Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение  
высшего образования «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

#### **Институт информационных технологий, математики и механики**

**ОТЧЕТ**

**Поразрядная сортировка для вещественных чисел  
с четно-нечетным слиянием Бэтчера**

**Выполнил**:студент группы 381506-3

Маячкин Арсений

**Проверил:** кандидат технических наук

доцент кафедры МОСТ, ИТММ

Сысоев А. В.

Нижний Новгород

2018

Оглавление

[Постановка задачи 3](#_Toc514709780)

[Метод решения 4](#_Toc514709781)

[Схема распараллеливания 5](#_Toc514709782)

[Описание программной реализации 6](#_Toc514709783)

[Руководство пользователя 6](#_Toc514709784)

[Руководство программиста 6](#_Toc514709785)

[Проверка корректности 8](#_Toc514709786)

[Результаты экспериментов 9](#_Toc514709787)

[Эксперименты проводились на компьютере со следующими техническими характеристиками: 9](#_Toc514709788)

[Вывод 10](#_Toc514709789)

# Постановка задачи

 Дан неупорядоченный массив, состоящий из N положительных и отрицательных элементов типа double. Требуется реализовать последовательную и параллельную версию алгоритма поразрядной сортировки для вещественных чисел (с четно-нечетным слияние Бэтчера для параллельных версий) и сравнить их время работы при различных значениях размера сортируемого массива, а также сделать вывод о том, эффективно ли использовать параллельную версию.

Параллельную версию необходимо реализовать с использование библиотек OpenMP и TBB.

# Метод решения

Рассмотрим последовательную версию. В данной работе был использован алгоритм восходящей поразрядной сортировки LSD Radix Sort (сортировка от младшего байта к старшему байту).

В реализованном алгоритме исходный массив разбивается на подмножества положительных и отрицательных элементов.

В памяти компьютера для хранения элементов типа double требуется 8 байт. Для хранения знака элемента типа double требуется 1 бит, для порядка 11 бит, а для мантиссы 52 бита.

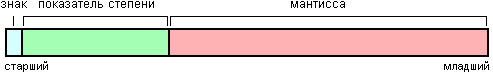


Рисунок 1. Хранение вещественного числа в памяти

LSD Radix Sort заключается в вызове сортировка подсчетом 8 раз для положительных и 8 раз для отрицательных чисел. Сортировка подсчетом выполняет упорядочивание элементов конкретного множества по i-му биту (i = 0..7).

Множества положительных и отрицательных элементов необходимо сортировать отдельно, потому что в случае с положительными числами большее значение мантиссы соответствует большему элементу, а для отрицательных - меньшему. Поэтому для отрицательных чисел сортировка должна выполняться в обратном порядке (1 «меньше» 0).

После того, как было сделано 16 вызовов функции сортировки подсчетом, объединяем соответствующим образом два отсортированных массива в один результирующий (элементы отрицательного множества помещаем в начальные позиции результирующего массива, а все положительные после отрицательных).

Алгоритм сортировки подсчетом выполняется следующим образом (рассмотрим для положительных элементов):

1. На вход алгоритму поступают массивы input и output, их размер N, а также байт i, по которому будет производиться сортировка на данный момент.
2. Завести массив counter, размером 256 элементов, который будет содержать количество встречаемости каждого байта во всех элементах массива input.
3. Завести переменную offset, в которой на основании данных из пункта 2 в цикле (256 итераций) вычисляется подсчёт смещений, и значение переменной offset заносится в массив counter (теперь counter будет содержать смещения, по которым будут сохраняться элементы в массив output).
4. Выполняем копирование элементов из массива input в массив output, используя массив counter, полученный на этапе 3.
5. Массив output будет содержать результат выполнения алгоритма сортировки подсчетом.

Вычислительная сложность алгоритма восходящей поразрядной сортировки будет линейной (от числа элементов исходного массива) в силу описанного алгоритма сортировки подсчетом (так как максимум проход по всем элементам массива осуществляется за O(n)).

# Схема распараллеливания

Для распараллеливания сортировки используется четно-нечетное слияние Бэтчера. Идея параллельной реализации с использование слияния заключается в выполнении следующих шагов:

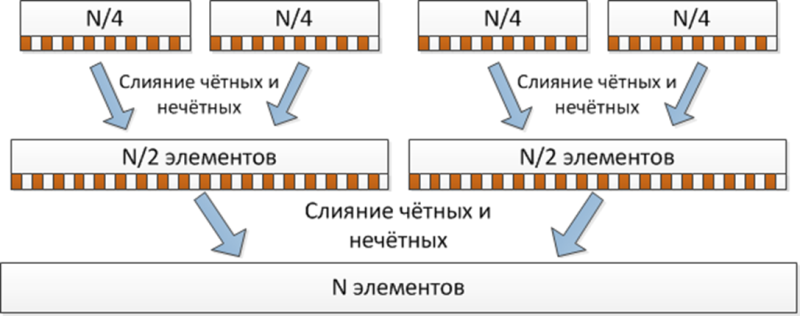
* Сортировка частей массива каждым процессом.
* Слияние отсортированных частей массива.

В первом пункте для сортировки частей мы будем использовать поразрядную сортировку.

Во втором пункте для слияния будет использоваться алгоритм Бэтчера.

Четно-нечетное слияние Бэтчера заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на четные и нечетные элементы. Далее выполняется слияние четных и нечетных элементов массивов. Чтобы массив стал окончательно отсортированным, достаточно сравнить пары элементов, стоящие на нечетной и четной позициях. Первый и последний элементы массива проверять не надо, т.к. они являются минимальным и максимальным элементов массивов.

Четно-нечетное слияние Бэтчера позволяет задействовать 2 потока при слиянии двух упорядоченных массивов. В этом случае слияние *n* массивов могут выполнять *n* параллельных потоков. На следующем шаге слияние *n*/2 полученных массивов будут выполнять *n*/2 потоков и т.д. На последнем шаге два массива будут сливать 2 потока.



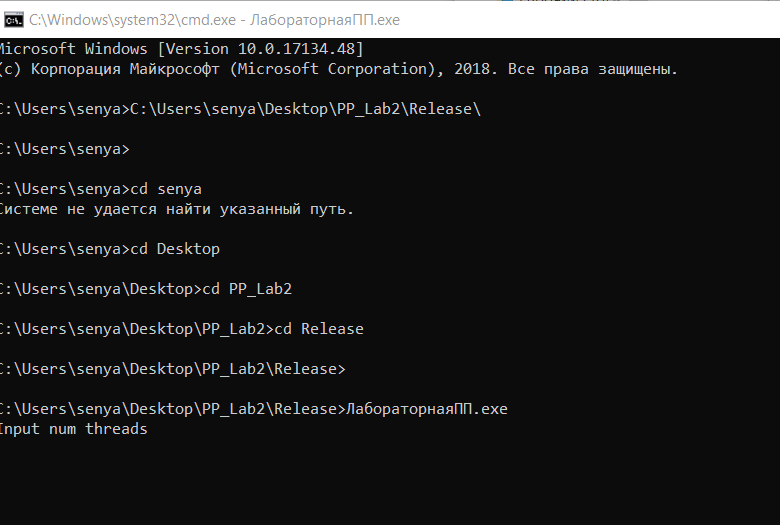
**Рисунок 2.** Четно-нечетное слияние Бэтчера

# Описание программной реализации

## Руководство пользователя

Для запуска программы необходимо в консоли указать путь до файла и выполнить его вызов.

1. Пример запуска программы:



1. Программа записывает результаты работы в файл result.txt

Результатом работы является вердикт по проверке корректности параллельной сортировки и время работы последовательной и параллельной версий.

## Руководство программиста

solution.cpp

|  |  |
| --- | --- |
| void PlusArrSort(double\* data, double\* temp, int dataLength, int numOfByte) | Функция сортировки положительных чисел по байту. |
| void MinusArrSort(double\* data, double\* temp, int dataLength, int numOfByte) | Функция сортировки отрицательных чисел по байту. |
| void RadixSort(double\* data, int size) | Функция выполняет алгоритм восходящей поразрядной сортировки LSD Radix Sort.  Для сортировки массива необходимо вызвать данную функцию. |

omp\_solution.cpp

|  |  |
| --- | --- |
| void RadixSort(double\* data, int dataLength) | Функция выполняет алгоритм восходящей поразрядной сортировки LSD Radix Sort. |
| void OddSplitter(double\* mas, double\* tmp, int size1, int size2) | Функция сортировки нечетных элементов двух смежных массивов |
| void EvenSplitter(double\* mas, double\* tmp, int size1, int size2) | Функция сортировки четных элементов двух смежных массивов |
| void SimpleComparator(double\* mas, int size) | Функция сравнения четных и нечетных пар элементов массива |
| void ParallelSort(double\* data, int size, int pnum) | Функция выполняет параллельную сортировку массива с заданным количеством процессов.  Для сортировки массива необходимо вызвать данную функцию. |

tbb\_solution.cpp

|  |  |
| --- | --- |
| Класс TBBEvenSplitter | Выполняет слияние четных элементов массивов |
| Класс TBBOddSplitter | Выполняет слияние нечетных элементов массивов |
| Класс TBBSimpleComparator | Выполняет сравнение четных и нечетных пар элементов массива, проходя по нему один раз. |
| void RadixSort(double\* data, int dataLength) | Функция выполняет алгоритм восходящей поразрядной сортировки LSD Radix Sort. |
| Класс TBBParallelSorter | Реализует рекурсивный алгоритм слияния, выполняя последовательную сортировку в случае, когда размер части массива меньше, чем размер исходного массива, деленного на количество процессов. |
| void TBBParallelSort(double \*input, int size, int nThreads) | Функция инициализации параллельной сортировки с использованием TBB.  Для сортировки массива необходимо вызвать данную функцию. |

generator.cpp

|  |  |
| --- | --- |
| int main(int argc, char\* argv[]) | Основная функция программы. В ней происходит генерация массивов размером от 1000 до 50 млн. элементов в диапазоне [-10000, 10000] . |

before\_code.cpp

|  |  |
| --- | --- |
| int main(int argc, char\* argv[]) | Основная функция программы. В ней происходит считывание исходного массива из файла, сортировка с помощью последовательного и параллельного алгоритма и запись полученных массивов в файлы . |

checker.cpp

|  |  |
| --- | --- |
| int main(int argc, char\* argv[]) | Основная функция программы. В ней происходит сравнение массивов, полученных при разных вариантах сортировки . |

## Проверка корректности

Проверка корректности параллельных методов производится в файле checker.cpp.

Из файла номер\_теста.ans считывается массив, полученный в результате последовательного метода сортировки.

Из файла номер\_теста\_par.ans считывается массив, полученный в результате параллельного метода сортировки.

Затем производится поэлементное сравнение массивов и результат проверки записывается в файл result.txt. Также в файл записывается время работы каждого из алгоритмов.

# Результаты экспериментов

# Эксперименты проводились на компьютере со следующими техническими характеристиками:

* Процессор Intel Core i7 7700HQ (4 ядра).
* Оперативная память 12 ГБ.

Выполнялось тестирование Release версии программы.

Время экспериментов измеряется в секундах, n – размер массива. Посл. Время – время работы последовательного алгоритма. proc () – время работы параллельного алгоритма на процессах. Первая строка – тесты OpenMP, вторая – TBB.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Входные данные  n | 1 proc (с) | 2 proc (с) | 4 proc (с) | 8 proc (с) |
| 1 | 1 млн. | 0.0438011 | 0.0387016 | 0.0348459 | 0.0369945 |
|  |  |  | 0.0347704 | 0.0328792 | 0.0307214 |
| 2 | 10 млн. | 0.451383 | 0.394086 | 0.359959 | 0.436929 |
|  |  |  | 0.389301 | 0.327723 | 0.36457 |
| 3 | 30 млн. | 1.33083 | 1.1219 | 1.05737 | 1.2548 |
|  |  |  | 1.06465 | 0.967528 | 1.04755 |
| 4 | 50 млн. | 2.22227 | 1.80798 | 1.75963 | 2.02076 |
|  |  |  | 1.73974 | 1.65263 | 1.79049 |

Время экспериментов измеряется в секундах, n – размер массива. proc Ускорение () – ускорение работы алгоритма на i процессах относительно времени работы на 1 процессе.

Первая строка – тесты OpenMP, вторая – TBB.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Входные данные  n | 1 proc | 2 proc  Ускорение | 4 proc  Ускорение | 8 proc  Ускорение |
| 1 | 1 млн. | 1 | 1.1317 | 1.2569 | 1.1839 |
|  |  |  | 1.2597 | 1.3321 | 1.4257 |
| 2 | 10 млн. | 1 | 1.1453 | 1.2539 | 1.033 |
|  |  |  | 1.1594 | 1.3773 | 1.2381 |
| 3 | 30 млн. | 1 | 1.1862 | 1.2586 | 1.06 |
|  |  |  | 1.25 | 1.3754 | 1.2704 |
| 4 | 50 млн. | 1 | 1.229 | 1.2629 | 1.099 |
|  |  |  | 1.2773 | 1.3446 | 1.2411 |

# Вывод

В результате выполнения работы была написана программа, выполняющая сортировка сгенерированного массива с помощь алгоритма поразрядной сортировки со слияние Бэтчера для вещественных чисел в последовательной и двух параллельных реализациях.

Было замерено время их работы и на основе этих результатов можно сделать следующие выводы:

* Имеет смысл применять параллельную версию поразрядной сортировки при больших значениях N и при достаточно малом количестве процессов (например, использовать 4 процесса при сортировке 107 элементов типа double).
* Ускорение при количестве процессов, большее 4-х, будет либо слишком мало. Это связано с тем, что сортировка LSD Radix Sort работает за линейное время от размера массива. В этом случае, при распараллеливании этого метода сортировки, процессам будет даваться небольшая работа (т.е. часть массива они отсортируют достаточно быстро), а время работы сортировки в параллельной версии будет существенно зависеть от механизмов взаимодействия между процессами и от самой схемы распараллеливания. При достаточно большом количестве процессов механизмы взаимодействия между ними не окупаются, а наоборот вредят.