ИИКС

Лабораторная работа №6: «Коллективные операции в MPI»

Грущин Илья

Б21-515

2023 г.

Описание используемой рабочей среды

Виртуальная машина: VirtualBox

Модель процессора: 11th Gen Intel(R) Core(TM) i5-11400 @ 2.70GHz, ядер: 6, логических процессоров: 12.

Логических процессоров выделенных под виртуальную машину — 6.

Объем оперативной памяти: 6.0 Гб

Тип оперативной памяти: DDR4

Операционная система: Fedora Linux 38 (Workstation Edition)

Разрядность ОС: 64

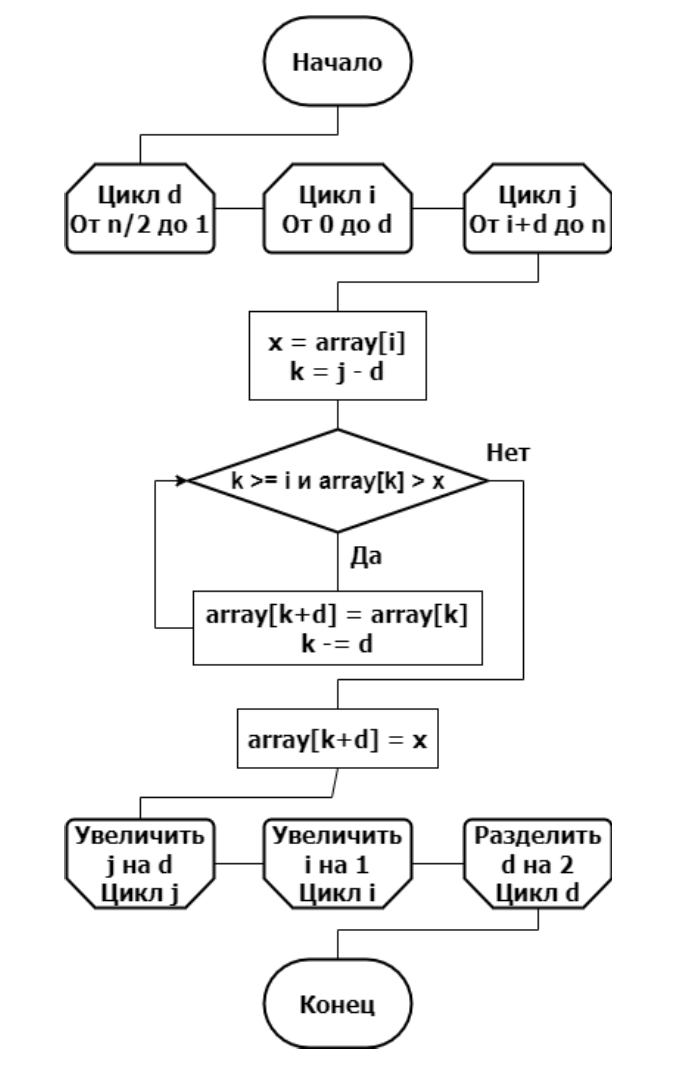
Среда разработки: gcc 13.2.1

Поддерживаемая версия OpenMP: 4.5

Анализ приведенного алгоритма

При сортировке Шелла исходный массив длины n разделяется на n/d подмассивов, которые образуют элементы исходного, находящиеся на расстоянии d. Далее они сортируются с помощью сортировки вставками. d с каждой итерацией уменьшается. Когда d = 1, совершается обычная сортировка вставками всего почти упорядоченного массива и алгоритм завершает свою работу. В обоих случаях массив длины 10^5 .

Блок-схема алгоритма:



Описание используемых директив и функций OpenMP и MPI

* #pragma omp parallel для инициализации области, выполняемой несколькими потоками
* #pragma omp for - для определения цикла, который будет разделен на потоки
* omp\_get\_wtime() для измерения времени исполнения параллельной области.
* MPI\_Init(&argc, &argv) - инициализирует MPI
* MPI\_Comm\_size(MPI\_COMM\_WORLD, &size) - узнать сколько всего потоков.
* MPI\_COMM\_WORLD - глобальный коммуникатор, который имеет доступ ко всем запущенным потокам.
* MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank) - возвращает номер процесса. Нужно для распараллеливания.
* MPI\_Scatter() и MPI\_Gather() для передачи частей исходного массива, нуждающихся в сортировке и сбора результатов сортировки
* MPI\_Barrier() для синхронизации всех процессов
* MPI\_Wtime() - астрономическое время в секундах.
* MPI\_Finalise() - завершить не основной поток.

Графики

На всех графиках ось абсцисс — количество потоков.

Заключение

При выполнении работы была написана программа, реализующая сортировку Шелла для массива, где шаг определяется так 𝑑\_0 = n / 2, d\_i = d\_(i - 1) / 2 с помощью двух технологий - MPI и OpenMP.

В результате измерения времени двух реализаций были построены графики времени работы, ускорения и эффективности. Реализации использующие MPI работали примерно столь же быстро, как и OpenMP, с поправкой на более медленное падение времени на 2 потоках в случае MPI (влияние необходимости пересылки сообщений между процессами) и увеличение времени на 5 потоках в случае OpenMP (вероятно из-за плохо сгенерированного массива или ошибок в работе виртуальной машины).

За исключением указанных случаев видно что графики времени и ускорения ведут себя примерно одинаково и выходит к асимптоте на 6 потоках, что соответствует 6 выделенным машине процессорам.

Приложение

Код файла openmp.c

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

int \*\*makeArr(int rows, int columns)

{

    int \*\*result = (int\*\*) malloc(rows \* sizeof(int\*));

    for (int i = 0; i < rows; ++i)

    {

        result[i] = (int\*) malloc(columns \* sizeof(int));

    }

    return result;

}

void cleanArr(int rows, int \*\*arrays) {

    for (int i = 0; i < rows; ++i)

    {

        free(arrays[i]);

    }

    free(arrays);

}

void fillRand(int \*\*arrays, int rows, int columns, int addon)

{

    for (int i = 0; i < rows; ++i)

    {

        srand(i + addon);

        for (int j = 0; j < columns; ++j)

        {

            arrays[i][j] = rand();

        }

    }

}

void printAvTime(int \*threadNums, double \*avtime, int threadNumsLen)

{

    printf("in milliseconds\n");

    for (int threadId = 0; threadId < threadNumsLen; ++threadId)

    {

        printf("%d: %.10lf\n", threadNums[threadId], avtime[threadId] \* 1000.0);

    }

}

void printDel(int minsCount)

{

    for (int minsId = 0; minsId < minsCount; ++minsId)

    {

        printf("-");

    }

    printf("\n");

}

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

{

    // Init data

    const int arLen = 1000000;

    const int arNum = 10;

    int threadNumsLen = 13;

    int threadsNum[13] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16, 32, 64};

    // Init measurement arrays

    double \*avTime = (double\*) malloc(threadNumsLen \* sizeof(double));

    // Init arrays

    int \*\*arrays = makeArr(arNum, arLen);

    // Start measuring

    for (int measId = 0; measId < threadNumsLen; ++measId)

    {

        int threadCount = threadsNum[measId];

        double totalTime = 0;

        for (int arId = 0; arId < arNum; ++arId)

        {

            fillRand(arrays, arNum, arLen, arId);

            double startTime = omp\_get\_wtime();

            for (int d = arLen / 2; d > 0; d /= 2)

            {

                int i, j, x, k;

                #pragma omp parallel for num\_threads(threadCount) shared(arrays, arId, arLen, d) private(i, j, x, k) default(none)

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

                {

                    for (j = i + d; j < arLen; j += d)

                    {

                        x = arrays[arId][j];

                        k = j - d;

                        while (k >= i && arrays[arId][k] > x)

                        {

                            arrays[arId][k + d] = arrays[arId][k];

                            k -= d;

                        }

                        arrays[arId][k + d] = x;

                    }

                }

            }

            totalTime += omp\_get\_wtime() - startTime;

        }

        avTime[measId] = totalTime / arNum;

    }

    cleanArr(arNum, arrays);

    printAvTime(threadsNum, avTime, threadNumsLen);

    printf("\n");

    return 0;

}

Код файла mpi.c

#include <stdlib.h>

#include <stdio.h>

#include <mpi.h>

#include <string.h>

#include <limits.h>

void shellSort(int \*A, int count)

{

    int x, k;

    for (int d = count / 2; d > 0; d /= 2)

    {

        for (int i = 0; i < d; ++i)

        {

            for (int j = i + d; j < count; j += d)

            {

                x = A[j];

                int k = j - d;

                while (k >= i && A[k] > x)

                {

                    A[k + d] = A[k];

                    k -= d;

                }

                A[k + d] = x;

            }

        }

    }

    return;

}

void afterSort(int \*input, int \*res, int block, int count, int size)

{

    int \*ps = (int\*) malloc(size \* sizeof(int));

    int x, nx;

    for (int i = 0; i < size; ++i)

    {

        ps[i] = i \* block;

    }

    for (int i = 0; i < count; ++i)

    {

        x = INT\_MAX;

        nx = -1;

        for (int j = 0; j < size; ++j)

        {

            if (ps[j] < (j + 1) \* block && input[ps[j]] < x)

            {

                x = input[ps[j]];

                nx = j;

            }

        }

        res[i] = x;

        ps[nx]++;

    }

    free(ps);

}

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

{

    MPI\_Init(&argc, &argv);

    if (argc != 1)

    {

        printf("Wrong usage");

        MPI\_Finalize();

        return 0;

    }

    srand(0);

    int size, rank;

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

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

    int rcount = 10e5;

    int count = rcount + (size - rcount % size);

    int block = count / size;

    int \*input, \*b, \*res;

    if (!rank)

    {

        input = (int\*) malloc(count \* sizeof(int));

        res = (int\*) malloc(count \* sizeof(int));

    }

    b = (int\*) malloc(block \* sizeof(int));

    if (!rank)

    {

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

        {

            input[i] = rand();

        }

    }

    MPI\_Scatter(input, block, MPI\_INT, b, block, MPI\_INT, 0, MPI\_COMM\_WORLD);

    double begin, end;

    begin = MPI\_Wtime();

    shellSort(b, block);

    MPI\_Gather(b, block, MPI\_INT, input, block, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Barrier(MPI\_COMM\_WORLD);

    if (!rank)

    {

        afterSort(input, res, block, count, size);

    }

    end = MPI\_Wtime();

    if (!rank)

    {

        free(input);

        free(res);

    }

    free(b);

    if (!rank)

    {

        FILE \*fp;

        fp = fopen("log.txt", "a");

        fprintf(fp, "%d %f\n", size, end - begin);

        fclose(fp);

    }

    MPI\_Finalize();

    return 0;

}

Приложение

Таблицы с теоретическими результатами и результатами вычислительных экспериментов Способ вычисления ускорения: T1 – время работы алгоритма с одним потоком

Tn – время работы алгоритма с n потоками

Sn – ускорение с n потоками, Sn = T1/Tn

En – эффективность с n потоками, En = Sn/n

