Казань: Изд-во Казан. ун-та, 2018. – 149c.
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] / 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 (point-to-point)

#include <cstdio>

#include <cstdlib>

#include "mpi.h"

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

{

double\* b, TotalSum, ProcSum = 0.0;

double\* a;

int ProcRank, ProcNum, N = 3200000, i, j, Q = 28, 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;

}

}

st\_time = MPI\_Wtime();

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);

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 программы с технологией MPI (collective)

#include <cstdio>

#include <cstdlib>

#include "mpi.h"

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

double\* b, TotalSum, ProcSum = 0.0;

double\* a;

int ProcRank, ProcNum, N = 3200000, i, j, Q = 28, 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;

}

}

st\_time = MPI\_Wtime();

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);

}

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 программы с технологией OpenMP

#include <math.h>

#include <omp.h>

#include <time.h>

#include <stdlib.h>

#include <locale.h>

#include <stdio.h>

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

omp\_set\_num\_threads(4);

int N = 3200000;

int i, j, Q = 28;

double sum = 0, total\_time = 0;

double\* a = new double[N];

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

a[i] = 1;

}

double st\_time, end\_time;

//reduction

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

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

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

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

sum += a[i];

}

}

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

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

total\_time = end\_time;

sum = sum / Q;

printf("\n\nReduction Q: %d", Q);

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

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

sum = 0;

total\_time = 0;

//atomic

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

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

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

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

#pragma omp atomic

sum += a[i];

}

}

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

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

total\_time = end\_time;

sum = sum / Q;

printf("\n\nAtomic Q: %d", Q);

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

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

sum = 0;

total\_time = 0;

//critical

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

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

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

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

#pragma omp critical

sum += a[i];

}

}

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

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

total\_time = end\_time;

sum = sum / Q;

printf("\n\nCritical Q: %d", Q);

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

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

delete a;

system("pause");

return 0;

}

ПРИЛОЖЕНИЕ Г

Код последовательной программы

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define N\_SIZE 3200000

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

int i, j, num\_iterations = 12, count, Q = 28;

double sum, total\_time = 0;

double\* a = new double[N\_SIZE];

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

a[i] = 1;

}

clock\_t start\_time, end\_time;

sum = 0;

// Последовательное суммирование

for (count = 0; count < num\_iterations; count++) {

start\_time = clock();

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

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

sum += 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;

}