Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

Институт Информационных Технологий, Математики и Механики

**Поразрядная сортировка для целых чисел с простым слиянием.**

Выполнил: студент группы 381608

Бедердинов Д.Д.

Проверил: ассистент каф. МОСТ, ИИТММ

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

Введение **3**

Постановка задачи **3**

Метод решения **4**

Представление целых в памяти компьютера  **6**

Описание последовательного алгоритма **7**

Схема распараллеливания **8**

Описание реализации параллельного алгоритма  **9**

Подтверждение корректности  **10**  Результаты экспериментов по оценке масштабируемости  **11**

Заключение **13**

**Введение**

Сортировка (англ. sorting — классификация, упорядочение) — последовательное расположение или разбиение на группы чего-либо в зависимости от выбранного критерия.

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений S = {a1, a2, ..., an} в порядке монотонного возрастания или убывания S ~ S' = {(a1', a2’, ..., an'): a1'<= a2' <=... <= an'}.

Возможные способы решения этой задачи широко обсуждаются в литературе. В данной работе рассмотрен метод поразрядной восходящей сортировки.

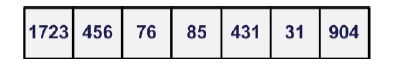
**Постановка задачи**

Дан неупорядоченный массив, состоящий из n положительных и отрицательных элементов типа int. Требуется реализовать последовательную и параллельную версию алгоритма восходящей поразрядной сортировки LSD Radix Sort для целых чисел и сравнить их время работы при различных значениях размера сортируемого массива.

**Метод решения**

Идея поразрядной восходящей сортировки LSD Radix Sort заключается в том, что выполняется последовательная сортировка чисел по разрядам (от младшего разряда к старшему).

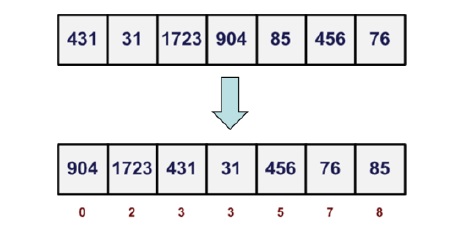
Рассмотрим алгоритм работы сортировки на примере массива десятичных чисел из 7 элементов:



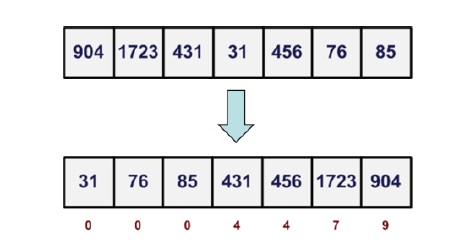
На первой итерации сортировки выполняется размещение элементов в отсортированном порядке по младшему разряду чисел:

# 

На следующей итерации сортируются элементы по второму разряду:



Далее выполняется упорядочивание элементов по третьему разряду:

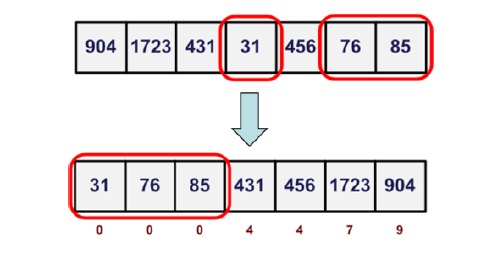


На последнем шаге выполняется сортировка по старшему разряду:

# 

В итоге получим отсортированный массив.

Поразрядная сортировка массива будет работать только в том случае, если сортировка, выполняющаяся по разряду, является устойчивой (элементы равных разрядов не будут менять взаимного расположения при сортировке по очередному разряду):



Современные процессоры предназначены для обработки данных, которые представлены в битах и байтах, поэтому выполнять сортировку по десятичным разрядам чисел не эффективно. Будем рассматривать побайтовую реализацию поразрядной сортировки, т.е. будем рассматривать число как набор 256-значных цифр. В таком случае для сортировки разряда будет удобно использовать сортировку подсчётом (с модификацией, которая не будет менять взаимного расположения элементов равных разрядов).

**Представление целых чисел в памяти компьютера**

Так как мы будем рассматривать побайтовую реализацию сортировки для элементов типа int, то покажем, как хранятся целые числа в памяти компьютера.

**Формат целочисленных типов**

Целочисленные типы без знака представляются в памяти таким образом (рассмотрим 4-байтовый тип ― «unsigned int»), что число x кодируется следующим образом:

Целочисленный тип со знаком кодируется таким образом, что при сложении равных по модулю отрицательного и положительного чисел получается нуль.

Кроме того, старший бит определяет знак числа:

* если старший бит равен нулю, то число положительное;
* если старший бит равен единице, то число отрицательное.

**Формат целых отрицательных чисел**

Целочисленные типы со знаком представляются в памяти таким образом (рассмотрим 4-байтовый тип ― «signed int»), что число x кодируется следующим образом:

При таком представлении отрицательные числа кодируются в так называемом дополнительном коде:

* записывается равное по модулю положительное число;
* выполняется инвертирование всех бит этого числа;
* к полученному значению добавляется ―1 по правилам сложения чисел без знака.

**Описание последовательного алгоритма**

Рассмотрим алгоритм для целых положительных чисел.

Сортировка по i-му байту будет проходить в два прохода:

1. при первом проходе по исходному массиву выполняется подсчёт i-ых байт в массиве, результат будет сохранён в массив count из 256 элементов.
2. в массив offset, на основании посчитанных данных, выполняется подсчёт индексов, по которым будут сохраняться элементы:

offset [0] = 0

для всех j от 1 до 256

offset [j] = offset [j - 1] + count [ j - 1]

1. при втором проходе по исходному массиву выполняется копирование элемента во вспомогательный массив по соответствующему индексу в массиве подсчётов, выполняется инкремент индекса.
2. Из вспомогательного массива все копируются в исходный массив.

Для отрицательных чисел алгоритм не изменится. Старший бит знаковых типов определяет знак числа, поэтому сортировку нужно выполнять с учётом того, что числа, у которых старший бит равен 0 (положительные) больше тех чисел, у которых старший бит равен 1 (отрицательные). Исходя из этого, получается, что при сортировке знаковых чисел необходимо изменить алгоритм сортировки только по старшему байту. Для этого достаточно в сортировке интерпретировать старший байт, как знаковое число (диапазон знаковых однобайтовых чисел от -128 до 127) и прибавлять смещение 128. Таким образом, в массиве подсчётов по индексу 0 будет находиться количество элементов массива, у которых в старшем байте минимальное число со знаком (-128).

**Схема распараллеливания**

Одними из параллельных алгоритмов являются те, которые имеют внутренний параллелизм. Рассмотрим общую схему одного из таких алгоритмов:

1. Все процессы должны обменяться массивами подсчётов. Создаётся общий массив подсчетов и массив смещений.

2. Массив смещений делится на диапазоны. После этого каждый процесс оставляет у себя только элементы из одного определённого диапазона, элементы из остальных диапазонов отправляются соответствующим процессам.

3. Каждый процесс размещает элементы из своего диапазона в отсортированном порядке относительно массива смещений.

4. Повторить все 3 шага 4 раза и в итоге получим отсортированный массив по всем 4 байтам.

**Описание реализации параллельного алгоритма**

Рассмотрим реализацию схемы распараллеливания:

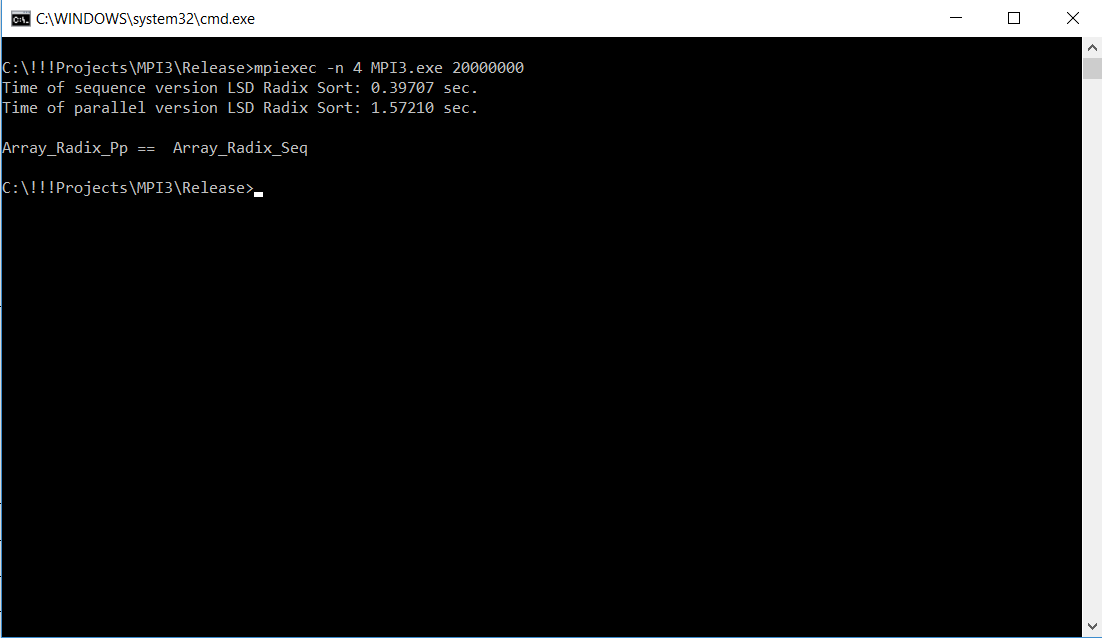
* Генерируется массив из N случайных элементов типа int процессом с рангом 0;
* Процесс 0 сообщает всем другим процессам о размере массива (с помощью функции MPI\_Bcast);
* Каждым процессом вычисляется массив работ и смещений.

Цикл по i от 0 до 3 (для каждого бита числа):

* При помощи использования массивов работ и смещений, процесс с рангом 0 рассылает соответствующие части исходного массива всем другим процессам и себе тоже (с помощью функции MPI\_Scatterv);
* Каждый процесс выполняет подсчёт элементов в свой массив подсчётов;
* Вычисляется общий массив подсчётов путем сложения массивов подсчётов всех процессов (с помощью функции MPI\_Allreduce);
* На основе общего массива подсчётов вычисляется массив смещений offset;
* Делим общий массив подсчётов на количество всех процессов. Каждый процесс создает несколько массивов, которые будут содержать элементы с байтами из соответствующих процессам диапазонов. Таких массивов будет создано столько, сколько всего процессов;
* Каждый процесс оставляет свой массив, другие массивы он отправляет соответствующим процессам (с помощью функции MPI\_Sendrecv);
* Каждый процесс объединяет свой и полученные от других процессов массивы в один массив;
* На основе массива смещений выполняется размещение элементов в соответствующие позиции;
* Происходит слияние частей массивов в один процесс 0 (с помощью функции MPI\_Gatherv).

**Подтверждение корректности**

Подтверждение корректности работы параллельной версии алгоритма выполняется процессом 0: производится сравнение полученных в параллельной версии результатов с версией последовательной. На рисунке ниже показано, что сортируется массив размером 20 000 000 за линейное время:



**Результаты экспериментов по оценке масштабируемости**

Исследования производились на массиве размера 10000000, 15000000 и 20000000, состоящие из отрицательных и положительных чисел:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 процесс  T, с | Параллельный алгоритм | | | | | | | | | |
| 2 процесса | | 4 процесса | | 8 процессов | | 16 процессов | | 32 процесса | |
| T, с | Acs | T, с | Acs | T, с | Acs | T, с | Acs | T, с | Acs |
| 10000000 | 0.20 | 0.85 | 0.23 | 0.77 | 0.25 | 1.20 | 0.16 | 2.97 | 0.06 | 12.25 | 0.01 |
| 15000000 | 0.30 | 1.33 | 0.22 | 1.12 | 0.26 | 1.67 | 0.17 | 3.51 | 0.08 | 12.22 | 0.02 |
| 20000000 | 0.40 | 1.65 | 0.24 | 1.57 | 0.25 | 2.21 | 0.18 | 4.17 | 0.09 | 13.93 | 0.02 |

Построим график зависимости ускорения от числа процессов:

Сравним полученный путём экспериментов график с графиком, полученным с помощью анализа сложности алгоритма.

Сложность последовательного алгоритма **O(n) –** линейный алгоритм.

Сложность параллельного алгоритма **O()**:

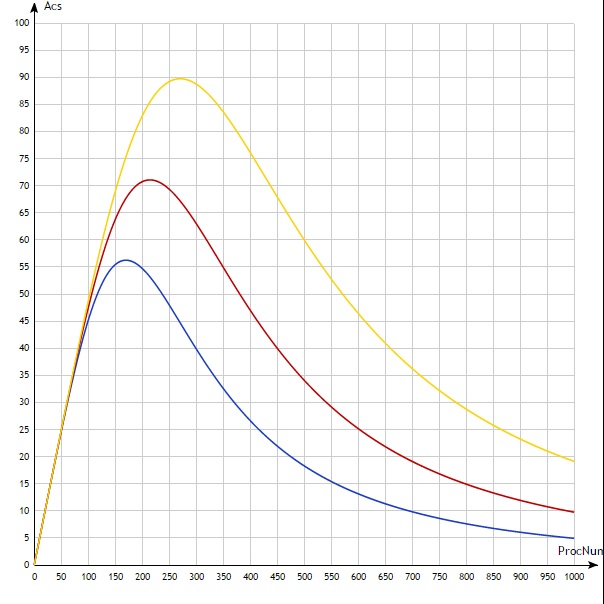
* Подсчёт элементов каждым процессом в свой массив подсчета за ;
* Создание каждым процессом дополнительных массивов за p;
* Определение размера для принимающего буфера за p;
* Распределение элементов каждый процессом по соответствующим массивам за ;
* Создание буферов, которые будут отправляться соответствующим процессам за p;
* Объединение каждым процессом полученных от других процессов буферов со своим буфером в один за p;
* Размещение элементов в соответствующие позиции за ;
* Подсчёт смещений для слияния частей массивов в один за p;
* Количество вызовов коммутативных операций p2

Ускорение **O().** Построим график для этой функции для выбранных n.

n = 10000000

n = 15000000

n = 20000000

**

Общий вид графиков приблизительно совпадают.

**Заключение**

Были реализованы последовательная и параллельная версия LSD Radix Sort для целых чисел. Было замерено время их работы. Следует заметить, что последовательная версия работает быстрее параллельной версии во всех случаях. Время работы сортировки в параллельной версии зависит от механизмов взаимодействия между процессами и от самой схемы распараллеливания. В рассмотренной схеме распараллеливания механизмы взаимодействия между процессами не окупаются, а наоборот вредят и с учётом того, что сортировка линейная, в нашем случае последовательная версия быстрее параллельной.