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 Н.В. Носов

(*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 | double |
| N | 3200000(+13) |
| Количество процессов/потоков | [4, 8, 16] |
| Q | 28 |

ВВЕДЕНИЕ

Одним из важных методов программирования является параллельное программирование. Оно позволяет разделить процесс решения целой задачи на несколько отдельных подзадач, которые могут выполняться одновременно на нескольких ядрах компьютера. Этот подход считается эффективным в сокращении времени работы программы [1].

В данной работе применяются две технологии параллельного программирования: MPI и OpenMP. Они используются для распараллеливания программ. Отличие между ними заключается в том, что первая работает с распределенной памятью, а вторая - с общей.

В данной лабораторной работе представлены результаты вычисления времени выполнения программы по суммированию векторов с использованием технологий MPI и OpenMP на различном количестве потоков.

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.

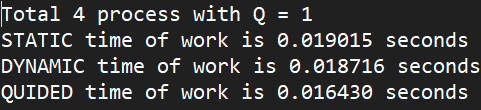


Рисунок 1 – Пример работы программы OpenMP на 4 процессах для static

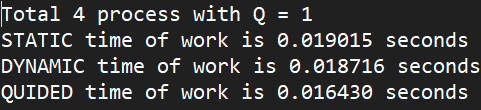


Рисунок 2 – Пример работы программы OpenMP на 4 процессах для dynamic

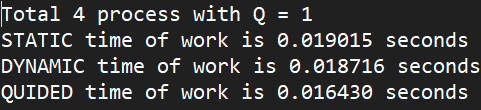


Рисунок 3 – Пример работы программы OpenMP на 4 процессах для guided



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



Рисунок 5 – Пример работы программы MPI на 4 процессах для некратных размерностей

Последовательная программа представляла собой программу вычисления суммы двух массивов одним процессом. Время работы последовательной программы составило 0,018333 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 |
| 4 | 0,019015 | 0,018716 | 0,016430 | 0,012380 | 0,011107 |
| 8 | 0,018876 | 0,017985 | 0,017141 | 0,010465 | 0,013185 |
| 16 | 0,023712 | 0,021652 | 0,021080 | 0,009126 | 0,009131 |

Таблица 4 – Ускорение параллельных программ без параметра Q

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во процессов, шт | Ускорение для OpenMP | | | Ускорение для MPI | |
| static | dynamic | guided | кратное | некратное |
| 4 | 0,964134 | 0,979536 | 1,115825 | 1,480856 | 1,650581 |
| 8 | 0,971233 | 1,019349 | 1,069541 | 1,751839 | 1,390444 |
| 16 | 0,773153 | 0,846712 | 0,869687 | 2,008876 | 2,007776 |

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

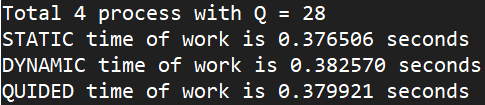


Рисунок 7 – Пример работы программы OpenMP на 4 процессах для static

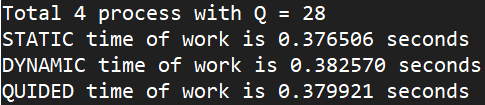


Рисунок 8 – Пример работы программы OpenMP на 4 процессах для dynamic

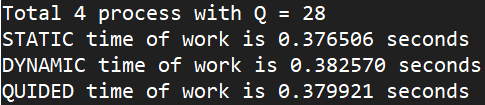


Рисунок 9 – Пример работы программы OpenMP на 4 процессах для guided



Рисунок 10 – Пример работы программы MPI на 4 процессах для кратных размерностей



Рисунок 11 – Пример работы программы MPI на 4 процессах для некратных размерностей

Усложненная последовательная программа представляла собой программу вычисления суммы двух массивов одним процессом. Время работы последовательной программы составило 0,431667 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 |
| 4 | 0,376506 | 0,382570 | 0,379921 | 0,011448 | 0,036194 |
| 8 | 0,363475 | 0,361968 | 0,356578 | 0,010786 | 0,025096 |
| 16 | 0,385991 | 0,378026 | 0,357636 | 0,009224 | 0,015639 |

Таблица 6 – Ускорение параллельных программ с параметром Q

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во процессов, шт | Ускорение для OpenMP | | | Ускорение для MPI | |
| static | dynamic | guided | кратное | некратное |
| 4 | 1,146508 | 1,128335 | 1,136202 | 37,706760 | 11,926480 |
| 8 | 1,187611 | 1,192556 | 1,210582 | 40,021050 | 17,200630 |
| 16 | 1,118334 | 1,141898 | 1,207001 | 46,798240 | 27,601960 |

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 программы с кратной размерностью показывают лучшее время. При использовании технологии OpenMP, guided показал лучший результат. Самыми медленными оказались dynamic и static. Это можно объяснить тем, что при использовании dynamic некоторые процессы могут получить последнюю порцию вычислений в тот момент, когда остальные процессы уже заканчивают свои вычисления. Это вызывает простой остальных процессов и, как следствие, замедляет выполнение программы.
2. Для технологии MPI максимальное ускорение без параметра Q получено при 16 процессах и равно 2,00888. Для программ с параметром Q максимальное ускорение получено при 16 процессах для технологии MPI и оно равно примерно 47. Для OpenMP максимальное ускорение без параметра Q получено при 4 процессах и равно 1,11582 для guided. Максимальное ускорение с параметром Q получено для guided при 8 процессах и равно 1,21058. Это объясняется тем, что при использовании guided нагрузка по процессам распределяется наиболее рациональным способом (количество передаваемых данных уменьшается со временем). Тем самым ресурсы процессоров используются по максимуму.
3. Без параметра и с параметром Q для всех технологий при переходе с 4 процессов на 8 процессов уменьшается время работы (ускорение увеличивается). При переходе с 8 процессов на 16 процессов OpenMP увеличивает свое время работы. Это объясняется спецификой суперкомпьютера, ведь на каждой материнской плате стоят по 8 процессоров, поэтому при переходе с 8 на 16 процессов материнские платы начинают взаимодействовать друг с другом по коммутационной шине, что замедляет время выполнения. При переходе с 8 процессов на 16 процессов время работы MPI уменьшается для всех его модификаций. Это можно объяснить тем, что технология MPI обычно используется для распределенных систем с большим количеством процессоров или компьютеров, где каждый процесс работает на отдельной машине. OpenMP обычно используется для распараллеливания вычислений на нескольких ядрах в пределах одной машины. Также в MPI каждый процесс имеет собственную память, поэтому это позволяет избежать различных блокировок между процессами.
4. Ускорение параллельных программ с параметром Q превышает ускорение программ без параметра Q. Для вариантов программы с использованием технологии OpenMP без параметра Q с увеличением числа процессов, ускорение в среднем уменьшается. Это можно объяснить тем, что при малых объемах вычислений использовать большое количество потоков не рационально, так как время в основном будет тратиться на создание этих потоков, а не на вычисления. В среднем для dynamic распределения итераций наблюдается рост ускорения с увеличением числа процессов (без использования параметра Q незначительно, при 4 процессах – 0,97954, при 8 нитях равно 1,01935, на 16 – 0,84671; с параметром Q на 4 нитях ускорение равно 1,12833, на 8 –1,19256, на 16 – 1,1419). При static распределении итераций с параметром Q получается наилучшее ускорение для OpenMP (на 4 нитях ускорение равно 1,14651, на 8 – 1,18761, на 16 – 1,11833). Операции MPI с параметром Q, усложняющим работу программ, дают наилучшее ускорение.

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 в общем случае время выполнения программ с увеличением числа процессов уменьшается и повышается ускорение. Лучше всего себя показала технология MPI с некратным числом элементов. Это можно объяснить тем, что в MPI не требуется ресурсы на создание среды и потоков, как это необходимо в OpenMP.

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 был сбор данных на кластере для каждой программы. Интерес вызвало изучение типов распределений итераций между процессами для написания параллельных программ.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Параллельное программирование: определение, преимущества и применение в отрасли: [сайт] // BUOM: Работа и карьера / BUOM. – 2021. – URL: https://buom.ru/parallelnoe-programmirovanie-opredelenie-preimushhestva-i-primenenie-v-otrasli (дата обращения: 31.10.2023).
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, когда количество элементов массива кратно количеству процессов

#include "mpi.h"

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <ctime>

#define NMAX 3200000

#define Q 28

#define ITERATIONS 12

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

{

srand(time(0));

double a[NMAX], b[NMAX], c[NMAX];

double \*a\_loc = NULL, \*b\_loc = NULL, \*c\_loc = NULL;

int ProcRank, ProcNum, N = NMAX, i, j, s;

MPI\_Status status;

double st\_time, average\_time = 0.0;

MPI\_Init(&argc, &argv);

for (s = 0; s < ITERATIONS; ++s)

{

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

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

int count = N / ProcNum;

if (ProcRank == 0)

{

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

{

a[i] = i;

b[i] = i;

}

}

count = NMAX / ProcNum;

a\_loc = (double \*)malloc(count \* sizeof(double));

b\_loc = (double \*)malloc(count \* sizeof(double));

c\_loc = (double \*)malloc(count \* sizeof(double));

MPI\_Scatter(a, count, MPI\_DOUBLE, a\_loc, count, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(b, count, MPI\_DOUBLE, b\_loc, count, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

if (ProcRank == 0)

{

st\_time = MPI\_Wtime();

}

for (i = 0; i < count; ++i)

{

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

{

c\_loc[i] = a\_loc[i] + b\_loc[i];

}

}

MPI\_Gather(c\_loc, count, MPI\_DOUBLE, c, count, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

if (ProcRank == 0)

average\_time += MPI\_Wtime() - st\_time;

free(a\_loc);

free(b\_loc);

free(c\_loc);

}

if (ProcRank == 0) {

printf("Total process = %d", ProcNum);

printf(" with Q = %d", Q);

printf("\nMPI time of work is %f\n", average\_time / ITERATIONS);

}

MPI\_Finalize();

return 0;

}

ПРИЛОЖЕНИЕ Б

Код программы с технологией MPI, когда количество элементов массива не кратно количеству процессов

#include "mpi.h"

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <ctime>

#define NMAX 3200013

#define Q 28

#define ITERATIONS 12

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

{

srand(time(0));

double a[NMAX], b[NMAX], c[NMAX], s = 0, j;

double \*a\_loc = NULL, \*b\_loc = NULL, \*c\_loc = NULL;

int ProcRank, ProcNum, i;

int \*sendCounts;

int \*displs;

double st\_time, average\_time = 0.0;

MPI\_Status status;

MPI\_Init(&argc, &argv);

for (s = 0; s < ITERATIONS; ++s)

{

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

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

int count = NMAX / ProcNum;

if (ProcRank == 0)

{

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

{

a[i] = i;

b[i] = i;

}

}

sendCounts = (int \*)malloc(ProcNum \* sizeof(int));

displs = (int \*)malloc(ProcNum \* sizeof(int));

for (i = 0; i < ProcNum; ++i)

{

sendCounts[i] = count;

}

for (i = 0; i < NMAX % ProcNum; ++i)

{

sendCounts[i] += 1;

}

displs[0] = 0; // массив смещений

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

{

displs[i] = displs[i - 1] + sendCounts[i - 1];

}

count = sendCounts[ProcRank];

a\_loc = (double \*)malloc(count \* sizeof(double));

b\_loc = (double \*)malloc(count \* sizeof(double));

c\_loc = (double \*)malloc(count \* sizeof(double));

MPI\_Scatterv(a, sendCounts, displs, MPI\_DOUBLE, a\_loc, count, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatterv(b, sendCounts, displs, MPI\_DOUBLE, b\_loc, count, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

if (ProcRank == 0)

{

st\_time = MPI\_Wtime();

}

for (i = 0; i < sendCounts[ProcRank]; ++i)

{

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

{

c\_loc[i] = a\_loc[i] + b\_loc[i];

}

}

MPI\_Gatherv(c\_loc, sendCounts[ProcRank], MPI\_DOUBLE, c, sendCounts, displs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

if (ProcRank == 0)

average\_time += MPI\_Wtime() - st\_time;

free(a\_loc);

free(b\_loc);

free(c\_loc);

free(sendCounts);

free(displs);

}

if (ProcRank == 0) {

printf("Total process = %d", ProcNum);

printf(" with Q = %d", Q);

printf("\nMPI with Scatterv and Gatherv time of work is %f\n", average\_time / ITERATIONS);

}

MPI\_Finalize();

return 0;

}

ПРИЛОЖЕНИЕ В

Код программы с технологией OpenMP

#include <omp.h>

#include <cstdlib>

#include <time.h>

#include "stdio.h"

#define CHUNK 100

#define NMAX 3200000

#define Q 28

#define NUM\_OF\_THREADS 4 // 4, 8, 16

#define ITERATIONS 12

int main() {

double a[NMAX], b[NMAX], sum[NMAX];

int i, step, t, j;

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

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

a[i] = i;

b[i] = i;

}

double st\_time, end\_time, time\_static = 0.0, time\_dynamic = 0.0, time\_quided = 0.0;

for (t = 0; t < ITERATIONS; t++) {

for (step = 0; step < 3; step++) {

srand(time(0));

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

#pragma omp parallel for shared (a,b,sum) private(i, j) schedule(static, CHUNK) if (step == 0)

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

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

sum[i] = a[i] + b[i];

}

}

#pragma omp parallel for shared (a,b,sum) private(i, j) schedule(dynamic, CHUNK) if (step == 1)

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

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

sum[i] = a[i] + b[i];

}

}

#pragma omp parallel for shared (a,b,sum) private(i, j) schedule(guided, CHUNK) if (step == 2)

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

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

sum[i] = a[i] + b[i];

}

}

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

if (step == 0)

time\_static += end\_time - st\_time;

else if (step == 1)

time\_dynamic += end\_time - st\_time;

else if (step == 2)

time\_quided += end\_time - st\_time;

}

}

printf("Total %d process", NUM\_OF\_THREADS);

printf(" with Q = %d", Q);

printf("\nSTATIC time of work is %f seconds", time\_static / ITERATIONS);

printf("\nDYNAMIC time of work is %f seconds", time\_dynamic / ITERATIONS);

printf("\nQUIDED time of work is %f seconds", time\_quided / ITERATIONS);

return 0;

}

ПРИЛОЖЕНИЕ Г

Код последовательной программы

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define N\_SIZE 3200000

#define ITERATION 12

#define Q 28

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

int i, j, count;

double sum, total\_time = 0;

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

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

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

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

a[i] = 1;

b[i] = 1;

}

clock\_t start\_time, end\_time;

for (count = 0; count < ITERATION; count++) {

start\_time = clock();

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

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

c[i] = a[i] + b[i];

}

}

end\_time = clock();

end\_time = end\_time - start\_time;

total\_time += end\_time;

}

sum = sum / (ITERATION \* Q);

printf("\nConsistent Q: %d", Q);

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

printf("\nTIME OF WORK IS %f ", total\_time / CLOCKS\_PER\_SEC / ITERATION);

return 0;

}

ПРИЛОЖЕНИЕ Д

Скрипт запуска программы для MPI

#!/bin/bash

#SBATCH --job-name=myjob

#SBATCH --time=00:10:00

#SBATCH --nodes=1 --ntasks-per-node=4

#SBATCH --mem=1gb

export I\_MPI\_LIBRARY=/usr/lib64/slurm/mpi\_pmi2.so

srun --mpi=pmi2 ./mpi

ПРИЛОЖЕНИЕ Е

Скрипт запуска программы для OpenMP

#!/bin/bash

#SBATCH --job-name=myjob

#SBATCH --time=00:10:00

#SBATCH --nodes=1 --ntasks-per-node=4

#SBATCH --mem=1gb

./openmp