Практическая работа № A-07

Методы высокопроизводительных вычислений

Альшаеб Басель, группа 24.M71-mm

04.12.2024

1- Задание

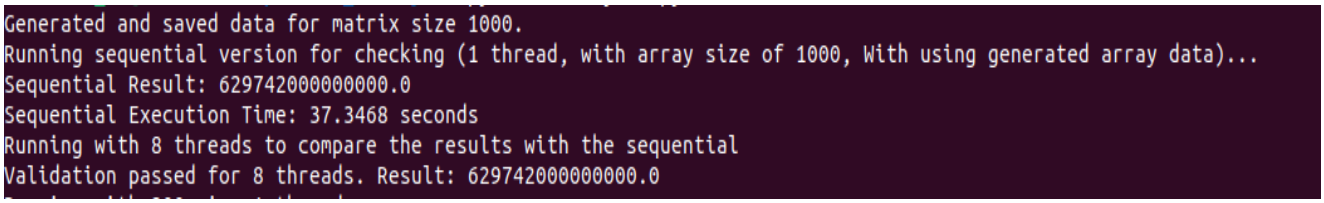
Написать программу вычисления выражения:
d = <B4 x, y>/<x, y> - <B3 x, y>/<x, y>,
где B – квадратная плотная матрица, элементы которой имеют тип double, элементы матрицы
задаются с помощью генератора псевдослучайных чисел, x и y – векторы (элементы задаются
псевдослучайными числами), <,> - скалярное произведение.
Распараллелить эту программу с помощью OpenMP (parallel, task).
Исследовать зависимость масштабируемости параллельной версии программы от ее
вычислительной трудоемкости (размера матриц).
Проверить корректность параллельной версии.
Проверка закона Амдала. Построить зависимость ускорение:число потоков для заданного примера.

2- Описание программно-аппаратной конфигурации

OS: Ubuntu 22.04.4 LTS
Kernel: 5.15.0-122-generic
Processer: Intel® Xeon® E-2136 CPU @ 3.30GHz
Cores: 12
Architecture: x86_64
Memory: 62 G
Compiler g++ 10.2.0

3- Проверка правильности параллельного решения

Для проверки исправления я сгенерировал фиксированное количество данных для размера массива 1000, а затем запустил программу для 1 потока и 8 потоков и сравнил результат. Я приложу изображения для сравнения результата из логов.



4- Исследование зависимости ускорения от количества потоков и размера массива

Параллельный раздел посвящен распределению скалярного произведения и каждой операции умножения, и это будет распределено между потоками.
Каждый раз, когда мы добавляем больше потоков, мы можем распределить операции по разным потокам. В результате это сделает программу быстрее до определенной точки, когда размер массива будет соответствовать количеству потоков.

Эксперимент проводится на трех размерах массива: [200, 500, 1000] и потоках [1, 2, 4, 8, 16, 32, 64]

Array Size: 200x200

Threads	Execution Time	Speedup
1	0.264 Sec	1.0
2	0.132 Sec	2.0
4	0.068 Sec	3.88
8	0.067 Sec	3.92
16	0.065 Sec	4.03
32	0.055 Sec	4.75
64	0.058 Sec	4.56

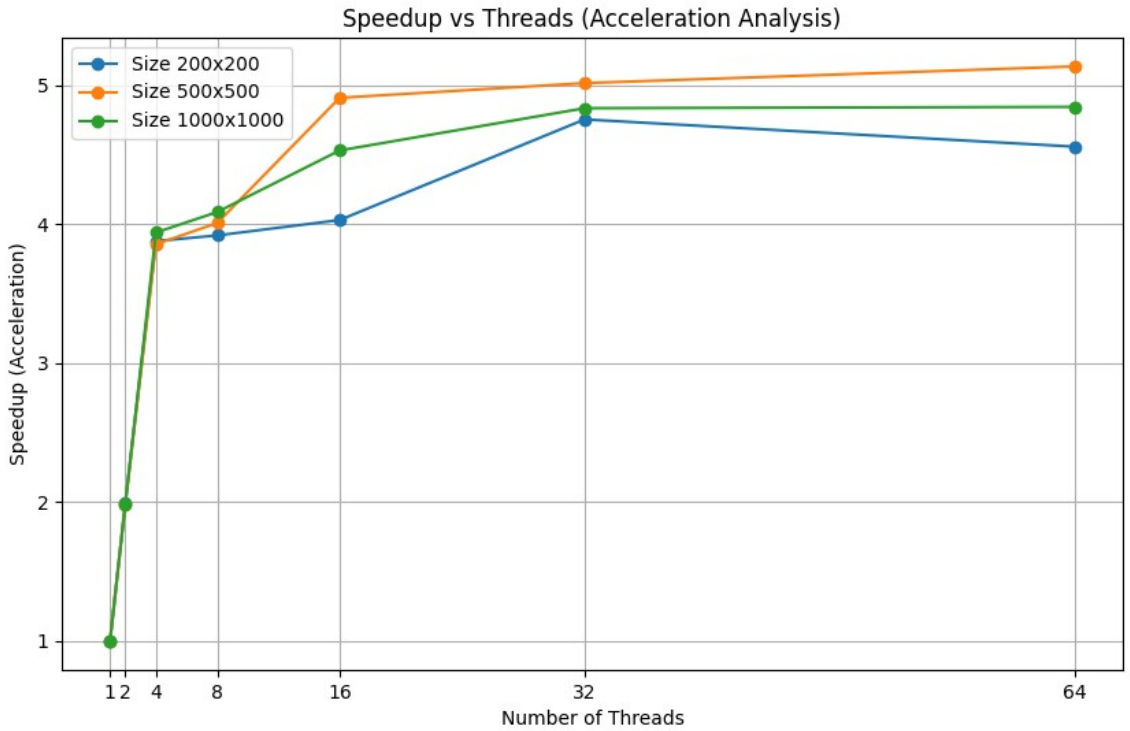
Array Size: 500

Threads	Execution Time	Speedup
1	4.438 Sec	1.0
2	2.232 Sec	1.99
4	1.151 Sec	3.85
8	1.107 Sec	4.01
16	0.904 Sec	4.91
32	0.885 Sec	5.01
64	0.864 Sec	5.13

Array Size: 1000

Threads	Execution Time	Speedup
1	37.5 Sec	1.0
2	18.930 Sec	1.98
4	9.539 Sec	3.94
8	9.195 Sec	4.09
16	8.296 Sec	4.53
32	7.774 Sec	4.83
64	7.759 Sec	4.84

Ниже вы можете увидеть график результатов по размеру, ускорению и потокам.



5- Объяснение результатов и выводы

Из графика мы видим, что ускорение увеличивается с увеличением потоков, а размер массива также играет роль. При увеличении размера ускорение также меняется в зависимости от размера, потому что это квадратный массив.

- Из этого графика можно сказать, что при достижении 5-кратного ускорения ускорение останавливается и начинает сглаживаться

- Во всех случаях мы видим, что ускорение стабилизируется на 64 потоках.

6- Исходный код

- Определение основных функций

```
ain.cpp > matrix_multiply(const vector<vector<double>>&, const vector<vector<double>>&, vector<vector<double>>&, int)
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <omp.h>
#include <fstream>

using namespace std;

void read_matrix(const string &filename, vector<vector<double>> &matrix, int n);
void read_vector(const string &filename, vector<double> &vec, int n);
void generate_matrix(vector<vector<double>> &matrix, int n);
void generate_vector(vector<double> &vec, int n);
double scalar_product(const vector<double> &x, const vector<double> &y, int n);
void matrix_multiply(const vector<vector<double>> &A, const vector<vector<double>> &B, vector<vector<double>> &C, int n);
```

- определение основных переменных и массивов. и добавление параметров, которые помогут нам провести эксперимент. Итак, у нас есть "use-files", который говорит нам использовать предопределенный массив, "threads" для указания количества потоков, "size" для установки размера массива.

```
int main(int argc, char *argv[])
{
    int n = 1000, threads = 1;
    int i, j;
    bool use_file = false;

    vector<vector<double>> B(n, vector<double>(n));
    vector<vector<double>> B3(n, vector<double>(n));
    vector<vector<double>> B4(n, vector<double>(n));
    vector<double> x(n);
    vector<double> y(n);

    for (int i = 1; i < argc; i++)
    {
        string arg = argv[i];
        if (arg == "--size" && i + 1 < argc)
        {
            n = stoi(argv[++i]);
        }
        else if (arg == "--threads" && i + 1 < argc)
        {
            threads = stoi(argv[++i]);
        } else if (arg == "--use-files") {
            use_file = true;
        }
    }

    omp_set_num_threads(threads);

    if (use_file)
    {
        cout << "Reading data from files..." << endl;
        read_matrix("matrix_B.txt", B, n);
        read_vector("vector_x.txt", x, n);
        read_vector("vector_y.txt", y, n);
    }
    else
    {
        cout << "Generating random data..." << endl;
        srand(time(NULL));
        generate_matrix(B, n);
        generate_vector(x, n);
        generate_vector(y, n);
    }
```

- Основная логическая область, где: 1- мы выполняем умножение массивов для создания B^2 и других, 2- мы вычисляем скалярное произведение. Для вычисления умножения векторов и массивов мы использовали «pragma omp parallel for», а для скаляра между вектором и массивом мы использовали «omp sections»

```
    double start_time = omp_get_wtime();

    vector<vector<double>> B2(n, vector<double>(n));
    matrix_multiply(B, B, B2, n);
    matrix_multiply(B, B2, B3, n);
    matrix_multiply(B2, B2, B4, n);

    double scalar_xy, scalar_B4xy, scalar_B3xy;

    scalar_xy = scalar_product(x, y, n);

    // Compute B^4 * x and B^3 * x
    vector<double> B4x(n, 0.0), B3x(n, 0.0);
#pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            B4x[i] += B4[i][j] * x[j];
            B3x[i] += B3[i][j] * x[j];
        }
    }
#pragma omp parallel sections
    {
#pragma omp section
        scalar_B4xy = scalar_product(B4x, y, n);

#pragma omp section
        scalar_B3xy = scalar_product(B3x, y, n);
    }

    double d = (scalar_B4xy / scalar_xy) - (scalar_B3xy / scalar_xy);

    double end_time = omp_get_wtime();
```

- Печать результата

```
    double end_time = omp_get_wtime();

    cout << "Result (d): " << d << endl;
    cout << "Execution Time: " << (end_time - start_time) << endl;

    return 0;
}
```

- Сгенерировать и заполнить массив и вектор случайными данными

```
// Function to generate a random matrix
void generate_matrix(vector<vector<double>> &matrix, int n)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            matrix[i][j] = (double)rand() / RAND_MAX * 10.0;
}

void generate_vector(vector<double> &vec, int n)
{
    for (int i = 0; i < n; i++)
        vec[i] = (double)rand() / RAND_MAX * 10.0;
}
```

- Скалярное произведение двух векторных функций Я использовал «parallel for reduction (+ : result)»

```
}

double scalar_product(const vector<double> &x, const vector<double> &y, int n)
{
    double result = 0.0;

#pragma omp parallel for reduction(+ : result)
    for (int i = 0; i < n; i++)
        result += x[i] * y[i];

    return result;
}
```

- Функцию умножения матриц я использовал «omp parallel for»

```
void matrix_multiply(const vector<vector<double>> &A, const vector<vector<double>> &B, vector<vector<double>> &C, int n)
{
#pragma omp parallel for
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
        {
            C[i][j] = 0.0;
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}
```

- Функции чтения из файлов

```cpp
void read_matrix(const string &filename, vector<vector<double>> &matrix, int n)
{
    ifstream file(filename);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            file >> matrix[i][j];
    file.close();
}

void read_vector(const string &filename, vector<double> &vec, int n)
{
    ifstream file(filename);
    for (int i = 0; i < n; ++i)
        file >> vec[i];
    file.close();
}
```

- Код Python, в котором мы определяем все тесты и создаем статические файлы для проверки программы, а также генерируем график для сравнения.

```python
import matplotlib.pyplot as plt
import os
import subprocess
import numpy as np
import math
from prettytable import PrettyTable

matrix_size = 1000
thread_counts = [1, 2, 4, 8, 16, 32, 64]
array_sizes = [200, 500, 1000]
program = "./main"
results = []
tolerance = 1e-6

# Function to generate and save deterministic matrix and vector data
def generate_and_save_data(size):
    np.random.seed(42)
    B = np.random.rand(size, size) * 10
    x = np.random.rand(size) * 10
    y = np.random.rand(size) * 10

    np.savetxt("matrix_B.txt", B, fmt="%.6f")
    np.savetxt("vector_x.txt", x, fmt="%.6f")
    np.savetxt("vector_y.txt", y, fmt="%.6f")
    print(f"Generated and saved data for matrix size {size}.")

def run_program(threads, size, use_files=False):
    command = [program, "--threads", str(threads), "--size", str(size)]
    if use_files:
        command.append("--use-files")
    result = subprocess.run(command, capture_output=True, text=True)
    execution_time = None
    computed_result = None

    # Parse the output
    for line in result.stdout.split("\n"):
        if "Execution Time" in line:
            execution_time = float(line.split(":")[1].strip())
        if "Result (d)" in line:
            computed_result = float(line.split(":")[1].strip())

    return execution_time, computed_result

# 1- prepare verification step
generate_and_save_data(matrix_size)

print("Running sequential version for checking (1 thread, with array size of 1000, With using generated array data)...")
seq_time, seq_result = run_program(1, use_files=True, size=1000)
if seq_time is None or seq_result is None:
    print("Failed to retrieve results for the sequential run.")
    exit()
```

(parameter) use_files: bool

```
print(f"Sequential Result: {seq_result}")
print(f"Sequential Execution Time: {seq_time} seconds")

print(f"Running with 8 threads to compare the results with the sequential")
par_time, par_result = run_program(8, use_files=True, size=1000)
if par_time is None or par_result is None:
    print(f"Failed to retrieve results for 8 threads.")
elif math.isclose(seq_result, par_result, rel_tol=tolerance):
    print(f"Validation passed for 8 threads. Result: {par_result}")
else:
    print(f"Validation failed for 8 threads! Sequential: {seq_result}, Parallel: {par_result}")


execution_times = {size: {} for size in array_sizes}  # Store execution times by size and thread count

# 2- running all threads
for size in array_sizes:
    for threads in thread_counts:
        print(f"Running with {size} size {threads} threads...")
        execution_time, computed_result = run_program(threads, use_files=False, size=size)

        print(execution_time)
        if execution_time is not None:
                execution_times[size][threads] = execution_time
        else:
            print(f"Failed to extract execution time for {threads} threads.")


# Calculate speedups
speedups = {size: {} for size in array_sizes}
for size in array_sizes:
    sequential_time = execution_times[size][1]  # Time for 1 thread as the baseline
    for threads in thread_counts:
        speedups[size][threads] = sequential_time / execution_times[size][threads]

# Print speedup tables to console
for size in array_sizes:
    table = PrettyTable()
    table.field_names = ["Threads", "Speedup"]
    for threads in thread_counts:
        table.add_row([threads, round(speedups[size][threads], 2)])
    print(f"\nSpeedup Table for Matrix Size {size}x{size}")
    print(table)


plt.figure(figsize=(10, 6))
for size in array_sizes:
    speedup_values = [speedups[size][threads] for threads in thread_counts]
    plt.plot(thread_counts, speedup_values, marker="o", label=f"Size {size}x{size}")
```

```
plt.title("Speedup vs Threads (Acceleration Analysis)")
plt.xlabel("Number of Threads")
plt.ylabel("Speedup (Acceleration)")
plt.xticks(thread_counts)
plt.grid()
plt.legend()
plt.savefig("speedup_vs_threads_analysis.png")
plt.show()
```