Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Нижегородский государственный университет им. Н.И. Лобачевского»
Институт информационных технологий, математики и механики

Отчет

по лабораторной работе №3

«Поразрядная сортировка для целых чисел с простым слиянием»

Выполнил:
студент группы 381606-1
Каганов Д.А.
Проверил:
Кустикова В.Д.

Нижний Новгород
2018

Содержание

Постановка задачи

В рамках данной работы ставится задача адаптации массива алгоритмом поразрядной сортировки с использованием простого слияния.

Метод решения

- Реализация разбиения массива на подмассивы.
- Отсортировать подмассивы алгоритмом поразрядной сортировки.
- Объединить подмассивы.
- Отсортировать получившийся массив простым слиянием.
- Сравнить время работы параллельного и линейного варианта реализации алгоритма.

Схема распараллеливания

Параллельный алгоритм поразрядной сортировки массива состоит из двух этапов. На первом этапе количество элементов массива целых чисел делится на количество процессов. Массив делится на отрезки, которые передаются процессам (в случае не целочисленного деления хвост обрабатывается в 0 процессе). На втором этапе на каждом процессе выполняется поразрядная сортировка отрезков массива, принадлежащих соответствующему процессу. Отсортированные части собираются на нулевом процессе, где по очереди сливаются в один массив.

Использование MPI функций для реализации параллельного алгоритма поразрядной сортировки:

- MPI_Bcast() – передача сообщения от одного процесса ко всем остальным процессам группы, включая его самого.
- MPI_Scatterv() – является векторным вариантом функции MPI_Scatter, позволяющим посылать каждому процессу различное количество элементов. Начало расположения элементов блока, посылаемого i-му процессу, задается в массиве смещений displs, а число посылаемых элементов - в массиве scounts.
- MPI_Send() – функция выполняет посылку scount элементов типа MPI_UNSIGNED сообщения с идентификатором tag нулевому процессу в области связи коммуникатора MPI_COMM_WORLD.
- MPI_Recv() – функция выполняет приём массива buff2 размера arrSize / procNum типа MPI_UNSIGNED с любым идентификатором от всех процессов в области коммуникатора MPI_COMM_WORLD.

Описание программной реализации

Руководство пользователя

Для запуска программы в командную строку нужно ввести следующее:

```
mpirun --oversubscribe -np A lab3.bin B C
```

где A – количество процессов, B – количество элементов массива, C – способ сортировки (0 - MSD или 1 – LSD сортировки соответственно).

Программа выведет размер массива, сам массив, затем отсортированный массив и время выполнения для линейного и параллельного алгоритма.

Пример:

```
MacBook-Air-dmtrii:demo dmitrijkaganov$ mpirun --oversubscribe -np 2 lab3.bin 20 0

Array size : 20
88 76 92 100 190 41 194 30 33 198 38 198 106 48 133 12 109 26 85 102

12 26 30 33 38 41 48 76 85 88 92 100 102 106 109 133 190 194 198 198
Liner time: 1.40667e-05
Array is sorted

12 26 30 33 38 41 48 76 85 88 92 100 102 106 109 133 190 194 198 198
Parallel Time: 0.000155926
Array is sorted

Arrays are the same
```

Рисунок 1. Пример запуска программы на двух процессах с размером массива 20 и MSD сортировкой

Руководство программиста

Описание алгоритмов

LSD сортировка

Метод поразрядной сортировки просматривает байты в направлении справа налево. На рисунке 2 показано, как задача сортировки трехбуквенных слов решается за три прохода по файлу.

now	sob	cab	ace
for	nob	wad	ago
tip	cab	tag	and
ilk	wad	jam	bet
dim	and	rap	cab
tag	ace	tap	caw
jot	wee	tar	cue
sob	cue	was	dim
nob	fee	caw	dug
sky	tag	raw	egg
hut	egg	jay	fee
ace	gig	ace	few
bet	dug	wee	for
men	ilk	fee	gig
egg	owl	men	hut
few	dim	bet	ilk
jay	jam	few	jam
owl	men	egg	jay
joy	ago	ago	jot
rap	tip	gig	joy
gig	rap	dim	men
wee	tap	tip	nob
was	for	sky	now
cab	tar	ilk	owl
wad	was	and	rap
tap	jot	sob	raw
caw	hut	nob	sky
cue	bet	for	sob
fee	you	jot	tag
raw	now	you	tap
ago	few	now	tar
tar	caw	joy	tip
jam	raw	cue	wad
dug	sky	dug	was
you	jay	hut	wee
and	joy	owl	you

Рисунок 2. Пример поразрядной LSD - сортировки

Метод LSD-сортировки упорядочивает трехбуквенные слова за три прохода (слева направо).

Файл сначала сортируется по последней букве (методом распределяющего подсчета), потом по средней букве, а затем по первой.

Если количество разрядов — константа, а k = O(n), то сложность цифровой сортировки составляет O(n), то есть она линейно зависит от количества сортируемых чисел. Где k – итерации, на каждой из которой выполняется устойчивая сортировка и не более O(1) других операций.

MSD-сортировка

Предположим, что нам нужно отсортировать числа, представленные в системе счисления по основанию R, рассматривая сначала наиболее значащие байты. Это требует разбиения массива не на 2, а на R различных частей. Традиционно эти части называются контейнерами, и мы будем представлять себе, что алгоритм работает с группой из R контейнеров, по одному для каждого возможного значения первой цифры, как показано на следующей диаграмме:

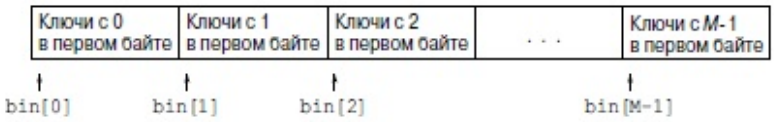

Рисунок 3. Диаграмма разбиения массива на R контейнеров

Мы последовательно просматриваем ключи, распределяя их по корзинам, затем рекурсивно сортируем содержимое каждого контейнера по ключам, которые короче на 1 байт.

На рисунке 4 показан пример работы MSD-сортировки на случайных перестановках целых чисел.

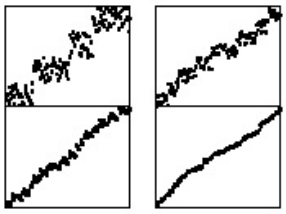

Рисунок 4. Динамические характеристики поразрядной MSD-сортировки

Как видно на данном примере случайно упорядоченного файла 8-разрядных целых чисел, всего лишь один шаг MSD-сортировки почти полностью упорядочивает данные. Первый шаг MSD-сортировки по двум старшим разрядам (слева) делит исходный файл на четыре подфайла. На следующем шаге каждый такой подфайл также делится на четыре подфайла. MSD-сортировка по трем старшим разрядам (справа) всего лишь за один проход распределяющего подсчета делит файл на восемь подфайлов. На следующем уровне каждый из этих подфайлов снова разбивается на восемь частей, после чего в каждой такой части содержится всего лишь несколько элементов.

Пусть значения разрядов меньше b, а количество разрядов — k. При сортировке массива из одинаковых элементов MSD-сортировкой на каждом шаге все элементы будут находится в неубывающей по размеру корзине, а так как цикл идет по всем элементам массива, то получим, что время работы MSD-сортировки оценивается величиной O(nk), причем это время нельзя улучшить. Хорошим случаем для данной сортировки будет массив, при котором на каждом шаге каждая корзина будет делиться на b частей. Как только размер корзины станет равен 1, сортировка перестанет рекурсивно запускаться в этой корзине. Таким образом, асимптотика будет $\Omega(n \log_b n)$. Это хорошо тем, что не зависит от числа разрядов.

Код программы можно просмотреть в разделе «Приложение».

`void swp(unsigned int *a, unsigned int *b)` – меняет местами элементы массива;

`unsigned int* merge(unsigned int *arr1, unsigned int num1, unsigned int *arr2, unsigned int num2)` – алгоритм простого слияния;

`void radixSortMSD` – MSD сортировка;

`void radixSortLSD(unsigned int *a, unsigned int count)` – LSD сортировка;

`void Check(unsigned int* arr, unsigned int procNum)` – проверка корректной работы сортировки;

`void CheckResult(unsigned int* resultL, unsigned int* result, unsigned int* arrSize)` – проверка массива, отсортированного линейно с массивом, отсортированным параллельно;

Подтверждение корректности

Для подтверждения корректности в программе реализованы функции проверки массива, отсортированного линейно с массивом, отсортированным параллельно, и проверки массива на корректность работы сортировки. Если массивы, отсортированные линейно и отсортированные параллельно не равны, то консоль выведет соответствующее сообщение. Если массив после работы функции сортировки не будет отсортирован корректно, то консоль также выведет соответствующее сообщение.

```
void Check(unsigned int* arr, unsigned int procNum) {
    for (int i = 0; i < procNum - 1; i++)
        if (arr[i] > arr[i + 1]) {
            cout << "Error! Array not sorted!\n";
            return;
        }
    cout << "Array is sorted\n";
}

void CheckResult(unsigned int* resultL, unsigned int* resultP, unsigned int arrSize) {
    for (int i = 0; i < arrSize; i++)
        if (resultL[i] != resultP[i]) {
            cout << "\nError! Linear and parallel results are not equal\n\n";
            return;
        }
    cout << "\nArrays are the same\n";
}
```

Результаты экспериментов

Результаты экспериментов запуска программы на 2 вычислительных узлах:

Количество элементов	10^4	10^5	10^6
Среднее ускорение при LSD сортировке	1.449	1.221	1.176
Среднее ускорение при MSD сортировке	1.558	1.703	1.704

Результаты экспериментов запуска программы на 4 вычислительных узлах:

Количество элементов	10^4	10^5	10^6
Среднее ускорение при LSD сортировке	0.963	1.549	1.248
Среднее ускорение при MSD сортировке	2.027	2.231	2.092

Заключение

Была разработана параллельная версия алгоритма поразрядной сортировки, которая эффективней линейной версии в два раза на достаточно больших данных.

Приложение

```
#include <mpi.h>
#include <ctime>
#include <iostream>
#include <assert.h>
#include <utility>
#include <limits>

#define MainProc 0

using namespace std;

void swp(unsigned int *a, unsigned int *b) {
    unsigned int tmp = *a;
    *a = *b;
    *b = tmp;
}

unsigned int* merge(unsigned int *arr1, unsigned int num1,
                    unsigned int *arr2, unsigned int num2) {
    unsigned int *res;
    unsigned int i = 0,
    j = 0,
    index = 0;

    res = new unsigned int[num1 + num2];

    while (i < num1 && j < num2) {
        if (arr1[i] < arr2[j])
            res[index++] = arr1[i++];
        else
            res[index++] = arr2[j++];
    }

    while (i < num1)
        res[index++] = arr1[i++];

    while (j < num2)
        res[index++] = arr2[j++];
    return res;
}

void radixSortMSD(unsigned int *from, unsigned int *to, unsigned int bit) {
    if (!bit || to < from + 1) return;

    unsigned int *ll = from, *rr = to - 1;

    for (;;) {
        while (ll < rr && !(*ll & bit)) ll++;
        while (ll < rr && (*rr & bit)) rr--;
        if (ll >= rr) break;
        swp(ll, rr);
    }

    if (!(bit & *ll) && ll < to) ll++;
    bit >>= 1;

    radixSortMSD(from, ll, bit);
    radixSortMSD(ll, to, bit);
}

void radixSortLSD(unsigned int *a, unsigned int count) {
    unsigned int mIndex[4][256] = { 0 };        // count and index matrix
    unsigned int *b = new unsigned int[count];  // allocate temp array
```

```
    unsigned int i, j, m, n;
    unsigned int u;
    for (i = 0; i < count; i++) {
        u = a[i];
        for (j = 0; j < 4; j++) {
            mIndex[j][static_cast<unsigned int>(u & 0xff)]++;
            u >>= 8;
        }
    }
    for (j = 0; j < 4; j++) {           // convert to indices
        m = 0;
        for (i = 0; i < 256; i++) {
            n = mIndex[j][i];
            mIndex[j][i] = m;
            m += n;
        }
    }
    for (j = 0; j < 4; j++) {           // radix sort
        for (i = 0; i < count; i++) {   // sort by current lsb
            u = a[i];
            m = static_cast<unsigned int>(u >> (j << 3)) & 0xff;
            b[mIndex[j][m]++] = u;
        }
        swap(a, b);  // swap ptrs
    }
    delete[] b;
}

void Check(unsigned int* arr, unsigned int procNum) {
    for (int i = 0; i < procNum - 1; i++)
        if (arr[i] > arr[i + 1]) {
            cout << "Error! Array not sorted!\n";
            return;
        }
    cout << "Array is sorted\n";
}

void CheckResult(unsigned int* resultL, unsigned int* resultP, unsigned int arrSize) {
    for (int i = 0; i < arrSize; i++)
        if (resultL[i] != resultP[i]) {
            cout << "\nError! Linear and parallel results are not equal\n\n";
            return;
        }
    cout << "\nArrays are the same\n";
}

int main(int argc, char** argv) {
    int status = 0;
    int procRank = 0;
    int procNum = 0;
    int *scounts = NULL;                    // use in scatterv
    int *displs = NULL;                     // use in scatterv
    unsigned int arrSize = 0;
    unsigned int *arr = NULL;               // data
    unsigned int *buff = NULL;              // buffers for message exchanging
    unsigned int *buff2 = NULL;
    unsigned int *resultL = NULL;           // results
    unsigned int *resultP = NULL;
    double linerStart = 0.0;
    double linerEnd = 0.0;
    double parallelStart = 0.0;
    double parallelEnd = 0.0;
    int typeSort = 0;
    int imin = numeric_limits<int>::min();

    status = MPI_Init(&argc, &argv);
```

```
    status = MPI_Comm_size(MPI_COMM_WORLD, &procNum);
    status = MPI_Comm_rank(MPI_COMM_WORLD, &procRank);

    // read data from the cmd
    if (procRank == MainProc) {
        if (argc > 1) {
            arrSize = atoi(argv[1]);
            if (argc > 2) typeSort = atoi(argv[2]);
        } else {
            arrSize = 10;
            typeSort = 0;
        }

        cout << "\nArray size : " << arrSize << endl;
        arr = new unsigned int[arrSize];
        resultL = new unsigned int[arrSize];
        resultP = new unsigned int[arrSize];
        srand((unsigned)time(NULL));

        for (int i = 0; i < arrSize; i++)
            arr[i] = resultL[i] = rand() % 200;
        if (arrSize < 31) {
            for (int i = 0; i < arrSize; i++)
                cout << arr[i] << " ";
            cout << endl;
        }

// LINEAR BLOCK
        linerStart = MPI_Wtime();
        if (typeSort == 0)
            radixSortMSD(resultL, resultL + arrSize, imin);
        else
            radixSortLSD(resultL, arrSize);
        linerEnd = MPI_Wtime();
        if (arrSize < 31) {
            cout << endl;
            for (int i = 0; i < arrSize; i++)
                cout << resultL[i] << " ";
        }
        cout << "\nLiner time: " << linerEnd - linerStart << endl;
        Check(resultL, arrSize);
// END LINEAR BLOCK

// PARALLEL BLOCK
        buff = new unsigned int[arrSize / procNum + arrSize % procNum];
        buff2 = new unsigned int[arrSize / procNum];
    }
    MPI_Barrier(MPI_COMM_WORLD);
    parallelStart = MPI_Wtime();
    MPI_Bcast(&arrSize, 1, MPI_UNSIGNED, MainProc, MPI_COMM_WORLD);
    MPI_Bcast(&typeSort, 1, MPI_UNSIGNED, MainProc, MPI_COMM_WORLD);

    scounts = new int[procNum];
    displs = new int[procNum];
    displs[0] = 0;
    scounts[0] = arrSize / procNum + arrSize % procNum;

    for (int i = 1; i < procNum; i++) {
        scounts[i] = arrSize / procNum;
        displs[i] = scounts[0] + (i-1)*scounts[i];
    }

    if (procRank != 0)
        buff = new unsigned int[scounts[procRank]];
```

```
    MPI_Scatterv(arr, scounts, displs, MPI_UNSIGNED, buff, scounts[procRank], MPI_UNSIGNED,
MainProc, MPI_COMM_WORLD);

    if (typeSort == 0)
        radixSortMSD(buff, buff + scounts[procRank], imin);
    else
        radixSortLSD(buff, scounts[procRank]);

    if (procRank != 0)
        MPI_Send(buff, scounts[procRank], MPI_UNSIGNED, MainProc, procRank, MPI_COMM_WORLD);


    if (procRank == 0) {
        for (int i = 0; i < static_cast<unsigned int>(scounts[0]); i++)
            resultP[i] = buff[i];
        for (int i = 1; i < static_cast<unsigned int>(procNum); i++) {
            MPI_Recv(buff2, arrSize / procNum, MPI_UNSIGNED, MPI_ANY_SOURCE, i,
MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
            resultP = merge(buff2, arrSize / procNum, resultP, scounts[0] + (i -
1)*scounts[1]);
        }
        // parallel results
        parallelEnd = MPI_Wtime();
        if (arrSize < 31) {
            cout << endl;
            for (unsigned int i = 0; i < arrSize; i++)
                cout << resultP[i] << " ";
        }
        cout << "\nParallel Time: " << parallelEnd - parallelStart << endl;
        Check(resultP, arrSize);
        CheckResult(resultL, resultP, arrSize);
        cout << "\nAverage acceleration: " << (linerEnd - linerStart) / (parallelEnd -
parallelStart) << endl;
        cout << endl;

        delete[] buff2;
        delete[] resultL;
        delete[] resultP;
        delete[] arr;
    }
    delete[] buff;
    delete[] scounts;
    delete[] displs;
    status = MPI_Finalize();

    return 0;
}
```