Министерство образования и науки Российской Федерации

Государственное образовательное учреждение высшего профессионального образования «Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий математики и механики

Кафедра: Программная инженерия

Специальность (направление): Программная инженерия

Отчет

по лабораторной работе

по дисциплине «Параллельное программирование»

тема:

«Поразрядная сортировка для вещественных чисел (тип double) с простым слиянием»

Выполнил:

студент группы 381508 Максименко А.Н.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Подпись

Научный руководитель:

Доцент, к.т.н. Сысоев А.В.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Подпись

Нижний Новгород 2018

**Содержание**

Условие задачи 3

Решение задачи3

Последовательная версия решения задачи(Простейшее решение)4

OpenMP версия решения задачи5

TBB версия решения задачи7

Инструменты для решения данной задачи11

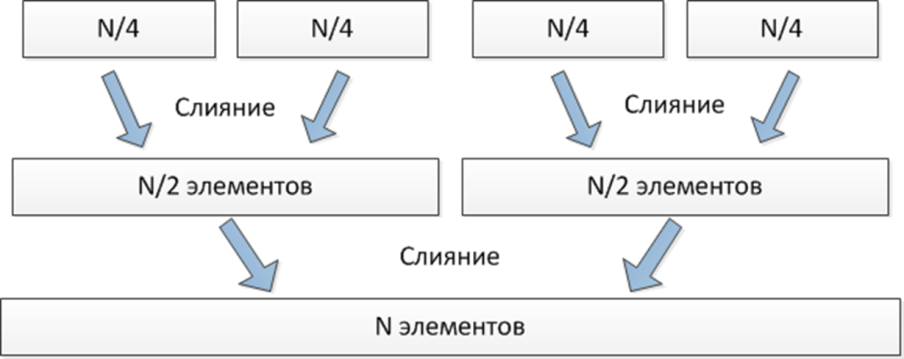
1. **Условие задачи**

Реализовать параллельные алгоритмы сортировки слиянием для массива Val с N элементами, с простым слиянием, с использованием технологии OpenMP и TBB.

Val – массив вещественных чисел из N элементов. Элементы массива имеют тип число с плавающей запятой двойной точности (double).

1. **Решение задачи**

Идея простого слияния заключается в том, что один поток может выполнять слияние двух отсортированных массивов по классическому алгоритму. В этом случае слияние *n* массивов могут выполнять *n*/2 параллельных потоков. На следующем шаге слияние *n*/2 полученных массивов будут выполнять *n*/4 потоков и т.д.



1. **Последовательная версия решения задачи (Простейшее решение)**

Для решения задачи алгоритму на вход необходимо получать данные, и на выходе он так же должен что-то отдавать. Для этого будут использоваться файлы. Так как чтение/запись файлов занимает некоторое время, что влияет на корректное измерение ускорения, полученного программой для нескольких потоков. Поэтому нужно отделить чтение/запись данных и реализацию алгоритма. Таким образом, у нас будет присутствовать два файла:

sol.h – решение задачи

before\_code.cpp – чтение данных и точка входа в программу

sol.h

void MergeSorting(int l, int r, double\* &x, int &count) {

if (r == l) // если колличество достаточно млао, то сливаем массивы

return;

if (r - l == 1) { // если в массиве два элемента, то сортируем их и сливаем массивы

if (x[r] < x[l])

std::swap(x[r], x[l]);

return;

}

int m = (r + l) / 2; // переменная определяет середину массива

MergeSorting(l, m, x, count);

MergeSorting(m + 1, r, x, count);

double\* buf; // указатель на воспомогательный массив [ВМ]

buf = new double[count]; // выделение памяти под ВМ

int xl = l; // позиция первого элемента 1-го массива

int xr = m + 1; // позиция первого элемента 2-го массива

int cur = 0; // счётчик элементов

while (r - l + 1 != cur) { // слияние

if (xl > m)

buf[cur++] = x[xr++];

else if (xr > r)

buf[cur++] = x[xl++];

else if (x[xl] > x[xr])

buf[cur++] = x[xr++];

else buf[cur++] = x[xl++];

}

for (int i = 0; i < cur; i++) // запись в основной массив

x[i + l] = buf[i];

delete[] buf;

}

Формат входных данных:

Функция получает в аргументах следующие переменные:

l – позиция первого элемента первого массива

r – позиция первого элемента второго массива

x – массив данных, в котором хранятся числа

count – количество элементов массива

before\_code.cpp

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

{

if (argc != 3) return 0; // если команда не соответствует требованиям, то завершаем программу

int count;

double\* values;

freopen(argv[1], "rb", stdin);

freopen(argv[2], "wb", stdout);

fread(&count, sizeof(count), 1, stdin);

values = new double[count];

fread(values, sizeof(\*values), count, stdin);

double time = clock();

MergeSorting(0, count - 1, values, count);

time = clock() - time;

fwrite(&time, sizeof(time), 1, stdout);

fwrite(&count, sizeof(count), 1, stdout);

fwrite(values, sizeof(\*values), count, stdout);

return 0;

}

Формат входных данных для командной строки:

argv[1] – имя .exe файла

argv[2] – имя файла из которого будем считывать данные

argv[3] – имя файла куда будем записывать данные

Если файла для записи нет с таким именем, то он будет создан.

Считывание данных будет производиться следующим образом:

* Первым элементом является число элементов массива. Оно считывается первым для выделения памяти под массив, который нужно отсортировать
* Остальными элементами являются непосредственно сами элементы массива

Запись данных производится в аналогичной последовательности, только второй элемент по записи будет выступать время, за которое был отсортирован массив.

Для чтения/записи бинарных файлов использованы функции fread fwrite. Функция freopen перенаправляет данные файла в заданный поток или наоборот.

1. **OpenMP версия решения задачи**

**omp\_merge\_sort.h**

vector<double> merge(const vector<double>& left, const vector<double>& right) // функция слияния

{

vector<double> result;

unsigned left\_it = 0, right\_it = 0;

while (left\_it < left.size() && right\_it < right.size()) // сравнивание первых элементов двух отсортированных массивов

{

if (left[left\_it] < right[right\_it])

{

result.push\_back(left[left\_it]);

left\_it++;

}

else

{

result.push\_back(right[right\_it]);

right\_it++;

}

}

// если в одном из массивов закончились элементы, то добавляем в результирующий вектор элементы из другого массива

while (left\_it < left.size())

{

result.push\_back(left[left\_it]);

left\_it++;

}

while (right\_it < right.size())

{

result.push\_back(right[right\_it]);

right\_it++;

}

return result;

}

vector<double> mergesort(vector<double>& vec, int threads) // Функция сортировки OpenMP

{

if (vec.size() == 1) // если массив состоит из одного элемента, то возвращаем его как результат

{

return vec;

}

std::vector<double>::iterator middle = vec.begin() + (vec.size() / 2); // определяем середину массива

vector<double> left(vec.begin(), middle);

vector<double> right(middle, vec.end());

if (threads > 1)

{

#pragma omp parallel sections

{

#pragma omp section

{

left = mergesort(left, threads / 2);

}

#pragma omp section

{

right = mergesort(right, threads - threads / 2);

}

}

}

else

{

left = mergesort(left, 1);

right = mergesort(right, 1);

}

return merge(left, right);

}

Формат входных данных:

На вход функции слияния (merge) подаётся два отсортированных массива, которые, в свою очередь, сливает в один.

На выход функции сортировки (mergeSort) подаётся сам массив данных и количество потоков, которое будет производить сортировку.

Сама суть сортировки версии OpenMP заключается в рекурсивно вызове функции сортировки, которая будет создавать параллельные секции для каждого потока для работы со своими данными. При каждом вызове функции, массив будет делиться пополам до тех пор, пока количество элементов в массиве станет равным 1. После чего, будет происходить сливание массивов в один.

**omp\_before\_code.cpp**

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

{

if (argc != 4) return 0; // если argc не соответствует формату входных данных, то завершаем программу

int count;

double\* values;

int threads = 1; // объявление переменной, отвечающей за кол-во потоков

threads = atoi(argv[3]); // переназначение переменной количества потоков данными из командной строки

freopen(argv[1], "rb", stdin);

freopen(argv[2], "wb", stdout);

fread(&count, sizeof(count), 1, stdin);

values = new double[count];

fread(values, sizeof(\*values), count, stdin);

vector<double> vector\_values(count);

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

vector\_values[i] = values[i];

}

omp\_set\_num\_threads(threads);

double time = omp\_get\_wtime();

vector\_values = mergesort(vector\_values, threads);

time = omp\_get\_wtime() - time;

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

values[i] = vector\_values[i];

fwrite(&time, sizeof(time), 1, stdout);

fwrite(&count, sizeof(count), 1, stdout);

fwrite(values, sizeof(\*values), count, stdout);

return 0;

}

Формат входных данных для командной строки:

argv[1] – имя .exe файла

argv[2] – имя файла из которого будем считывать данные

argv[3] – имя файла куда будем записывать данные

argv[4] – число, отвечающее за количество потоков

Всё по аналогии, как для последовательной версии.

Так же, здесь присутствует инициализация потоков, количество которых будет указано в поле threads.

1. **TBB версия решения задачи**

tbb\_merge\_sort.h

class ParallelSorter :public task

{

private:

double \*values;

double \*add;

int count;

int portion;

void Split(int size1, int size2)

{

for (int i = 0; i < size1; i++)

add[i] = values[i];

double \*values2 = values + size1;

int a = 0;

int b = 0;

int i = 0;

while ((a != size1) && (b != size2))

{

if (add[a] <= values2[b])

{

values[i] = add[a];

a++;

}

else

{

values[i] = values2[b];

b++;

}

i++;

}

if (a == size1)

for (int j = b; j<size2; j++)

values[size1 + j] = values2[j];

else

for (int j = a; j<size1; j++)

values[size2 + j] = add[j];

}

void SortDouble(double\* values, double\* add, int count)

{

if (count == 1)

return;

else

if (count == 2) {

if (values[0] > values[1])

swap(values[0], values[1]);

return;

}

int m = count / 2;

SortDouble(values, add, m);

SortDouble(values + m, add + m, count - m);

int a = 0;

int b = m;

int j = 0;

while ((a != m) && (b != count))

{

if (values[a] > values[b])

{

add[j] = values[b];

b++;

}

else

{

add[j] = values[a];

a++;

}

j++;

}

if (a == m)

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

{

add[j] = values[i];

j++;

}

else

{

for (int i = a; i < m; i++)

{

add[j] = values[i];

j++;

}

}

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

values[i] = add[i];

}

public:

ParallelSorter(double\* \_values, double\* \_add, int \_count, int \_portion) : values(\_values), add(\_add), count(\_count), portion(\_portion)

{}

task\* execute()

{

if (count <= portion)

{

SortDouble(values, add, count);

}

else

{

ParallelSorter &sorter1 = \*new(allocate\_child())ParallelSorter(values, add, count / 2, portion);

ParallelSorter &sorter2 = \*new(allocate\_child())ParallelSorter(values + count / 2, add + count / 2, count - count / 2, portion);

set\_ref\_count(3);

spawn(sorter1);

spawn\_and\_wait\_for\_all(sorter2);

Split(count / 2, count - count / 2);

}

return NULL;

}

};

void ParallelSortDouble(double \*mas, int count, int nThreads)

{

double \*temp = new double[count];

int portion = count / nThreads;

if (count%nThreads != 0)

portion++;

ParallelSorter& sorter = \*new(task::allocate\_root())ParallelSorter(mas, temp, count, portion);

task::spawn\_root\_and\_wait(sorter);

delete[] temp;

}

Формат входных данных:

Для данной версии был реализован класс ParallelSorter, который включает в себя:

* Поле values – указатель на массив данных, с которым будем работать
* Поле add – указатель на вспомогательный массив, который будет служить для функции последовательной сортировки
* Поле count – количество элементов в массиве
* Поле portion – количество элементов, которое будет отдано каждому потоку. Так же определяет, будет ли вызываться последовательный алгоритм поразрядной сортировки или, начиная с этого размера, будет выполняться слияние

Так же присутствую функции:

* Split(size1, size2) – функция слияния двух массивов, входные данные которой являются количества элементов в двух сливаемых массивах
* SortDouble(values, add, count) – функция последовательной сортировки, входные данные которой являются указателями на первые элементы основного и вспомогательного массивов и количество элементов

Так же перегружен виртуальный метод task\* task::execute(), в котором выполняются вычисления.

В нём производятся проверка количества элементов в массиве на приемлемую, путём сравнения с порцией вычисления. Если же она приемлема, то будет выполняться функция SordDouble. Иначе будет создано две подчинённые задачи типа ParallelSorter, данные которой будут разделены на двое. Так же для данной задачи устанавливается число задач, у которых текущая задача будет указана в поле (2 + 1). После этого добавляем задачи в пул готовых к выполнению и ожидает завершения всех подчинённых задач. После чего происходит слияние двух отсортированных массивов.

Кроме этого реализована функция ParallelSortDouble, которая создаёт корневой task, начиная с которого будет разворачиваться рекурсия. Так же в этой функции создаётся вспомогательный массив размера count и выполняется вычисление порции.

tbb\_before\_code.cpp

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

{

if (argc != 4) return 0;

int count;

double\* values;

int threads = 1;

threads = atoi(argv[3]);

freopen(argv[1], "rb", stdin);

freopen(argv[2], "wb", stdout);

fread(&count, sizeof(count), 1, stdin);

values = new double[count];

fread(values, sizeof(\*values), count, stdin);

task\_scheduler\_init init(threads);

tick\_count start = tick\_count::now();

ParallelSortDouble(values, count, threads);

tick\_count finish = tick\_count::now();

double time = (finish - start).seconds();

fwrite(&time, sizeof(time), 1, stdout);

fwrite(&count, sizeof(count), 1, stdout);

fwrite(values, sizeof(\*values), count, stdout);

return 0;

}

Формат входных данных для командной строки:

argv[1] – имя .exe файла

argv[2] – имя файла из которого будем считывать данные

argv[3] – имя файла куда будем записывать данные

argv[4] – число, отвечающее за количество потоков

Так же, здесь присутствует инициализация потоков, количество которых будет указано в поле threads.

1. **Инструменты для решения данной задачи**

**Генератор тестов**

Для автоматического создания тестов присутствует генератор тестов. Генерация значений происходит в диапазоне от [-10000, 10000].

testGenerator.cpp

// каждому номеру теста соответствует колличество элементов в массиве

int n\_tests[] = { 1, 2, 3, 5, 7, 8, 9, 10, 10, 1000, 10000, 10000, 10000, 500000, 100000, 1000000, 10000000, 25000000, 50000000, 100000000 };

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

// на вход должно подаваться одно число в диапозоне от [0,20] с целью уникальности теста

if (argc != 2 || atoi(argv[1]) < 1 || atoi(argv[1]) > 20) return 0;

// определяем размерность нашего массива

int n = n\_tests[atoi(argv[1]) - 1];

// создаём файл(или перезаписываем если он существует) в который будем записывать результат работы программы

freopen(argv[1], "wb", stdout);

// создаём генератор случайных чисел с seed равным количеству времени с начала эпохи

default\_random\_engine generator(static\_cast<unsigned int>(chrono::system\_clock::now().time\_since\_epoch().count()));

// создаём равномерное распределение случайной величины типа double в диапазоне [-10000, 10000]

uniform\_real\_distribution <double> distribution(-1e4, 1e4);

// записываем кол-во элементов массива

fwrite(&n, sizeof(n), 1, stdout);

double \*values = new double[n];

// заполнение массива одинаковыми элементами

if (n == 12) {

double value = distribution(generator);

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

values[i] = value;

}

else

// заполнение массива по невозрастанию

if (n == 13) {

double value = distribution(generator);

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

values[i] = value - i;

}

else

// заполнение массива не по убыванию

if (n == 14) {

double value = distribution(generator);

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

values[i] = value + i;

}

else

// заполнение массива случайными элементами типа double в диапозоне от [-10000, 10000]

for (int i = 0; i < n; i++) {

values[i] = distribution(generator);

}

// запись массива в файл

fwrite(values, sizeof(\*values), n, stdout);

return 0;

}

Формат входных данных для командной строки:

argv[1] – номер n\_tests, отвечающий за количество элементов, который будет сгенерирован

**Cheker**

Checker – программа проверки (от английского check – проверять).

Цель программы – осуществляет проверку на правильность отсортированного массива и выносит вердикт.

Checker.cpp

enum verdict { NO = 1, AC, WA, CE, ML, TL, RE, IL, PE, DE };

class result {

private:

FILE \* bur;

public:

enum ext\_cls { NO = 1, VERDICT, MESSAGE, TIME, MEMORY };

result(bool read = false) {

if (read)

bur = fopen("result.txt", "r");

else

bur = fopen("result.txt", "w");

}

~result() {

fclose(bur);

}

void write\_type(ext\_cls t) {

fwrite(&t, sizeof(t), 1, bur);

}

void write\_verdict(verdict v) {

write\_type(ext\_cls::VERDICT);

fwrite(&v, sizeof(v), 1, bur);

}

void write\_message(string str) {

write\_type(ext\_cls::MESSAGE);

int l = str.size();

fwrite(&l, sizeof(l), 1, bur);

fwrite(&str[0], sizeof(str[0]), l, bur);

}

void write\_time(long long x) {

write\_type(ext\_cls::TIME);

fwrite(&x, sizeof(x), 1, bur);

}

void write\_memory(unsigned long long x) {

write\_type(ext\_cls::MEMORY);

fwrite(&x, sizeof(x), 1, bur);

}

} checker\_result;

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

if (argc != 2) return 0;

FILE \* buo = fopen(argv[1], "rb");

int res\_n;

double res\_time;

fread(&res\_time, sizeof(res\_time), 1, buo);

fread(&res\_n, sizeof(res\_n), 1, buo);

bool error = false;

double lower\_value = 0, more\_value = 0;

fread(&lower\_value, sizeof(lower\_value), 1, buo);

for (int i = 1; i < res\_n; i++) {

fread(&more\_value, sizeof(more\_value), 1, buo);

if (lower\_value < more\_value) {

error = true;

break;

}

lower\_value = more\_value;

}

if (error) {

checker\_result.write\_message("WA. Output is not correct.");

checker\_result.write\_verdict(verdict::WA);

}

else {

checker\_result.write\_message("AC. Numbers are equal.");

checker\_result.write\_verdict(verdict::AC);

}

checker\_result.write\_time(res\_time \* 1e7);

fclose(buo);

return 0;

}

Формат входных данных для командной строки:

argv[1] – имя файла, в котором хранится отсортированный массив

**Typer**

Программа получает на вход имя текстового файла (через аргумент командой строки), в котором записаны входные данные в текстовом виде, и преобразует их в бинарный вид.

typer.cpp

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

int count;

double\* values;

double time;

FILE \*fp;

fp = fopen(argv[1], "r");

fscanf(fp, "%lf\n", &time);

fscanf(fp, "%d\n", &count);

values = new double[count];

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

fscanf(fp, "%lf\n", &values[i]);

freopen(argv[2], "wb", stdin);

fwrite(&time, sizeof(time), 1, stdin);

fwrite(&count, sizeof(count), 1, stdin);

fwrite(values, sizeof(\*values), count, stdin);

return 0;

}

Формат входных данных для командной строки:

argv[1] – имя текстового файла

argv[2] – имя бинарного файла

**Viewer**

Программа получает на вход имя файла и преобразует его содержимое из бинарного вида в текстовый.

viewer.cpp

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

int count;

double\* values;

double time;

freopen(argv[1], "rb", stdin);

fread(&time, sizeof(time), 1, stdin);

fread(&count, sizeof(count), 1, stdin);

values = new double[count];

fread(values, sizeof(\*values), count, stdin);

FILE \*fp;

fp = fopen(argv[2], "w");

fprintf(fp, "%lf\n", time);

fprintf(fp, "%d\n", count);

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

fprintf(fp, "%lf\n", values[i]);

return 0;

}

Формат входных данных для командной строки:

argv[1] – имя бинарного файла

argv[2] – имя текстового файла