ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

РАЗРАБОТКА АЛГОРИТМОВ СОРТИРОВКИ ДАННЫХ БОЛЬШОГО ОБЪЕМА С ИСПОЛЬЗОВАНИЕМ ПАРАЛЛЕЛЬНЫХ ТЕХНОЛОГИЙ

Магистерская диссертация
обучающейся по направлению подготовки 02.04.01 Математика и компьютерные науки
очной формы обучения, группы 07001531
Абакумовой Алины Сергеевны

Научный руководитель
к.т.н., доцент
Михелев В.М.

Рецензент
доцент кафедры информационных технологий
Жихарев А.Г.

БЕЛГОРОД 2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время компьютерные системы имеют огромны вычислительные возможности. Эти системы моментально могут выполнить объемы вычислений, которые одному человеку потребовалось бы очень много времени. В связи с возрастанием производительности компьютерной техники, повышаются сложности решаемых задач, расширяются круги исследуемых проблем.

Алгоритмы сортировок — одна из самых популярных тем в программировании. Однако ее обсуждение обычно сводится к тому, что быстрее сортировки Хоара (или, как ее называют, быстрой сортировки) еще не придумано. Но в современном мире требования к точности и скорости решения таких задач постоянно возрастают.

Алгоритмы сортировки данных широко обсуждаю в литературе. Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой, поэтому с каждым годом появляется необходимость все более усовершенствовать этот процесс. Особенно остро становится эта проблема при использовании Big Data или Большие Данные. Это понятие уже давно у всех на слуху. Но не каждый точно знает, что же представляет собой это понятие.

Появление «больших данных» и высокопроизводительных вычислений позволяют затрагивать тему повышения эффективности работы алгоритмов и делает тему магистратской работы актуальной.

Есть много тем для решения данной проблемы, которые еще не полностью изучены, но имеются уже другие подходы к разработке алгоритмов, например, параллельное программирование.

В данной магистерской диссертации будут изучаться различные алгоритмы сортировки данных. Излагаются как общие принципы, применяемые при распараллеливании, так и конкретные алгоритмы.

Были поставлены следующие конкретные цели: разработка и исследование алгоритмов сортировки данных большого размера с использованием параллельных технологий.

Для достижения поставленной цели решаются следующие задачи:

- изучение и исследование существующих методов сортировки данных;
- изучение и термина «BIG DATA»;
- изучение параллельных технологий;
- сравнительный анализ существующих алгоритмов сортировки данных для больших данных;
- разработка последовательного и параллельного алгоритмов сортировки данных;
-оптимизация алгоритмов сортировки для данных большого объема.

ГЛАВА 1 ОБЗОР И АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

Главной целью магистерской диссертации – анализ и разработка алгоритмов сортировки данных большого объема с использованием параллельных технологий. Для достижения поставленной цели необходимо изучить существующие методы алгоритмов сортировки.

После изучения известных алгоритмов, необходимо провести анализ и выявить алгоритмы которые подходят для использования больших данных. Также есть много тем, которые еще не полностью изучены, но имеются уже другие подходы к разработке алгоритмов, например, параллельное программирование.

1.1. Исследование существующих методов сортировки данных и их классификация

Алгоритм – это формально описанная вычислительная процедура, которая берет исходные данные, называемые его аргументом, и выдающая результат вычислений на выход. Алгоритмы необходим для решения тех или иных вычислительных задач [8].

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = a_1, a_2, \ldots, a_n \quad (1.1)$$

в порядке монотонного возрастания или убывания:

$$S \sim S' = (a'_1, a'_2, \ldots, a'_n) : a'_1 \leq a'_2 \leq \ldots \leq a'_n \quad (1.2)$$

Вычислительная трудоемкость алгоритмов сортировки данных является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций находятся квадратичной зависимостью от числа упорядочиваемых данных $T \sim n^2$.

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстрая сортировка) трудоемкость определяется величиной: $T \sim n \log_2 n$.

Ускорение сортировки может быть обеспечено при использовании нескольких ($p>1$) процессоров. Исходный упорядочиваемый набор в этом случае разделяется между процессорами; в ходе сортировки данные пересылаются между процессорами и сравниваются между собой. Результирующий (упорядоченный) набор, как правило, также разделен между процессорами; при этом для систематизации такого разделения для процессоров вводится та или иная система последовательной нумерации и обычно требуется, чтобы при завершении сортировки значения, располагаемые на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами [8].

Оставляя подробный анализ проблемы сортировки для отдельного рассмотрения, здесь основное внимание мы уделим изучению параллельных способов выполнения для ряда широко известных методов внутренней сортировки, когда все упорядочиваемые данные могут быть размещены полностью в оперативной памяти ЭВМ [20].

Методы сортировки важны при обработке данных, с ними связаны многие фундаментальные приемы построения алгоритмов.

Сортировки могут быть выполнены с использованием различных алгоритмов: как простых, так и усложненных (но более эффективных). Основное требование к методам сортировки: экономное использование памяти и быстродействие. Первое требование может быть выполнено, если переупорядочение элементов будет выполняться на том же месте [24].

Существует множество алгоритмов сортировки.

Устойчивая (стабильная) сортировка — иногда называют стабильная сортировка, это означает что она не меняет относительного порядка сортируемых элементов, с одинаковыми ключами [8].

Алгоритмы устойчивой сортировки:

1. Сортировка пузырьком (Bubble sort).
2. Сортировка перемешиванием (Cocktail sort)
3. Гномья сортировка(Gnome sort).
4. Сортировка вставками (Insertion sort)
5. Сортировка слиянием (Merge sort)
6. Сортировка с помощью двоичного дерева (англ. Tree sort).
7. Сортировка Timsort (англ. Timsort)
8. Сортировка подсчётом (Counting sort).
9. Блочная сортировка (Bucket sort)
10. Поразрядная сортировка (она же цифровая сортировка) [29]

Исходя из определения устойчивой сортировки, можно прийти что неустойчивая сортировка – это такая сортировка, при которой сортируемые элементы могут менять порядок [24].

1. Алгоритмы неустойчивой сортировки;
2. Сортировка выбором (англ. Selection sort) ;
3. Сортировка расчёской (Comb sort);
4. Сортировка Шелла (Shell sort);
5. Пирамидальная сортировка (сортировка кучи, Heapsort);
6. Плавная сортировка (Smoothsort);
7. Быстрая сортировка (Quicksort);
8. Интроспективная сортировка (Introsort);
9. Терпеливая сортировка (Patience sorting);

Рассмотрим некотрые методы более подробно. [29]

Сортировка пузырьком (Bubble sort).

Данный алгоритм ссортировки многим известный и часто используется, так как является довольно простым. Его суть заключается, в том, что в ходе прохождения алгоритма меняются местами два соседних элемента, если первый элемент массива больше второго. Процесс выполняется до тех пор, пока алгоритм не обменяет местами все неотсортированные элементы. Сложность выполнения данного алгоритма равна O(n^2) [35].

Для рассмотрения сортировки на рис. 1.1. приведен пример, на котором видно, что в процессе работы алгоритм выполняет проходы по массиву и меняет местами элементы, нарушающих порядок [24].

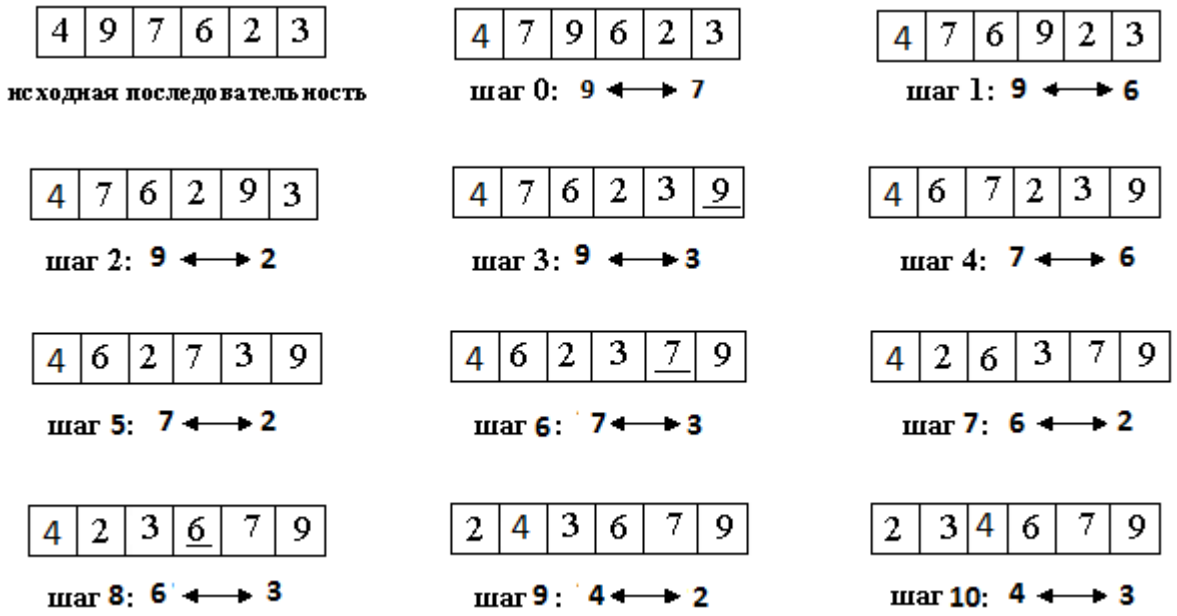

Рис. 1.1. Пример сортировки пузырьком

Сложность такого алгоритма равно $O(n^2)$ [24].

Для определения сложности алгоритма, определим сколько раз выполняется алгоритм, внешний цикл выполняется n-1 раз, а внутренний – в среднем n/2 раз.

Следовательно, сортировка требует $\frac{(n^2-n)}{2}$ сравнений. Так можно узнать, что этот алгоритм порядка n2, и это является недостатком, т.к. из-за этого он

считается слишком медленным, если использовать массив большим количеством элюентов [35].

Сортировка перемешиванием (Cocktail sort).

Другое название данной сортировки - Шейкерная сортировка, и она является из разновидностей пузырьковой сортировки [13].

При анализе метода пузырьковой сортировки, можно выделить два обстоятельства, которые влияют на появление модификации:

- Если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения.
- При движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо [8].

Эти два обстоятельства приводят к данной модификации в методе пузырьковой сортировки. Границы рабочей части массива (то есть части массива, где происходит движение) устанавливаются в том месте, где был последний обмен на каждой итерации. Массив рассматривается справа налево и слева направо поочередно. Сложность алгоритма $O(n^2)$ [8].

Гномья сортировка (Gnome sort).

Гномья сортировка — еще один алгоритм сортировки. Данный алгоритм похож на сортировку вставками. Главное отличие от сортировки вставками перед установкой элемента на нужное место происходит серия обменов, как в сортировке пузырьком [13].

Входе выполнения алгоритма он находит два соседних элемента, которые стоят в неправильном порядке и меняет их метами. Он пользуется тем фактом, что обмен может породить новую пару, стоящую в неправильном порядке, только до или после переставленных элементов. После каждой такой итерации,

элементы после текущей позиции отсортированы, таким образом, нужно только проверить позицию до переставленных элементов [14].

Пример сортировки приведен ниже. Если мы хотим отсортировать массив с элементами [4] [2] [7] [3] от большего к меньшему, то на итерациях цикла while будет происходить следующее:

[4] [2] [7] [3] (начальное состояние: i == 1, j == 2);

[4] [2] [7] [3] (ничего не произошло, но сейчас i == 2, j == 3);

[4] [7] [2] [3] (обмен a[2] и a[1], сейчас i == 1, а j == 3 по-прежнему);

[7] [4] [2] [3] (обмен a[1] и a[0], сейчас i == 3, j == 4);

[7] [4] [3] [2] (обмен a[3] и a[2], сейчас i == 2, j == 4);

[7] [4] [3] [2] (ничего не произошло, но сейчас i == 4, j == 5);

цикл закончился, т. к. i не < 4.

Время работы выполнения работы алгоритма - $O(n^2)$[13].

Сортировка вставками (Insertion sort).

Данный алгоритм является довольно известным. Его суть в том, что алгоритм сортирует массив по мере прохождения по его элементам. На каждой итерации берется элемент и сравнивается с каждым элементом в уже отсортированной части массива. После он находим подходящие место, после и вставляет элемент на свою позицию. Алгоритм выполняется до тех пор, пока он не пройдет по всему массиву. На выходе получим отсортированный массив. Сложность данного алгоритма равна O(n^2) [20].

Общая суть проста, на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем, как показано на рис. 1.2 приведен пример данной сортировки. Стоит отметить что массив из 1-го элемента считается отсортированным [21].

Рис. 1.2. Пример сортировки вставками

Сложность алгоритма вставками равна $O(n^2)$ [36].

При сравнении алгоритма вставками, с алгоритмом сортировки выбором, то отличие главное отличие исходный порядок ключей во входных данных. Например, если файл большой, а ключи уже упорядочены (или почти упорядочены), то сортировка вставками выполняется быстро, а сортировка выбором — медленно [24].

Сортировка слиянием (Merge sort).

Сортировка слиянием—один из видов алгоритмом сортировки, который упорядочивает списки в определённом порядке.

Списки - это абстрактный тип данных, который представляет собой набор значений, в котором может значения встречать более одного раза и эти значения определённым образом упорядочены [24].

К сортировке слиянием можно применить принцип «разделяй и властвуй». Основной смысл сортировки в том, что сначала последовательность разбивается на несколько маленьких подзадач. Затем каждая подзадача решается отдельно с помощью рекурсивного метода. После решения соединяются вместе и получается решение исходной большой задачи [8]. Пример сортировки приведен на рис. 1.3.

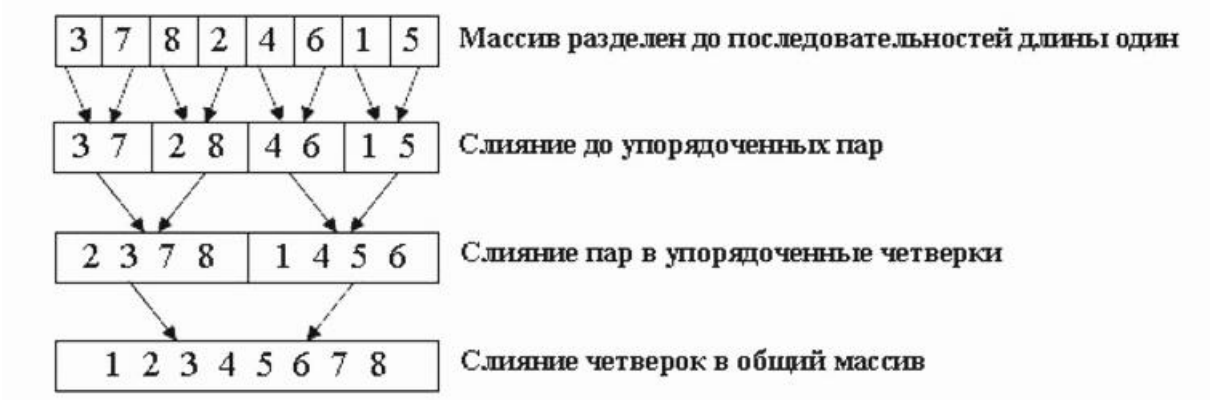

Рис. 1.3. Пример сортировки слиянием

Что бы оценить время работы алгоритма необходимо разобраться в формулах. Примем $T(n)$ — время выполнения алгоритма сортировки длины n, тогда для сортировки слиянием справедливо

$$T(n) = 2T(n/2) + O(n) \tag{1.5}$$

$O(n)$ — время, необходимое на то, чтобы слить два массива [8].

При рассмотрении достоинств и недостатков данного метода, выделяют:

Достоинства:

1.Есть возможность написать эффективную многопоточную программу сортировки;

2.При нехватке оперативной памяти, можно сохранять алгоритм сортировки на периферийных устройствах.

К недостаткам относится:

3. Требуется дополнительно $O(n)$ памяти, но можно модифицировать до $O(1)$

4. При любых входных данных время работы — $O(n)$ [20].

Сортировка с помощью двоичного дерева (Tree sort).

Сортировка с помощью двоичного дерева (*tree sort*) — алгоритм сортировки, на основе которого строится двоичное дерево поиска по ключам массива (списка), далее строится результирующий массив путём обхода узлов дерева в необходимом порядке следования ключей [8].

Для более подробного рассмотрения сортировки, рассмотрим пример, дана последовательность:

4, 3, 5, 1, 7, 8, 6, 2

Корнем дерева считается начальный элемент последовательности от него и будет строиться дерево. Далее проверяются все элементы, и если элемент меньшие корневого, то он располагаются в левом поддереве, а элементы, которые больше корневого, записываются в правом поддереве. Так выполняется на каждом уровне дерева и строится вид дерева. На рис. 1.4 рассмотрен пример двоичного дерева [32].

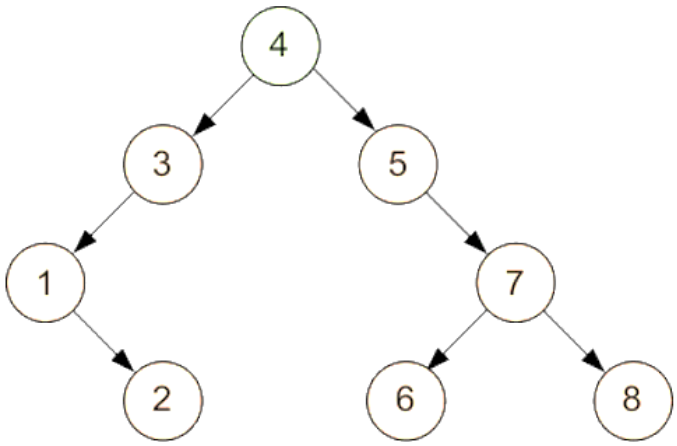

Рис. 1.4. Пример сортировки с помощью двоичного дерева

Средняя алгоритмическая сложность алгоритма составляет O(log(n)). Для n объектов сложность будет составлять O(n log(n)), это относит сортировку к группе «быстрых сортировок». Однако, сложность добавления объекта в разбалансированное дерево может достигать O(n), что может привести к общей сложности порядка $O(n^2)$ [32].

Если данные необходим представить в виде дерева, данная сортировка может быть очень удобна. Но недостаток метода сортировки - требуется много памяти [35].

Сортировка Timsort (Timsort).

Данный алгоритм сортировки является гибридным, включает в себя сортировку вставками и сортировку слиянием. Он был опубликованный в 2002 году Тимом Петерсом [35].

Основная идея алгоритма в том, что часто втречаются уже отсортированные массивы и их удобнее использовать, так как они содержат в себе уже упорядоченные подмассивы [8].

Основные этапы алгоритма включают:

1. Входной массив определенным образом разбивается на подмассивы.
2. Каждый подмассив сортируется с помощью сортировки вставками.
3. Далее каждый отсортированный подмассив объединяются вместе в единый массив с помощью модифицированной сортировки слиянием.

Сложность сортировки равно $O(n \log n)$, но требуется дополнительная память $O(n)$ [35].

Сортировка подсчётом (Counting sort).

Сортировка подсчётом— алгоритм сортировки, в котором для подсчета совпадающих элементов используется диапазон чисел сортируемого. Применять алгоритм сортировки целесообразно, когда сортируемые числа имеют диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством. Используется при небольших данных [36].

Рассмотрим алгоритм сортировки, последовательность имеет длин n, а в конце в массиве A. Также используется вспомогательный массив C с индексами от 0 до $\mathrm{k}-1$, и изначально заполняется 0.

1. Последовательно пройдём по массиву A и запишем в $C[i]$ количество чисел, равных i.

2. Теперь достаточно пройти по массиву C и для каждого $number \in \{0, \ldots, \mathrm{k}-1\}$ в массив A последовательно записать число $number$ $C[number]$ раз.

Этот тип сортировки может обладать линейным временем исполнения, поэтому есть некоторые модификации данного алгоритма [36].

Блочная сортировка (Bucket sort).

Блочная сортировка— алгоритм сортировки, в котором сортируемые элементы распределяются между конечным числом отдельных блоков (карманов, корзин) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем. Далее сортируется отдельно каждый блок. После сортировки все элементы помещаются обратно в массив. Этот тип сортировки может обладать линейным временем выполнения [29].

Основная идея алгоритма блочной сортировки в том, чтобы разбить отрезок [0,1) на *n* одинаковых отрезков (карманов), и далее входные элементы разбить по этим карманам. Затем последовательно сортируются числа в карманах. Отсортированный массив получается путём последовательного перечисления элементов каждого кармана [29].

Возьмём исходной массив 100, 97, 3, 28, 15. Разделяем числа по блокам, исходя из разряда единиц, как показано на рис. 1.5.

Рис. 1.5. Первая итерация блочной сортировки

Копируем числа обратно справа налево по очереди в исходный массив. Пример показан на рис. 1.6. Теперь исходный массив: 100, 3, 15, 97, 28

Рис. 1.6. Вторая итерация блочной сортировки

Копируем числа обратно справа налево по очереди в исходный массив. Данная итерация показана на рис. 1.7. Теперь массив выглядит следующим образом: 100, 3, 15, 28, 97

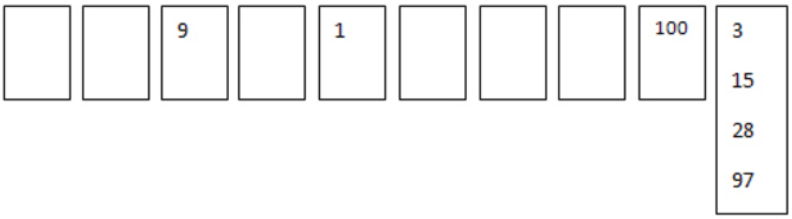

Рис. 1.7. Третья итерация блочной сортировки

Копируем числа обратно справа налево по очереди в исходном массив. В итоге массив отсортирован [32].

При анализе метода можно выделить преимущества и недостатки. Главным преимуществом является линейное время исполнения O(N) (на удачных входных данных).

К недостаткам: при использовании похожих элементов сильно. В некоторых таких случаях для строк, возникающих в реализациях основанного на сортировке строк алгоритма сжатия BWT, оказывается, что быстрая сортировка строк значительно превосходит блочную сортировку по сравнению скорости [35].

Поразрядная сортировка.

Суть алгоритма заключается в том, что массив несколько раз перебирается по разрядам и элементы перегруппировываются в зависимости от того, какая цифра находится в определённом разряде. При этом разряды могут обрабатываться в противоположных направлениях — от младших к старшим или наоборот [35].

Бывает два вида сортировки. Сортировка по младшим разрядам и по страшим разрядам.

Допустим у нас есть числа: 39, 47, 54, 59, 34, 41, 32

1. Создаем пустые списки, количество которых равно числу range.

2. Распределяем исходные числа по этим спискам в зависимости от величины младшего разряда (по возрастанию).

Для нашего примера получим:

41

32

54, 34

47

59, 39

3. Собираем числа в той последовательности, в которой они находятся после распределения по спискам.

Получим: 41, 32, 54, 34, 47, 59, 39

4. Повторяем пункты 2 и 3 для всех более старших разрядов поочередно. Для двузначных чисел мы должны сделать еще один проход. Распределение по спискам будет выглядеть так:

32, 34, 39

41, 47

54, 59

Объединив числа в последовательность, получим отсортированный массив [35].

Сортировка выбором (англ. Selection sort).

Суть алгоритма заключается в проходе по массиву от начала до конца в поиске минимального элемента массива и перемещении его в начало [36].

При рассмотрении алгоритма более подробно, сначала определяется наименьший элемент массива и меняют его местами с элементом, стоящим первым в сортируемом массиве. Далее находится второй наименьший элемент и меняют его местами с элементом, который стоит вторым в исходном массиве. Алгоритм выполняется до тех пор, пока весь массив не будет отсортирован. На рис. 1.8. рассмотрим пример для наглядности алгоритма [29].

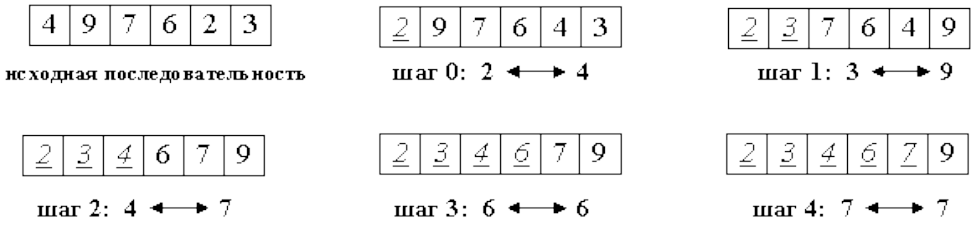

Рис. 1.8. Пример алгоритма сортировки выбором

Сложность такого алгоритма O(n2).

При сравнении с другими методами сортировки, недостатком сортировки выбором будет то, что время ее выполнения почти не зависит от упорядоченности исходного файла. При рассмотрении массива и нахождении минимального элемента за один проход дает мало сведений для поиска минимального элемента при преследующем поиске элемента. Хоть и алгоритм является простым и, сортировка выбором является совершенным методом при случае: когда элементы очень велики, а ключи очень малы [32].

Сортировка Шелла (Shell sort).

Идея работы алгоритма крайне похожа на сортировку обменом, но главным отличием является то, что сравниваются не два соседних элемента, а элементы на промежутке, к примеру, в пять элементов. Это обеспечивает от избавления мелких значений в конце, что способствует ускорению сортировки в крупных массивах [32].

В литературе опубликованы результаты исследований различных последовательностей шагов, некоторые из них хорошо зарекомендовали себя на практике, однако возможно, что наилучшая последовательность еще не найдена. Обычно на практике используются убывающие последовательности шагов, близкие к геометрической прогрессии, так что число шагов логарифмически зависит от размера файла. [25]

Лемма 1 В результате h-сортировки k-упорядоченного файла получается h- и k-упорядоченный файл.

Лемма 2. Сортировка Шелла выполняет менее N(h — 1)(k — 1)/g сравнений при g-сортировке h- и k-упорядоченного файла, если h и k взаимно просты.

Лемма 3. Сортировка Шелла выполняет менее
O(N3/2) сравнений для последовательности шагов
1 4 13 40 121 364 1093 3280 9841 . . .

Для больших шагов имеются h подфайлов размером N/h, и в худшем случае трудоемкость равна примерно N2/h . При малых шагах из леммы 6.8 следует, что трудоемкость составляет приблизительно Nh.

Лемма 4. Сортировка Шелла выполняет менее O (N4/3) сравнений для последовательности шагов 1 8 23 77 281 1073 4193 16577 . . .

Лемма 5. Сортировка Шелла выполняет менее O(N (logN)2) сравнений для последовательности шагов 1 2 3 4 6 9 8 12 18 27 16 24 36 54 81 . . .
Число шагов из треугольника, которые меньше N, определенно меньше (log2N)2. [32]

Пирамидальная сортировка (Heapsort).

Еще один алгоритм сортировки — это пирамидальная сортировка. Анализ сложности $\Theta(n \log n)$ операций при сортировке n элементов. Количество применяемой служебной памяти не зависит от размера массива (то есть, O(1)).

Рассматривается как усовершенствованная сортировка пузырьком, в которой элемент всплывает (min-heap) / тонет (max-heap) по многим путям [20].

Сортировка пирамидой использует бинарное сортирующее дерево. Основными этапами алгоритма являются условия:

1. Каждый лист имеет глубину либо d, либо d-1, при этом d — максимальная глубина дерева.

2. Значение в любой вершине не меньше (либо при другом варианте не больше) значения её потомков.

Алгоритм сортировки будет состоять из двух основных шагов:

Выстраиваем элементы массива в виде сортирующего дерева:

$$\text{Array}[i] \geq \text{Array}[2i+1]$$
$$\text{Array}[i] \geq \text{Array}[2i+2]$$
$$\text{при } 0 \leq i < n/2. \qquad \textbf{(1.6)}$$

Этот шаг требует $O(n)$ операций.

На следующем шаге удаляются элементы из корня по одному за раз и перестраивается дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда Array[1], Array[2], …, Array[n] — упорядоченная последовательность.

К достоинства относится сложность алгоритма $\Theta(n \log n)$, и требуемая память. К недостаткам Сложность реализации, неустойчивость, на одном шаге выборку приходится делать хаотично по всей длине массива [28].

Быстрая сортировка *(Quicksort).*

Суть алгоритма заключается в разделении массива на два под-массива, средней линией считается элемент, который находится в самом центре массива. В ходе работы алгоритма элементы, меньшие чем средний будут перемещены в лево, а большие в право. Такое же действие будет происходить рекурсивно и с под-массива, они будут разделяться на еще два под-массива до тех пор, пока не будет чего разделать (останется один элемент). На выходе получим отсортированный массив [13]. Пример сортировки показан на рис. 1.9.

Рис. 1.9. Пример быстрой сортировки

Сложность алгоритма зависит от входных данных и в лучшем случае будет равняться O(n×2log2n). В худшем случае O(n^2). Существует также среднее значение, это O(n×log2n) [29].

1.2. Термин «Большие данные»

На сегодняшний день большой интерес представляют собой технологии класса BIG DATA. Данный интерес связан с постоянным ростом данных, с которыми возникает необходимость работать большим компаниям. Накапливаемая информация представляет собой важные активы, которыми компания пользуется в дальнейшей работе [30]. Но возникают трудности с обработкой и извлечением пользы из накопленных данных, так как они постоянно растут в объёме, что так же сказывается и на финансовых затратах компаний [26].

Большие данные – это растущая способность обработки, колоссальных по своим объёмам, массивов информации, их мгновенный анализ и возможность порой получить неожиданные выводы. Объём анализируемой информации велик настолько, что операции по обработке и анализу таких объёмов данных, при использовании стандартных программ и аппаратных средств является крайне сложным и долгим процессом [34].

В информационных технология большие данные – это ряд подходов, методов и инструментов по обработке неструктурированных и структурированных данных колоссальных объёмов, с дальнейшим получением большого разнообразия результатов, эффективность которых связана с условием постоянного прироста и распределением по большому числу узлов в вычислительной сети. [34]

Сложилось мнение, что большие данные являются совокупностью технологий, предназначение которых состоит в совершении трёх операций:

1. обрабатывать колоссальные по своим объёмам данные, в сравнении с обычным представлением;

2. иметь возможность работать с данными, поступающими в невероятно больших объёмах. То есть, подразумевается, что данные поступают в большом объёме не единоразово, а они постоянно становятся всё больше и больше.

3. организовывать работу как со структурированными, так и с плохо структурированными типа данных параллельно для разных аспектов. То есть предполагается, что большие данные поступают на вход алгоритма не только в виде структурированной информации и что существует возможность извлечь из них более чем одну идею [30].

В качестве типичного примера больших данных можно рассматривать информацию, поступающую от различного научного оборудования, служащего для проведения экспериментов. В качестве примера можно взять большой андройный коллайдер, постоянно производящий большие объёмы данных. Большие данные, поступающие от андройдонго коллайдера, проходят анализ и дальнейшую структуризацию, а учёные на основе полученной информации решают множество задач. [30]

Большинство того, что относится к большим данным, а также различные подходы для их анализа, существует довольно долгое время. В качестве примера, можно взять обработку потока изображений, с камер наблюдения, где подразумевается обработка не одного статичного изображения, а большого потока данных. Или же, в качестве ещё одного примера, можно рассмотреть навигацию роботов, когда происходит постоянный приём данных из внешнего окружения робота и последующая обработка полученной информации, для составления маршрута движения. Всё это существует на протяжении довольно большого количества времени. Просто на данный момент задача обработки больших данных затронула большее количество идей и людей [27].

Что же касается вопроса, насколько огромными являются большие данные – в современном понятии Big Data описывают как данные объёма в порядках терабайт.

В случаях, когда речь идёт о данных измеряемых в гигабайтах или терабайтах, хранить и управлять такими данными легко с помощью «традиционных» баз данных и оборудования предназначенного для этого (серверы баз данных) [27].

Организация и хранения Big Data обычно осуществляется в файловых системах распределённого типа. Если рассматривать в общих чертах, то хранение информации происходит на нескольких (их количество может измеряться в тысячах) жёстких дисках, на обычных компьютерах. Для отслеживания на каком компьютере или диске хранится требуемая часть информации, используется так называемая «карта» (map)[30].

С целью обеспечить отказоустойчивость и надёжность, обычно каждую из частей информации сохраняют по несколько раз, например – трижды.

Основная часть информации, хранящейся в распределённой файловой системе, состоит является не однородной и состоит таких данных как: изображение, текст или видеозаписи. В этом кроются как свои преимущества, так и недостатки [27].

К преимуществу можно отнести то, что возможность хранения больших данных предоставляет возможность хранить все типы данных, не задумываясь над тем, какая из частей хранимых данных является актуальной для последующего анализа и принятия решения.

К недостаткам же относится то, что при необходимости извлечения полезной информации, потребуется обработка этих огромных по своим объёмам неструктурированных данных.

Однако не все операции являются трудоёмкими (например, простые подсчёты и т.д.), однако для выполнения других – более сложных операций,

требуются сложны алгоритмы, предназначенные для эффективной работы с распределённой файловой системой. [30]

1.3. Технологии параллельного программирования

Для разработки программного обеспечения должного уровня, следуя тенденциям текущего времени, программисту необходимо знать и применять на практике знания о параллельном и распределённом программировании. Но чем больше функциональности требуется от программного обеспечения, тем больше требований предъявляется к самому программному обеспечению. Для удовлетворения минимальных требований пользователя, программы должны быть как можно более интеллектуальными и производительными. Проектирование программного обеспечения должно осуществляться на таком уровне, чтобы оно могло воспользоваться всеми преимуществами многопроцессорных систем. Так как на данный момент сетевые компьютеры не являются чем-то исключительным, то к целям, которые должны придерживаться программисты при проектировании программного обеспечения, также относится эффективное и корректное выполнение различных частей программного кода на нескольких различных компьютерах. Для достижения данной цели необходимо использовать методы, реализующие параллелизм посредством распределённого и параллельного программирования [1].

Целью параллелизма является процесс выявления подзадач, выполнение которых может быть одновременным. Два события называются одновременными, если эти события происходят в течение одного и того же временного интервала [33].

Выявление параллелизма: это построение структуры поставленной задачи таким образом, чтобы можно было эффективно выполнять подзадачи. Для достижения этой цели зачастую необходимо найти зависимости между

подзадачами и сделать организацию исходного кода такой, чтобы этими подзадачами можно было эффективно управлять [4].

Параллельные задачи могут выполняться в одно- или многопроцессорной среде. В однопроцессорной среде существование параллельных задач происходит в одном и том же интервале времени за счёт использования контекстного переключения. В случае с многопроцессорными средами, когда свободными являются несколько процессоров, параллельные задачи выполнение параллельных задач может быть в одни и те же моменты времени в течение одного и того же периода времени. Фактор оказывающий влияние на степень приемлемости для параллелизма того или иного интервала времени, определяется конкретным приложения [6].

Параллельное и распределённое программирование являются двумя базовыми подходами по достижению параллельного выполнения составляющих программного обеспечения. Данные подходы являются двумя различными моделями программирования, которые иногда могут пересекаться. Благодаря методам параллельного программирования есть возможность распределить выполняемую программу между двумя (или более) процессорами в пределах одного физического или одного виртуального компьютера [9]. С помощью методов распределённого программирования возможно осуществить распределение работы программы между двумя (или более) процессами, при этом процессы могут существовать как на одном компьютере, так и на разных. Говоря другими словами, выполнение распределённой программы чаще всего происходит на разных компьютерах, связанных между собой по сети, или же как минимум в различных процессах. Выполнение программы, содержащая в своём коде параллелизм, осуществляется на одном и том же физическом или же виртуальном компьютере. Такую программу можно разбить на процессы (process) или потоки (thread) [31].

Для реализации параллельных программ используются специализированные языки программирования и специальные системы поддержки параллельного

программирования, например, MPI и OpenMP. Сложность параллельных программ, по сравнению с последовательными, заключается как в написании кода, так и в последующей его отладке [3].

В языке C++ параллелизм достигается с помощью разложения программы на несколько процессов или потоков[19].

Процесс (process) — это некоторая часть (единица) работы, создаваемая операционной системой. Важно отметить, что процессы и программы — необязательно эквивалентные понятия [23].

Для того, чтобы назвать некоторую часть работы процессом, ей необходимо иметь своё адресное пространство, которое назначает операционная система, а также идентификатор, или идентификационный номер процесса (id процесса). Также процесс обязан иметь свой элемент в таблице процессов и обладать определённым статусом [12].

1.3.1. Основы технологии OpenMP

OpenMP – API предназначенный для реализации многопоточных приложений на многопроцессорных системах с общей памятью. Над спецификациями OpenMP работают несколько крупных производителей вычислительной техники и программного обеспечения. OpenMP поддерживается основными компиляторами [13].

Когда речь идёт о многопроцессорных вычислительных системах с общей памятью подразумевается, что данные системы обладают процессорами с идентичной производительностью, имеют равные права на доступ к общей памяти, а также, что у процессоров одинаковое время доступа к памяти (при одновременном обращении нескольких процессоров к одному и тому же элементу памяти очерёдность и синхронизация доступа обеспечивается на аппаратном уровне). Основная схема обращения к обещай памяти приставлена

на рис. 1.10. Системы подобного типа именуются симметричными мультипроцессорами [19].

В самом общем виде системы с общей памятью (см. рис. 10) могут быть представлены в виде модели параллельного компьютера с произвольным доступом к памяти [9].

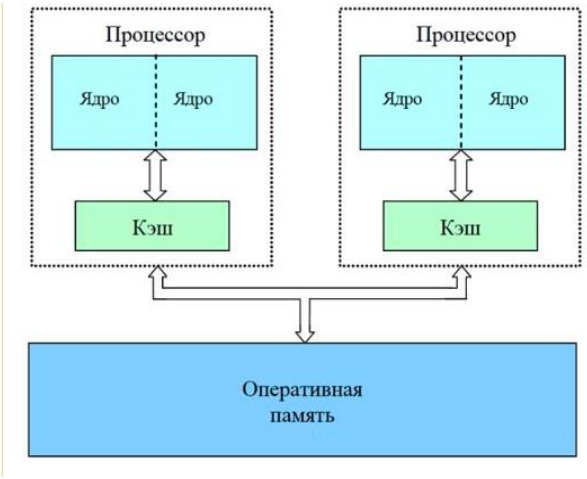

Рис. 1.10. Архитектура многопроцессорных систем с общей (разделяемой) с однородным доступом памятью.

OpenMP достиг большого успеха среди языков параллельного программирования. Он находится на каждом из выходящих на рынок компьютере, с совместным использованием памяти. Так же недавно компания Intel создала вариант OpenMP с поддержкой кластеров [9]. В OpenMP поддерживается такой стиль программирования, когда параллелизм добавляется постепенно, до тех пор, пока последовательная программа не перерастёт в параллельную. Но тем не менее, данное преимущество является так же и самым слабым из мест OpenMP. В случаях, когда параллелизм добавляется поэтапно, может случится так, что программист не выполнит широкомасштабную перестройку программы, в результате чего не будет достигнута максимальная производительность программы [17].

Программа, использующая возможности OpenMP, начинает свою работу с одного потока. В случаях, когда программисту необходимо добавить параллелизм в программу, выполняется разветвление на несколько потоков, с

целью создания группы потоков. Выполнение таких потоков происходить параллельно, в рамках фрагмента кода, называемого параллельным участком. После того, как параллельный участок заканчивается, все потоки заканчивают свою работу и происходит их объединение. Далее исходный («главный») поток продолжит выполняться до того момента, пока не начнётся новый параллельный участок (или не будет достигнут конец программы) [9].

OpenMP не показывает потоки в коде. Чтобы показать компилятору, где происходит распараллеливание кода используются директива #pragma. Основываясь на данной информации, компилятор генерирует приложение, которое состоит из одного главного потока, который в свою очередь создаст другие потоки для параллельных блоков кода. В конце параллельного блока происходит синхронизация потоков, после чего происходит возврат к одному главному потоку [5]. Схема распределения кода по потокам показана на рисунке 1.11.

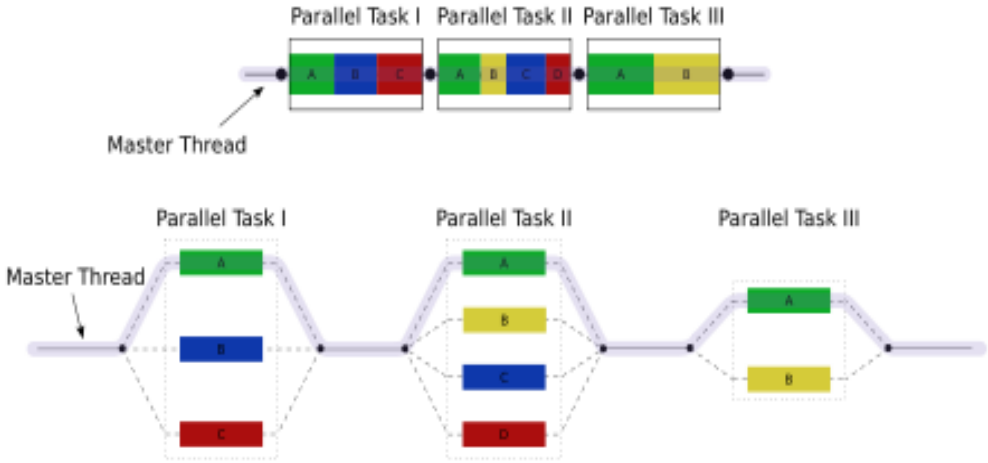

Рис.1.11. Общая схема выполнения параллельной программы при использовании технологии OpenMP

Использование OpenMP предусматривается со следующими алгоритмическими языками: Fortran 77, Fortran 90, Fortran 95, C++, C99, C90. В общем виде формат директив OpenMP можно представить в следующем виде: #pragma omp <имя_директивы> [<параметр>[[,] <параметр>]...].

Поскольку вызов OpenMP находится под контролем директивы #pragma, то последовательный код будет корректно выполнен любым компилятором, потому как неподдерживаемые директивы должны быть проигнорированы. Также API OpenMP содержит несколько функций, которые необходимо подключить через заголовочный файл, если нужно ими воспользоваться [10]. Самым простым из способов, для определения поддержки OpenMP компилятором, это подключить заголовочный файл omp.h: #include <omp.h>.

Конструктивно в составе технологии OpenMP можно выделить:

1. Директивы,
2. Библиотеку функций,
3. Набор переменных окружения.

Для выделения параллельных фрагментов программы следует использовать директиву parallel:

#pragma omp parallel [<параметр> ...] <блок_программы>

Для блока, использующего данную директиву (как и для всех блоков, которые используют директивы OpenMP) необходимо выполнение правила «один вход – один выход», то есть не допускается передача управления извне к блоку и из блока за пределы блока не допускается [11].

Parallel – это одна из основных директив OpenMP. Существуют следующие правила, которые определяют действие директивы:

- При достижении программы директивы parallel, происходит процесс создания набора (team) из числа N потоков; исходный поток программы – это основной поток этого набора (master thread), которому присваивается 0 номер.

- Код блока, следующего за объявлением директивы, подвергается дублированию или же может быть разделён с помощью специальных директив, для параллельного выполнения между потоками.

- По окончанию блока директивы происходит синхронизация между потоками – ожидание окончания выполнения кода на всех потоках; после чего работа потоков завершается – далее свою работу продолжает только главный

поток (в зависимости от того, какая выбрана среда реализации OpenMP потоки могут и не завершиться, а всего лишь приостановиться, до тех пор, пока не будет достигнут следующий параллельный участок кода, что позволяет снизить затраты на создание и удаление потоков) [2].

В таблице 1.1 приведен перечень параметров директивы parallel

Таблица 1.1

Сводный перечень директив OpenMP

Директива	**Описание**
private (list)	Параметр для создания локальных копий для перечисленных в списке переменных для каждого имеющегося потока. Исходные значения копий не определены. Директивы: parallel, for, sections, single
firstprivate(list)	Тоже что и параметр private и дополнительно инициализация создаваемых копий значениями, которые имели перечисленные в списке переменные перед началом параллельного. Директивы: parallel, for, sections, single
lastprivate(list)	Тоже что и параметр private и дополнительно запоминание значений локальных переменных после завершения параллельного фрагмента. Директивы: for, sections
shared (list)	Параметр для определения общих переменных для всех имеющихся потоков. Директивы: parallel
default(shared | none)	Параметр для установки правила по умолчанию на использование переменных в потоках. Директивы: parallel
reduction(operator: list)	Параметр для задания операции редукции. Директивы: parallel, for, sections
nowait	Параметр для отмены синхронизации при завершении директивы. Директивы: for, sections, single
if(expression)	Параметр для задания условия, только при выполнении которого осуществляется создание параллельного фрагмента. Директивы: parallel
ordered	Параметр для задания порядка вычислений в распараллеливаемом цикле. Директивы: for
schedule (type [, chunk])	Параметр для управления распределением итераций распараллеливаемого цикла между потоками. Директивы: for
copyin (list)	Параметр для инициализации постоянных переменных потоков. Директивы: parallel
copyprivate(list)	Копирование локальных переменных потоков после выполнения блока директивы single. Директивы: single

OpenMP позволяет разделять итеративно-выполняемые действия в циклах, для того, чтобы явно указать с какими данными необходимо выполнять соответствующие вычисления. Данная возможность является важной, так как в большинстве случаев именно в циклах происходит основная часть трудоёмких вычислений [9]. Для распараллеливания циклов в OpenMP применяется директива for: #pragma omp for [<параметр> ...] <цикл_for> [33].

После объявления данной директивы происходит распределение итераций цикла между потоками, в результате чего они могут выполняться параллельно. Возможность такого распараллеливания возможно только в том случае, если между итерациями цикла отсутствует информационная зависимость [34].

По умолчанию, чтобы перейти к дальнейшим вычислениям, каждый поток ожидает окончания выполняемых итераций в цикле, даже если в случае, когда некоторые уже прекратили свои вычисления – окончание цикла является неким барьером, преодолеть который потоки могут только совместно. Синхронизацию потоков можно отменить, если воспользоваться параметром nowait в директиве for - в результате потоки продолжат вычисления вне цикла, если для них отсутствуют итерации цикла для выполнения [31].

1.3.2. Основы технологи MPI

MPI – это интерфейс обмена сообщениями, являющийся одним из числа старейших API-интерфейсов в параллельном программирование, применяющихся по сей день. Программы на MPI представляют собой набор из независимых процессов, взаимодействие между которыми происходит за счёт отправки и получения сообщений между собой. Сильной стороной MPI является то, что данный интерфейс имеет низкие требования по отношению к аппаратной части параллельного компьютера [40]. Единственное требование этого интерфейса – это чтобы процессы или ядра совместно использовались в одной сети, пригодной для обмена сообщениями между двумя процессами. Благодаря

этому MPI может работать на любой из стандартных параллельных систем, от многопроцессорных симметричных систем до систем с распределённой памятью, от суперкомпьютеров с высокой степенью параллелизма до кластеров [12].

Большинство программ, написанных на MPI, придерживаются шаблона «Одна программа, разные данные» (Single Program Multiple Data или SPMD). В основе которого лежит принцип: каждый из элементов обработки (processing element или PE) выполняет одну и туже программу. Каждому из элементов получается уникальный целочисленный идентификатор, определяющий его ранг в общем наборе элементов обработки. Ранг используется программой, для того чтобы распределить между обработчиками работу и получить возможность определить, какой из элементов PE какую работу выполняет. Проще говоря, программа одна, но благодаря тому, что есть выбор в соответствии с идентификатором, данные, для каждого из элементов обработки, могут быть разными [40].

MPI является изящной и ясной системой обмена сообщениями, которая была разработана для поддержания широкого спектра аппаратных средств. Также она была спроектирована таким образом, чтобы можно было поддерживать сложные программные архитектуры с точной модульной структурой [29].

По умолчанию, во время запуска MPI программы создаётся коммуникатор MPI_COMM_WORLD, передаваемый каждой MPI программе в качестве первого элемента. Другие аргументы служат для определения источника сообщения и буферов для хранения сообщений. Программы на MPI в качестве параметра ошибки возвращают целочисленное значение, что даёт возможность узнать о любой проблеме, имеющей место при выполнении программы [17].

Зачастую в начале MPI программы осуществляется вызов трёх подпрограмм, для настройки применения MPI [9].

Листинг 1.1. Основные функции MPI
int my_id, numprocs; *MPI_Init(&argc, &argv) ;* *MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;* *MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;*
Конец листинга 1.1.

После чего следует уже рабочая часть MPI программы. Большая часть программы является обычным последовательным кодом. В тех местах, где необходима связь или любое другое взаимодействие между процессами вставляются MPI подпрограммы. [1]

Каждая MPI программ должна иметь в конце подпрограмму, которая закрывает среду. Данная функция производит возврат целочисленного значения, представляющее собой код ошибки [18].

1.4. Сравнительный анализ сортировки данных

При внимательном рассмотрении способов упорядочивания данных, применяемых в алгоритмах сортировки, можно заметить, что многие методы основаны на применении одной и той же базовой операции "сравнить и переставить" (compare-exchange), состоящей в сравнении той или иной пары значений из сортируемого набора данных и перестановке этих значений, если их порядок не соответствует условиям сортировки [13].

Целенаправленное применение данной операции позволяет упорядочить данные; в способах выбора пар значений для сравнения, собственно, и проявляется различие алгоритмов сортировки [36].

Ни одна другая проблема не породила такого количества разнообразнейших решений, как задача сортировки. Универсального, наилучшего алгоритма сортировки на данный момент не существует. Однако, имея приблизительные характеристики входных данных, можно подобрать метод, работающий

оптимальным образом. Для этого необходимо знать параметры, по которым будет производиться оценка алгоритмов [13].

Для большинства алгоритмов количество выполняемых ими операций напрямую зависит от размера входных данных.

Если известно, что на входе будут поступать данные не больших размеров, то вопрос о выборе эффективного алгоритма не является первостепенным – можно использовать «простой» алгоритм. Вопросы, связанные с эффективностью алгоритмов, приобретают смысл при больших размерах входных данных – при $n \to \infty$. Но ели использовать большие данные, то многие алгоритмы просто не подходят для сортировки [37].

Есть несколько критериев по которым сравниваются алгоритмы. В основном алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти. Но есть более подробные критерии. Разберём более подробно:

1. Время — основной параметр, который характеризует скорость работы алгоритма. Называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах мощности входного множества A. Алгоритмы сортировки, использующие только абстрактную операцию сравнения ключей всегда нуждаются по меньшей мере в сравнениях.

2. Память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. Как правило, эти алгоритмы требуют $O(\log n)$ памяти. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы (так как всё это потребляет $O(1)$). Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к сортировкам на месте.

3. Устойчивость — устойчивая сортировка, для которой расположение элементов с идентичными ключами не меняется.

4. Естественность поведения — эффективность метода при обработке уже упорядоченных или частично упорядоченных данных. Если учитывать входную последовательность, то она работает лучше при таких характеристиках [13].

- потребности в дополнительной памяти или её отсутствию
- потребности в знаниях о структуре данных, выходящих за рамки операции сравнения, или отсутствию таковой.

Использование операции сравнения**.** Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях [20].

Также алгоритмы классифицируются по трудоёмкости худшего случая для этих алгоритмов составляет $O(n \cdot \log n)$, но главное отличие таких алгоритмов, это гибкость применения [33].

В более общем смысле при исследовании алгоритмов необходимо ответить на 4 вопрос.

1. Размер входных данных

2. Качество реализации алгоритма на языке программирования

3. Качество скомпилированного кода

4. Производительность вычислительной машины

Для большинства алгоритмов количество выполняемых ими операций напрямую зависит от размера входных данных [32]. На рис. 1.12. представлена схема.

Рис.1.12. операции выполнения алгоритма

Все эти условия влияют на использование того или иного алгоритма в определенной ситуации [34].

В данной магистратской работе рассмотрено применение больших данных и необходимо изначально провести анализ методов, которые можно применить. При анализе эффективности алгоритмов будет оценивать их на потребности в вычислительных ресурсах: процессором времени, памяти, пропускной способности сети. И на количество выполняемых операций – временна сложность алгоритмов показывает на сколько быстро алгоритм работает. И сравним алгоритмы между собой [34].

Временная сложность алгоритма определяет время работы, используемое алгоритмом, как функции от длины строки, представляющей входные данные.

Обычно класс сложности обозначают как представлено в Талице 1.2.

Таблица 1.2

Класс сложности алгоритмов

Класс сложности	**Название**
$O(1)$	Константная сложность
$O(\log(n))$	Логарифмическая сложность
$O(n)$	Линейная сложность
$O(n\log n)$	Линейно-логарифмическая сложность
$O(n^2)$	Квадратичная сложность
$O(n^3)$	Кубическая сложность
$O(2^n)$	Экспоненциальная сложность
$O(n!)$	Факториальная сложность

Порядок роста *O(1)* означает, что вычислительная сложность алгоритма не зависит от размера входных данных.

Порядок роста *O(n)* означает, что сложность алгоритма линейно растет с увеличением входного массива.

Порядок роста $O(\log n)$ означает, что время выполнения алгоритма растет логарифмически с увеличением размера входного массива.

Линеарифметический (или линейно-логарифмический) алгоритм имеет порядок роста $O(n \cdot \log n)$ [34].

Время работы алгоритма с порядком роста $O(n^2)$ зависит от квадрата размера входного массива. Несмотря на то, что такой ситуации иногда не избежать, квадратичная сложность — повод пересмотреть используемые алгоритмы или структуры данных [33]. Время выполнения алгоритма для небольших n показано на рис. 1.13.

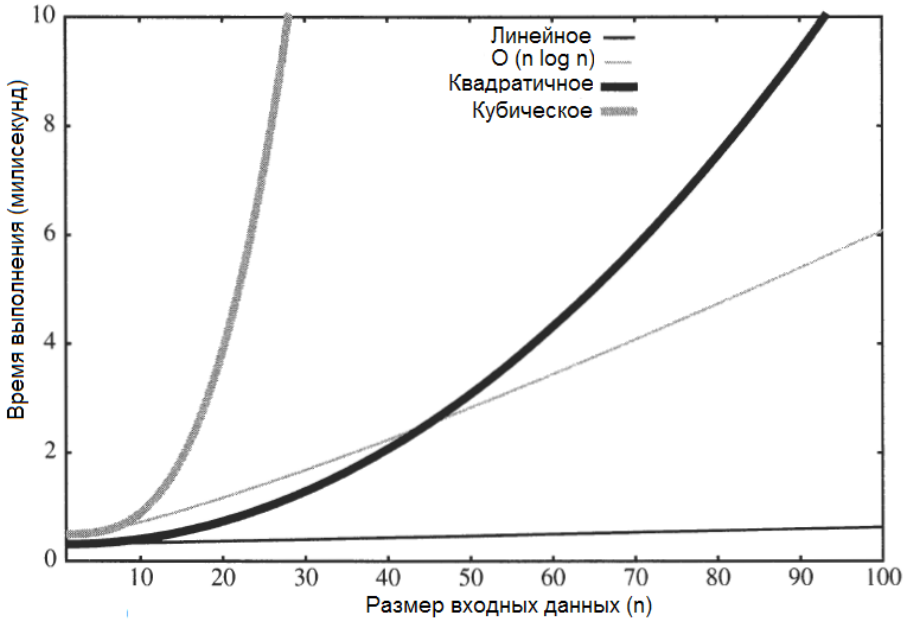

Рис. 1.13. Время выполнения алгоритма для небольших n

Время выполнения алгоритма для больших n показано на рис. 1.14.

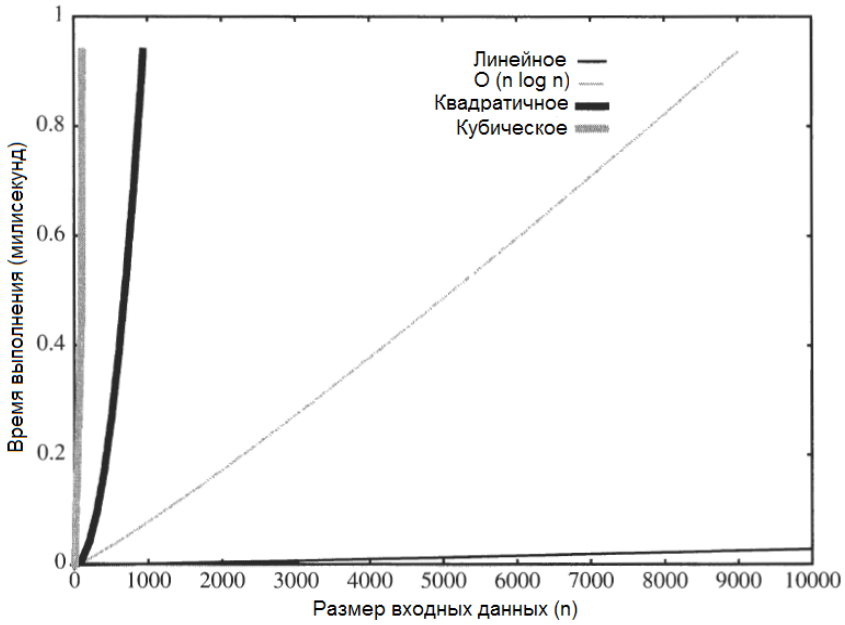

Рис. 1.14. Время выполнения алгоритма для больших n

Исходя из графиков видно, что сортировке больших данных наиболее подходящий алгоритм, для которого сложность, является линейно-логарифмический [34].

Разберёмся с выражением "время выполнения в среднем", оно понимается как усреднение времени выполнения сортировки по всем возможным упорядочениям исходных наборов элементов в предположении, что все упорядочения равновероятны [24]. Однако, такая оценка достаточно сложна:

1. исходные данные разбиваются на группы так, что трудоемкость алгоритма (ti) для любого набора данных одной группы одинакова;

2. исходя из доли наборов данных группы в общем числе наборов, рассчитывается вероятность для каждой группы (pi);

3. оценка среднего случая вычисляется по формуле, указанной ниже.

$$\sum i=1mpi \cdot ti. \qquad (1.7)$$

Подсчет количества операций позволяет сравнить эффективность алгоритмов. Однако, аналогичный результат можно получить более простым путем. Анализ проводят с расчетом на достаточно большой объем обрабатываемых данных ($n \to \infty$), поэтому ключевое значение имеет скорость роста функции сложности, а не точное количество операций [15].

В этой таблице, п есть число записей, подлежащих сортировке. Колонки «Средняя» и «Худшие» дают временную сложность в каждом конкретном случае. «Память» обозначает количество вспомогательной памяти, необходимой за что используется в самом списке, при этом предположении [29].

В таблицы 1.3 приведены пример как алгоритмы с ключами одинакового размера так и с ключами размера k , размера цифр d и r диапазон чисел, которые нужно отсортировать. Многие из них основаны на предположении, что размер ключа достаточно велик, и все записи имеют уникальные ключевые значения, и, следовательно, $n \; 2^{k}$, где« означает «гораздо меньше» [28].

Таблица 1.3

Сравнение алгоритмов

Сортировка	**Лучший**	**Средний**	**Наихудший**	**Память**	**Устойчивость**
Сортировка Пузырьком	*n*	n^2	n^2	1	да
Сортировка вставками	*n*	n^2	n^2	1	да
Сортировка слиянием	*nlogn*	*nlogn*	*nlogn*	n Гибридная сортировка слияния блоков - O (1) mem.	да
Сортировка двоичного дерева	*nlogn*	*nlogn*	*nlogn*	n	да
Блочная сортировка	*n*	*nlogn*	*nlogn*	1	да
Сортировка выбором	n^2	n^2	n^2	1	нет
Пирамидальная сортировка	*nlogn*	*nlogn*	*nlogn*	1	нет
Интроспективная сортировка	*nlogn*	*nlogn*	*nlogn*	logn	нет
Терпеливая сортировка	*n*	-	*nlogn*	n	нет

Выше приведен фрагмент таблицы, всю таблицу можно посмотреть в приложении на странице 65.

Простейшие из этих алгоритмов затрачивают время порядка *O(n^2)* для упорядочивания n объектов и потому применимы только к небольшим множествам объектов [28].

Исходя из таблицы выше, по скорости алгоритмы у которых время *O(nlogn)* являются более оптимальны для больших данных.

Один из наиболее популярных алгоритмов сортировки, так называемая быстрая сортировка, выполняется в среднем за время *O(n logn).* Быстрая сортировка хорошо работает в большинстве приложений, хотя в самом худшем случае она также имеет время выполнения *O(n2).* Существуют другие методы сортировки, такие как пирамидальная сортировка или сортировка слиянием, которые в самом худшем случае дают время порядка *O(n logn),* но в среднем (в статистическом смысле) работают не лучше, чем быстрая сортировка [28].

Для реализации возьмем два алгоритма. Сортировку слиянием и быструю сортировку, посмотрим, как работают алгоритмы при последовательном и параллельном алгоритме и исползаем программы для сортировки больших данных [33].

ГЛАВА 2 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. Проектирование алгоритмов сортировки

На рисунке 2.1 приведена блок-схема параллельного алгоритма быстрой сортировки.

Входными данными алгоритма являются размер массива для сортировки.

Суть алгоритма быстрой сортировки выражается в двух шагах:

a. Разбиение массива на n/p, где n-количество элементов массива, а p-количество ядер;

b. Каждый процесс разбивает свою часть до парного состояния;

c. Сортировка парных частей уже всего массива [33].

В ходе работы алгоритма исходный набор окажется разделенным на две части, меньшая из которых передастся другому свободному процессору, большая останется на исходном для дальнейшей обработки.

Далее обе части опять будут разделены и опять на двух исходных останутся большие части (на первом процессоре всё равно большая), а меньшие отправятся другим процессорам. В этом заключается ускорение алгоритма. При задействовании всех процессов, все части параллельно будут сортироваться последовательным алгоритмом.

Рис. 2.1 блок схема параллельного алгоритма быстрой сортировки

На рисунке 2.2 приведена блок-схема параллельного алгоритма сортировки слияния.

Входными данными алгоритма являются размер массива для сортировки.

Суть алгоритма быстрой сортировки выражается в двух шагах:

1. Разбиение массива на n/p, где n-количество элементов массива, а p-количество ядер
2. Поиск опорного элемента каждом массиве и сравнение элементов в каждом процессе отдельно.
3. Слияние массивов из каждого процесса в один [33].

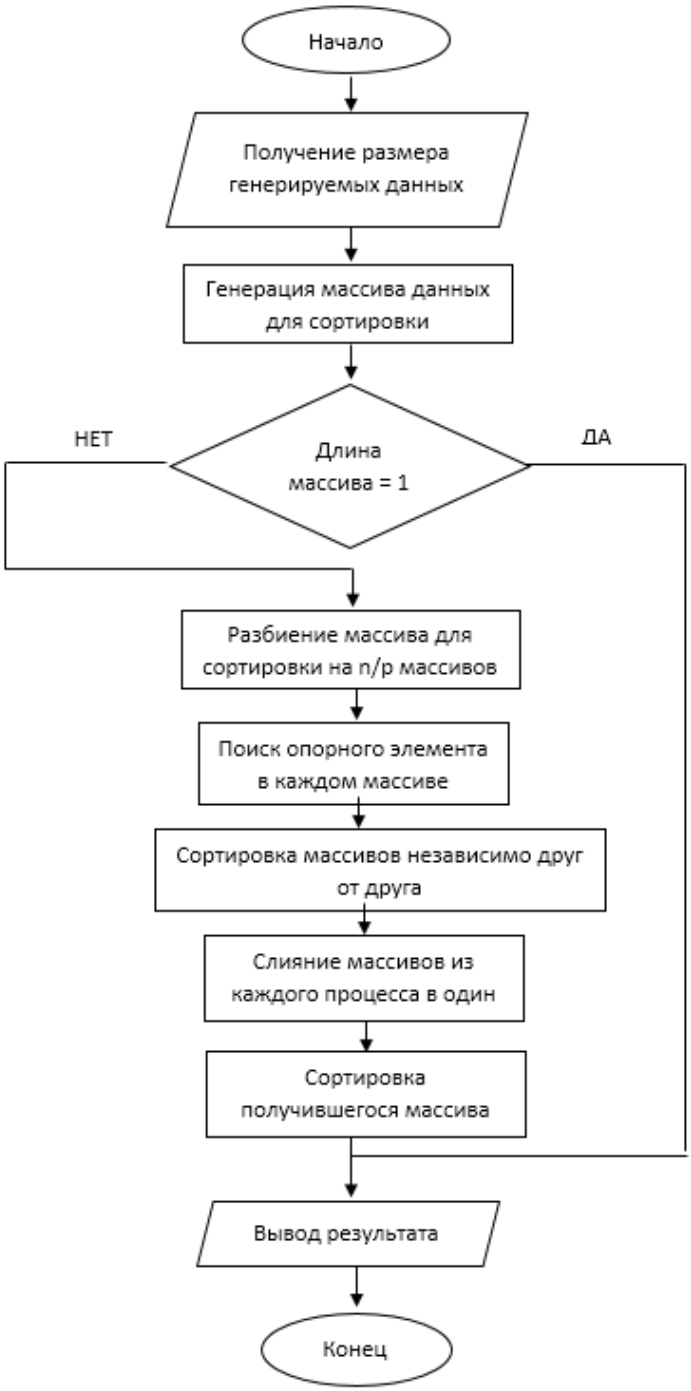

Рис. 2.2 блок схема параллельной сортировки слияния

2.2. Реализация алгоритма быстрой сортировки

В данном разделе рассмотрим результаты вычислительных экспериментов из раздела 3.

Разработка и вычислительный эксперимент проводился на компьютере с характеристиками: Intel core i7 4770 3.9 гигагерц, 8 ядер. ОЗУ: 8 гигабайт. Разработка делалась на Linux дистрибутив - Ubuntu 16.04. с версией ядра 9.20160110. Компилятор g++ входящий в поставке gcc верси 5.4.0.

В алгоритме быстрой сортировки, исходный массив разбивается на 2 части, обработка которых ведется не зависимо (поэтому может выполняться параллельно).

Всякий раз, когда i-тый элемент оказывается больше pivot, выполняется поиск *j*-того элемента меньше pivot. Найденные элементы обмениваются местами. В качестве pivot можно выбрать любой элемент массива.

Такое время выполнения алгоритма является более эффективным при работе с большими данными [33].

Таблица 2.1

Временная сложность быстрой сортировки

Сложность	Наилучший случай	В среднем	Наихудший случай
Время	O(n·log n)	O(n·log n)	$O(n^2)$
Память	O(1)	O(1)	O(1)

При оптимальном выборе ведущих элементов, когда разделение каждого блока происходит на равные по размеру части, трудоемкость алгоритма совпадает с быстродействием наиболее эффективных способов сортировки

$$(T_1 = n \log_2 n) \tag{2.1}$$

Общая схема алгоритма быстрой сортировки может быть представлена в следующем виде (в качестве ведущего элемента выбирается первый элемент упорядочиваемого набора данных) [36].

В блоке do производим изменения начала и конца до тех пор, пока начало меньше или равно концу. Увеличиваем начальный элемент сортировки, пока текущий элемент меньше того, с которого нужно начинать сортировку. И уменьшаем конечный элемент сортировки, если текущий элемент больше, чем элемент, которым должна заканчиваться сортировка. Когда начальный элемент, меньше конечного элемента, меняем их местами [37].

Листинг 2.1. Заполнение массива

```
do{     while(v[b] < d){ ++b; }
                while(v[e] > d){ -e;}
        if(b <= e){ //                  swap(v[b],v[e]);
                ++b; //обновляем начальный указатель сортировки
                --e; //обновляем конечный указатель сортировки
        }
} while(b<=e);          //разделение массива на две части
if(e > b0){
        sort(v,b0,e);   }
if(b < e0){
        sort(v, b,e0);
}}
```

Конец листинга 2.1.

Приведем код, выполняющей последовательный алгоритм быстрой сортировки. В качестве ведущего элемента выбирается первый элемент упорядочиваемого набора данных. В этом блоке заполняем массив.

Листинг 2.2. Последовательный алгоритм быстрой сортировки

```
        void QuickSort(double A[], int i1, int i2) {
 if (i1 < i2) {
  double pivot = A[i1];
  int is = i1;
  for (int i = i1 + 1; i < i2; i++)
   if (A[i] J pivot) {
    is = is + 1;
    swap(A[is], A[i]);     }
  swap(A[i1], A[is]);
  QuickSort(A, i1,    is);
  QuickSort(A, is + 1, i2);  }}
```

Конец листинга 2.2.

Преимущества параллельного метода *быстрой сортировки*, как и в последовательном варианте, во многом зависит от правильности выбора значений ведущих элементов. Определение общего правила для выбора этих значений представляется затруднительным. Сложность такого выбора может быть снижена, если выполнить упорядочение локальных блоков процессоров перед началом сортировки и обеспечить однородное распределение сортируемых данных между процессорами вычислительной системы [37].

Определим вначале вычислительную сложность алгоритма сортировки. На каждой из $\log_2 p$ итераций сортировки каждый процессор осуществляет деление блока относительно ведущего элемента, сложность этой операции составляет n/p операций (будем предполагать, что на каждой итерации сортировки каждый блок делится на равные по размеру части).

При завершении вычислений процессор выполняет сортировку своих блоков, что может быть выполнено при использовании быстрых алгоритмов за $(n/p)\log_2(n/p)$ операций [24].

При распараллеливании алгоритма сортировки наиболее простой случай, когда потоки параллельной программы могут быть организованы в виде N-

мерного гиперкуба (т.е. количество вычислительных элементов p=2N). Тогда способы выполнения алгоритма состоят в следующем:

1. выбрать каким-либо образом, ведущий элемент и разослать его по всем процессорам системы (например, в качестве ведущего элемента можно взять среднее арифметическое элементов, расположенных на выбранном ведущем процессоре);

2. далее на каждом процессоре разделить имеющийся блок данных на две части;

3. образовать пары процессоров, для которых битовое представление номеров отличается только в позиции N, и осуществить взаимообмен данными между этими процессорами [32].

После выполнения пунктов исходный массив оказывается разделенным на две части, одна из которых располагается на процессорах, в битовом представлении номеров которых бит N равен 0. Таких процессоров всего p/2, и, таким образом, исходный N -мерный гиперкуб также оказывается разделенным на два гиперкуба размерности N-1. К этим подгиперкубам, в свою очередь, может быть параллельно применена описанная выше процедура. После N - кратного повторения подобных итераций для завершения сортировки достаточно упорядочить блоки данных, получившиеся на каждом отдельном процессоре вычислительной системы [29].

Функция ParallelQuickSort выполняет параллельный алгоритм быстрой сортировки:

Листинг 2.3. Последовательный алгоритм сортировки слиянием

```
void quickSort(int num, float* a, const long n) {
long i = 0, j = n;
float    pivot = a[n / 2]; // опорный элемент
#ifdef DEBUG
#pragma omp critical
        {printf("enter: %d\n", num);          }
```

Конец листинга 2.3.

Также как видно на примере кода критическая секция предотвращает множественный одновременный доступ к определенному сегменту кода. Поток получает доступ к критической секции лишь в том случае, когда эта критическая секция не обрабатывается другим потоком. Рассмотрим пример с организацией неименованных критических секций [28].

Листинг 2.4 Функция critical

```
#pragma omp critical
        {
                printf("divide #%d by %3.2f [%d] -> (%d|%d) : %d\n", num, pivot, n/2, j, n - i, n);
        }
```

Конец листинга 2.4.

Для задания списка переменных, общих для всех нитей, используется директива:

Листинг 2.5 Дириктива task shared(a)

```
#pragma omp task shared(a)
{
        if (j > 0) quickSort(num * 2 + 1, a, j);        } // #pragma omp task
```

Конец листинга 2.5.

Для гарантированного завершения в точке вызова всех запущенных задач используется директива taskwait. Нить, выполнившая данную директиву, приостанавливается до тех пор, пока не будут завершены все ранее [36].

Листинг 2.6 Директива omp taskwait

```
#pragma omp taskwait}
int main(int argc, char *argv[]) {
        if (argc < 2) {
                printf("error arg\n");
                return 1;  }
```

Конец листинга 2.6

2.3 Реализация алгоритма сортировки слиянием

Стоит отметить, что в отличие от линейных алгоритмов сортировки, сортировка слиянием будет делить и склеивать массив вне зависимости от того, был он отсортирован изначально или нет. Поэтому, несмотря на то, что в худшем случае он отработает быстрее, чем линейный, в лучшем случае его производительность будет ниже, чем у линейного.

Таблица 2.2

Сложность алгоритма сортировки слиянием

Сложность	**Наилучший случай**	**В среднем**	**Наихудший случай**
Время	O(n·log n)	O(n·log n)	O(n·log n)
Память	O(n)	O(n)	O(n)

Следующий, из обсуждаемых алгоритмов, алгоритм требует выполнения O(n·log n) действий [33]. Он основан на идее рекурсивного слияния упорядоченных фрагментов массива.

Само слияние двух упорядоченных массивов с длинами n и m требует O( ) n + m действий, при использовании отдельного массива для записи результата слияния. Поскольку в рекурсивном алгоритме результат должен быть размещен в том же массиве, что и исходный массив, необходимо на каждом шаге выполнить копирование результата, что значительно замедляет обработку [26]. В связи с этим предлагается использовать аналогичный алгоритм, не использующий рекурсию.

Листинг 2.7. Последовательный алгоритм сортировки слиянием

```
void mergeSort(vector<int> &a, size_t start, size_t end){
if(end - start < 2){
        return;
}       if(end - start == 2){
```

```
            if(a[start] > a[start+1]){
                swap(a[start], a[start+1]);
            }       }
        mergeSort(a, start, start + (end-start) / 2);
        mergeSort(a, start + (end-start) /2, end);
```

Конец листинга 2.7.

Разберёмся в чем суть параллельной реализации сортировки слиянием: сначала выполняется передача, разбиение массива элементов по всем процессам. Далее каждый процесс параллельно сортирует свой массив последовательным алгоритмом.

Листинг 2.8. Директива generate_list

```
        void generate_list(int * x, int n) {
  int i,j,t;
  for (i = 0; i < n; i++)
    x[i] = i;
  for (i = 0; i < n; i++) {
    j = rand() % n;
    t = x[i];
    x[i] = x[j];
    x[j] = t;
  }}
```

Конец листинга 2.8.

Когда мы уже имеем несколько отсортированных массивов, ставится задача слияния этих массивов в один, результирующий. Слияние будет проводиться итерационно: на первом шаге процессы делятся на пары, и правый массив передаёт свои данные левому. На втором шаге также объединяем процессы в пары, но только те, которые были левыми на первом шаге. Опять, также, правый процесс передаёт свои данные левому, а тот их сортирует. Это выполняется до тех пор, пока не останется один процесс [33]. Его массив и будет результатом параллельной сортировки.

ГЛАВА 3 ВЫЧИСЛИТЕЛЬНЫЙ ЭКСПЕРИМЕНТ

Для проведения тестирования разработанных алгоритмов сортировки слиянием и быстрой сортировки данные брались случайным образом.

3.1. Быстрая сортировка данных

Сначала проверим работоспособность алгоритма при последовательной сортировке данных. Зададим случайным образом массив размерностью 20 элементов. Результатом работы алгоритма при маленьких данных будет отсортированная последовательность, как показано на рисунке 3.1. Также на рисунке видно, как меняется последовательность алгоритма на каждом шаге алгоритма.

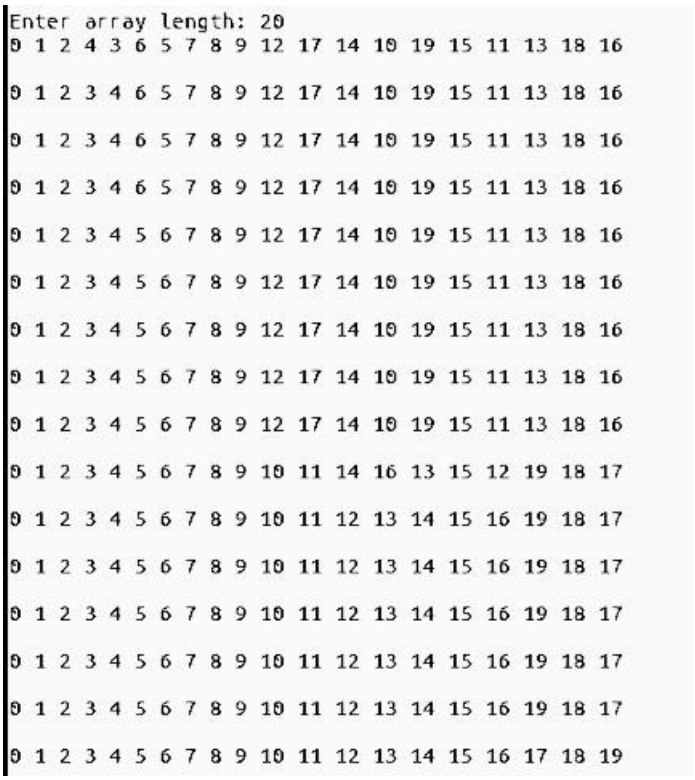

Рис 3.1. результат работы алгоритма быстрой сортировки

Массив отсортирован.

Попробуем также при последовательной сортировке данных увеличить размерность массива. В результате исследования видно, что при больших данных время ожидания выполнения работы алгоритма значительно увеличивается.

Так как целью магистерской работы является использование алгоритмов при использовании больших данных, то при заполнении массива возьмем n=900000000. При компиляции программы падает ошибка работы с памятью, как показано на рисунке 2.3. Для выполнения последовательной программы не хватает пространства памяти.

```
vladislavprudnikov@LinuxPC:~$ time ./merge_sort
Enter array length: 9000000000
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Аварийный останов (сделан дамп памяти)

real    0m14.980s
```

Рис. 2.3. Запуск последовательного алгоритма быстрой сортировки данных.

Последовательный алгоритм не подходит для сортировки больших данных или время ожидания становится очень большим, что просто делает сортировки не подходящей.

Для улучшения программы. Что бы можно было использовать сортировку, попробуем использовать параллельную версию программы.

Проверим работоспособность программы. Введем неотсортированную последовательность и проверим как работа алгоритма отсортируем массив на небольших данных. Как видно на рисунке 3.4 массив отсортировывается.

```
vladislavprudnikov@LinuxPC:~$ ./otasks 50
Unsorted array:
90365816069669158880193995132534660363981147073181
Sorted array:
00000111111112333333344555566666666778888889999999
vladislavprudnikov@LinuxPC:~$ ./otasks 60
Unsorted array:
21756462354168989420886890508890087634061597587429609519561З
Sorted array:
00000001111122223334444455555556666666677778888888899999999
vladislavprudnikov@LinuxPC:~$
```

Рис. 3.4. Результат работы алгоритма быстрой сортировки

Так как тема магистерской работы является разработка алгоритмов сортировки больших данных, увеличим число входных параметров. Запустим алгоритм для сортировки больших данных. Результат работы алгоритма показан на рис. 3.5.

```
vladislavprudnikov@LinuxPC:~$ time ./merge_omp 900000000

real    0m39.236s
```

Рис. 3.5. Время выполнения алгоритма быстрой сортировки при n=900000000

На графике показано как меняется время при увеличении ядер процессора. Самое высокое время работы занимает при последовательной сортировке данных, т.е. на одном ядре. Как показано рис. 3.6 при увеличении ядер время работы уменьшается.

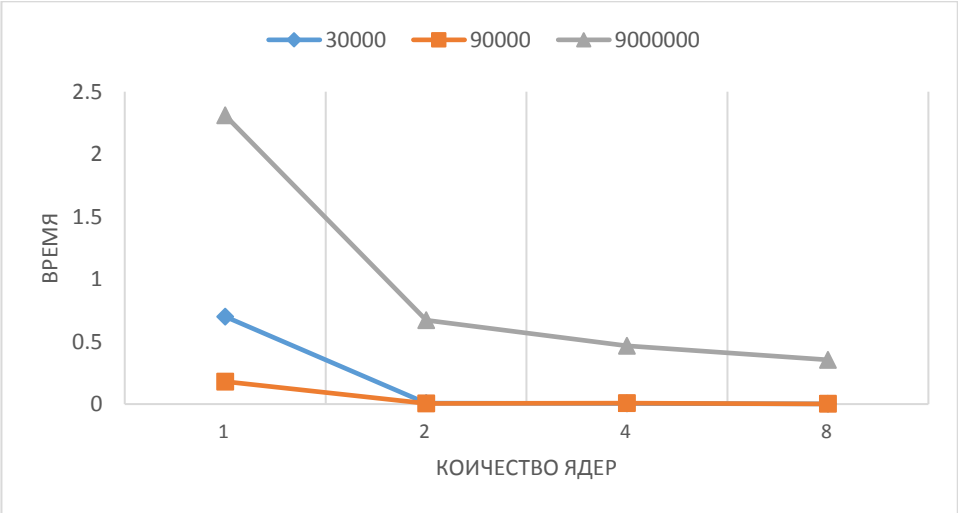

Рис.3.6. График изменения времени при увеличении количества ядер

3.2. Сортировка слиянием

Как и на быстрой сортировке проверим работоспособность алгоритма при последовательном алгоритме. На рис. 3.7 показан результат работы программы при размере массива 30 элементов.

```
vladislavprudnikov@LinuxPC:~$ ./merge_sort
Enter array length: 30
13 10 11 28 5 15 16 8 2 19 12 20 26 4 17 6 1 27 22 9 21 29 7 25 0 23 3 24 18 14
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
```

Рисунок 3.7. Результата работы алгоритма сортировки слиянием

Увеличим размерность массива до более больших данных, т.е возьмем n=900000000. При компиляции программы падает ошибка работы с памятью. Последовательный алгоритм сортировки данных не подходит для большого числа данных, время ожидания работы очень большое, и завершиться алгоритм не может. Для выполнения последовательной программы не хватает пространства памяти, как показано на рисунке 3.8.

```
vladislavprudnikov@LinuxPC:~$ time ./merge_sort
Enter array length: 900000000
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Аварийный останов (сделан дамп памяти)

real    0m13.080s
```

Рисунок 3.8. Результат работы алгоритма последовательной сортировки слиянием при n=900000000

Что бы можно было использовать сортировку, попробуем использовать параллельную версию программы.

Проверим работоспособность программы. Введем неотсортированную последовательность и проверим как работа алгоритма отсортируем массив на небольших данных. Как видно на рисунке 3.9. массив отсортировывается.

```
vladislavprudnikov@LinuxPC:~$ ./merge_omp
List Before Sorting...
15 8 26 25 23 3 17 28 21 14 27 7 20 18 4 6 0 1 22 13 5 9 2 16 10 29 24 19 11 12
List After Sorting...
0 0 1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
Time: 0.00456587
```

Рисунок 3.9. Результат работы алгоритма параллельной версии сортировки слиянием

Увеличим размерность массива как делали при последовательной сортировке данных и запустим параллельный алгоритм сортировки слиянием. Размерность возьмем, как и во всех случаях n=90000000. Результат работы алгоритма показан на рис. 3.10 алгоритм завершил работу без ошибок.

```
Enter array length: 90000000

real    3m49.418s
```

Рисунок 3.10. Время работы алгоритма параллельной сортировки слиянием

В данном работе было реализовано 2 алгоритма. При использовании больших данных, последовательный алгоритмы работы занимают очень много времени. Либо не могут завершить работу, а при использовании параллельных методов, массивы были отсортированы.

На графике показано как меняется время при увеличении ядер процессора при сортировке слиянием. Самое высокое время работы занимает при последовательной сортировке данных, т.е. на одном ядре. Как показано рис. 3.11 при увеличении ядер время работы уменьшается.

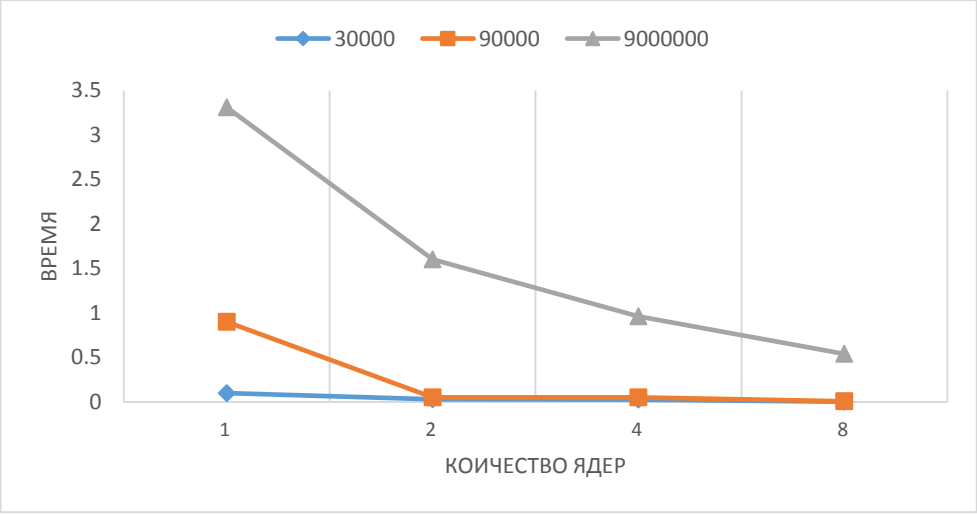

Рис.3.11. График изменения времени при увеличении количества ядер

ЗАКЛЮЧЕНИЕ

В ходе прохождения исследовательской работы была достигнута поставленная цель: исследование и реализация алгоритмов сортировки данных большого объема с использованием параллельных технологий.

При этом были решены следующие задачи:

- изучение и исследование существующих методов сортировки данных;
- изучение и термина «BIG DATA»;
- изучение параллельных технологий;
- сравнительный анализ существующих алгоритмов сортировки данных для больших данных;
- разработка последовательного и параллельного алгоритмов сортировки данных;
-оптимизация алгоритмов сортировки для данных большого объема.

Программы, надлежащее качество проектирования которых позволяет воспользоваться преимуществами параллелизма, могут выполняться быстрее, чем их последовательные эквиваленты, что повышает их рыночную стоимость. Иногда скорость может спасти жизнь. В таких случаях быстрее означает лучше.

Результаты вычислительных экспериментов показали высокую перспективность использования параллельных технологий для выполнения алгоритмов при больших размерностях массива.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Knuth, D. E. (1997). The Art of Computer Programming. Volume 3: Sorting and Searching, second edition. – Reading, MA: Addison-Wesley
2. Viktor Mayer-Schönberger. BIG DATA/ Viktor Mayer-Schönberger, Kenneth Cukier:, 2014 – 240с
3. Афанасьев К.Е. Многопроцессорные вычислительные системы и параллельное программирование / Афанасьев К.Е. - Кемерово: Кузбассвузиздат, 2003 – 268с
4. Антонов А.С. Технологии параллельного программирования MPI и OpenMP / Антонов А.С. – Москва: Издательство МГУ, 2012- 344 с.
5. Бобровский А.Н. Практическое программирование на C++ / Бобровский А.Н. – СПб.: БХВ-Петербург, 2012. – 496 с.
6. Богачев К. Основы параллельного программирования / Богачев К – М.:БИНОМ. Лаборатория заданий, 2003,-342с.
7. Воеводин В.В. Параллельные вычисления / Воеводин В.В. - СПб.: БХВ-Петербург, 2002- 608 с.
8. Вирт Н. Алгоритмы и структуры данных / Вирт Н. - ИздательсвоМир, 1989, 360 стр.
9. Гергель В.П. Современные языки и технологии параллельного программирования / Гергель В.П. - Издательство Московского Университета, 2012. - 408 с.
10. Грегори Р. Э. Основы многопоточного, параллельного и распределенного программирования / Грегори Р. Э. – Москва, Изд.: Вильямс, 2003. — 512 с.
11. Есаулов А.О. Параллельные вычисления на многопроцессорных вычислительных системах / Есаулов А.О, Старченко А.В - Томск: ТГУ, 2002 - 120
12. Деменев А.Г Параллельные вычислительные системы: основы программирования и компьютерного моделирования / Деменев А.Г - Пермь: ПГПУ, 2001 – 165с

13. Дупленко А. Г. Сравнительный анализ алгоритмов сортировки данных в массивах / Дупленко А. Г. - Молодой ученый. — 2013. — №8. — С. 50-53.

14. Кормен Т. Алгоритмы: построение и анализ / Кормен Т., Лейзерсон Ч., Ривест Р. - Москва: МЦНТО, 1999 – 863с

15. Касперски К. Техника оптимизации программ. Эффективное использование памяти. / Касперски К. - Спб.: БХВ-Петербург, 2003. – 464с

16. Кнут Д. Искусство программирования для ЭВМ/ Кнут Д. – Москва, изд.: Издательский дом "Вильямс", 2007 – 160с

17. Лабутина А.А. Учебный курс "Введение в методы параллельного программирования" / Лабутина А.А. – ННГУ, Нижний Новгород, 2007, 42 с.

18. Малышкин В.Э. Параллельное программирование мультикомпьютеров / Малышкин В.Э., Корнеев В.Д. - Изд-во НГТУ, 2011.

19. Немнюгин С.А. Параллельное программирование для многопроцессорных вычислительных систем / Немнюгин С.А., Стесик О.Л. - БХВ-Петербург, 2002 - 400 с.

20. Пышкин Е. В. Структуры данных и алгоритмы: реализация на C/C++ / Пышкин Е. В. - СПб.: ФТК СПБГПУ, 2009.- 200 с

21. Роберт Седжвик. Фундаментальные алгоритмы на С. Структуры данных / Роберт Седжвик – Санкт-Петербург, СПб.: ДиаСофтЮП, 2003, 672 с.

22. Скиена С. Алгоритмы. Руководство по разработке / Скиена С. — 2-е изд.,СПб.: БХВ-Петербург, 2011. — 720 с.

23. Якобовский М.В. Последовательности псевдослучайных чисел для многопроцессорных приложений / Якобовский М.В. – 2008-480с

24. Алгоритмы. Построение и анализ / Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест и др. - Москва: Изд-во МЦНМО, 2016 – 1328с

25. Анализ эффективности алгоритмов / [Электронный ресурс] / Режим доступа: https://www.slideshare.net/mkurnosov/dsa-spring2015lec1

26. Блог программиста / [Электронный ресурс] / Режим доступа: https://pro-prof.com/archives/1660

27. Большие данные (Big Data) / [Электронный ресурс] / Режим доступа: http://www.tadviser.ru/index.php/Статья:Большие_данные_(Big_Data)

28. Википедия — свободная энциклопедия / [Электронный ресурс] / Режим доступа: https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

29. Википедия — свободная энциклопедия / [Электронный ресурс] / Режим доступа: https://ru.wikipedia.org/wiki/Алгоритм_сортировки

30. Википедия — свободная энциклопедия / [Электронный ресурс] / Режим доступа: https://ru.wikipedia.org/wiki/Большие_данные

31. Введение в технологии параллельного программирования (OpenMP) / [Электронный ресурс] / Режим доступа: http://www.intuit.ru/studies/courses/4447/983/lecture/14925

32. Интуит - Алгоритмы сортировки массивов/ [Электронный ресурс] / Режим доступа: /http://www.intuit.ru/studies/courses/648/504/lecture/11473

33. Интуит. Лекция по теме «Параллельные методы сортировки» / [Электронный ресурс] / Режим доступа: http://www.intuit.ru/studies/courses/1156/190/lecture/4958

34. Параллельные алгоритмы сортировки больших объемов данных/ [Электронный ресурс] / Режим доступа: http://lira.imamod.ru/FondProgramm/Sort/ParallelSort.pdf

35. Сортировка массива / [Электронный ресурс] / Режим доступа: http://edunow.su/site/content/algorithms/sortirovka_massiva

36. Сортировка данных / [Электронный ресурс] / Режим доступа: http://www.math.csu.ru/~rusear/DipKurs/ParMetSort.html

37. Сортировка целых чисел при нехватке памяти / [Электронный ресурс] / Режим доступа: https://habrahabr.ru/post/266557/

38. Что такое большие данные / [Электронный ресурс] / Режим доступа: https://habrahabr.ru/post/305738/

39. Что такое Big Data? / [Электронный ресурс] / Режим доступа: https://postnauka.ru/faq/46974

40. Supercomputer Software Department/ [Электронный ресурс] / Режим доступа: ssd.sscc.ru/ru/content/параллельные-алгоритмы-сортировки

ПРИЛОЖЕНИЯ

Таблица 1.3

Сравнение алгоритмов

Сортировка	**Лучший**	**Средний**	**Наихудший**	**Память**	**Устойчивость**
Сортировка Пузырьком	*n*	n^2	n^2	1	да
Сортировка перемешиванием	*n*	n^2	n^2	1	да
Гномья сортировка	*n*	n^2	n^2	1	да
Сортировка вставками	*n*	n^2	n^2	1	да
Сортировка слиянием	*nlog n*	*nlog n*	*nlogn*	n Гибридная сортировка слияния блоков - O (1) mem.	да
Сортировка двоичного дерева	*nlog n*	*nlog n*	*nlogn*	n	да
Timsort	*n*	*nlog n*	*nlogn*	n	да
Сортировка подсчетом		*n+r*	*n+r*	n+r	да
Блочная сортировка	*n*	*nlog n*	*nlogn*	1	да
Сортировка выбором	n^2	n^2	n^2	1	нет
Сортировка расческой	*n*	n^2	n^2	1	нет

Сортировка Шелла	*nlogn*	*nlog²n*	Зависит от последовательности; Самым известным является $nlog^2n$	1	нет
Пирамидальная сортировка	*nlogn*	*nlogn*	*nlogn*	1	нет
Плавная сортировка	*n*	*nlogn*	*nlogn*	1	нет
Быстрая сортировка	*nlogn*	*nlogn*	n^2	logn в среднем, наихудшая сложность пространства n ;Изменение logn худший случай.	Типичная сортировка на месте нестабильна; Существуют стабильные версии.
Интроспективная сортировка	*nlogn*	*nlogn*	*nlogn*	logn	нет
Терпеливая сортировка	*n*	-	*nlogn*	n	нет

Последовательная быстрая сортировка сортировка

```
#include <stdio.h>
#include <math.h>
#include <vector>
#include <iostream>

using namespace std;

void sort(vector<int> &v, int b0, int e0 ){ //алгоритм сортировка
	int d = v[e0]; //элемент массива, с которым будет сравниваться текущий элемент
	int b = b0; //начальный элемент сортировка
	int e = e0; //конечный элемент сортировка

	do{ //производим изменения начала и конца до тех пор, пока начало меньше или равно концу
		while(v[b] < d){ //увеличиваем начальный элемент сортировки, пока текущий элемент меньше того, с которого нужно начинать сортировку
			++b;
		}
			while(v[e] > d){ //уменьшаем конечный элемент сортировки, если текущий элемент больше, чем элемент, которым должна заканчиваться сортировка
				--e;
			}
		if(b <= e){ //когда начальный элемент, меньше конечного элемента, меняем их местами
			swap(v[b],v[e]);
			++b; //обновляем начальный указатель сортировки
			--e; //обновляем конечный указатель сортировки
		}
	} while(b<=e);

	//разделение массива на две части

	if(e > b0){
		sort(v,b0,e);
	}

	if(b < e0){
		sort(v, b,e0);
	}
}
int main(){
	vector<int> v; //создаём массив
	for(int i=0; i<20; ++i){ //заполняе его
		v.push_back(i);
	}
	for(size_t i=0; i<v.size(); ++i){  //перемешиваем элементы
		swap(v[i], v[rand() % (v.size() -i) +i]);
```

```
	}

	for(int i=0; i<v.size(); ++i){ // выводим массив
		if(i==v.size()-1){
			cout<<v[i]<<endl;
		}else{
			cout<<v[i]<<" ";
		}
	}
	sort(v, 0, v.size()-1); //отсортировать от 0 до 19
	//sort(какой массив, начиная с какого элемента, по какой элемент);

	for(int i=0; i<v.size(); ++i){ //выводим отсортированный массив
		if(i==v.size()-1){
			cout<<v[i]<<endl;
		}else{
			cout<<v[i]<<" ";
		}	}}
```

Параллельная быстрая сортировка

```
#include <omp.h>
#include <algorithm>
#include <stdio.h>
#include <time.h>

//#define DEBUG

/*!
* quickSort - реализация алгоритма быстрой сортировки
* @param num номер вызова функции (для отладки)
* @param a сортируемый массив
* @param n индекс последнего элемента массива (не размер массива!)
*/
void quickSort(int num, float* a, const long n) {
	long i = 0, j = n;
	float	pivot = a[n / 2]; // опорный элемент

#ifdef DEBUG
	#pragma omp critical
		{
			printf("enter: %d\n", num);
		}
#endif

	do {
		while (a[i] < pivot) i++;
		while (a[j] > pivot) j--;
```

```
                if (i <= j) {
    std::swap(a[i], a[j]);
                        i++; j--;
                }
        } while (i <= j);

#ifdef DEBUG
        #pragma omp critical
                {
                        printf("divide #%d by %3.2f [%d] -> (%d|%d) : %d\n", num, pivot, n/2, j, n - i,
n);
                }
#endif

        if (n < 100) {
                if (j > 0) quickSort(num * 2 + 1, a, j);
                if (n > i) quickSort(num * 2 + 2, a + i, n - i);
                return;
        }

#pragma omp task shared(a)
        {
                if (j > 0) quickSort(num * 2 + 1, a, j);
        } // #pragma omp task
#pragma omp task shared(a)
        {
                if (n > i) quickSort(num * 2 + 2, a + i, n - i);
        } // #pragma omp task
#pragma omp taskwait
}
int main(int argc, char *argv[])
        if (argc < 2) {
                printf("error arg\n");
                return 1;
 }

        float *a;
        long n;
 sscanf(argv[1], "%d", &n);
        a = new float[n];
        srand(time(NULL));
        for (int i = 0; i < n; ++i)
                a[i] = rand() % 10;

#ifdef DEBUG
        for (int i = 0; i < n; ++i)
                printf("%3.2f ", a[i]);
```

```
        printf("\n");
#endif

#pragma omp parallel shared(a)
        {
                #pragma omp single nowait
                {
                        quickSort(0, a, n - 1);
                } // #pragma omp single
        } // #pragma omp parallel

 /* проверка правильности сортировки
        for (int i = 1; i < n; ++i) {
                if (a[i] < a[i-1])
    printf("error\n");
        }
        */

 delete []a;
}
```

Последовательная сортировка слиянием

```
#include <stdio.h>
#include <math.h>
#include <vector>
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <time.h>
#include <ctime>
#include <sys/time.h>
#include <omp.h>
using namespace std;
void mergeSort(vector<int> &a, size_t start, size_t end){
        if(end - start < 2){
                return;
        }
        if(end - start == 2){
                if(a[start] > a[start+1]){
                        swap(a[start], a[start+1]);
                }
        }
        mergeSort(a, start, start + (end-start) / 2);
        mergeSort(a, start + (end-start) /2, end);
        vector<int> b;
        size_t b1 = start;
        size_t e1 = start + (end - start) / 2;
        size_t b2 = e1;
```

```
        while(b.size() < end - start){

                if(b1 >= e1 || (b2 < end && a[b2] <= a[b1])){
                        b.push_back(a[b2]);
                        ++b2;
                }else{
                        b.push_back(a[b1]);
                        ++b1;
                }
        }
        for(size_t i = start; i< end; ++i){
                a[i] = b[i - start];
        }
}
int main(){
        long int length = 0;
        cout<<"Enter array length: ";
        cin>>length;
        vector<int> v; //создаём массив
        for(int i=0; i<length; ++i){ //заполняе его
                v.push_back(i);
        }
        for(size_t i=0; i<v.size(); ++i){  //перемешиваем элементы
                swap(v[i], v[rand() % (v.size() -i) +i]);
        }
}
```

Параллельная сортировка слиянием

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "omp.h"
#define MAX_SIZE 1000
void generate_list(int * x, int n) {
  int i,j,t;
  for (i = 0; i < n; i++)
    x[i] = i;
  for (i = 0; i < n; i++) {
    j = rand() % n;
    t = x[i];
    x[i] = x[j];
    x[j] = t;
  }
}
void print_list(int * x, int n) {
  int i;
  for (i = 0; i < n; i++) {
     printf("%d ",x[i]);
```

```
  }
}
void merge(int * X, int n, int * tmp) {
  int i = 0;
  int j = n/2;
  int ti = 0;
  while (i<n/2 && j<n) {
    if (X[i] < X[j]) {
      tmp[ti] = X[i];
      ti++; i++;
    } else {
      tmp[ti] = X[j];
      ti++; j++;
    }
  }
  while (i<n/2) { /* finish up lower half */
    tmp[ti] = X[i];
    ti++; i++;
  }
    while (j<n) { /* finish up upper half */
      tmp[ti] = X[j];
      ti++; j++;
  }
  memcpy(X, tmp, n*sizeof(int));
} // end of merge()
void mergesort(int * X, int n, int * tmp)
{
  if (n < 2) return;
  #pragma omp task firstprivate (X, n, tmp)
  mergesort(X, n/2, tmp);
  #pragma omp task firstprivate (X, n, tmp)
  mergesort(X+(n/2), n-(n/2), tmp);
  #pragma omp taskwait
   /* merge sorted halves into sorted list */
  merge(X, n, tmp);
}
int main()
{
  int n = 100;
  double start, stop;
  int data[MAX_SIZE], tmp[MAX_SIZE];
    generate_list(data, n);
  printf("List Before Sorting...\n");
  print_list(data, n);
  start = omp_get_wtime();
  #pragma omp parallel
  {
    #pragma omp single
```

```
    mergesort(data, n, tmp);
  }
  stop = omp_get_wtime();
  printf("\nList After Sorting...\n");
  print_list(data, n);
  printf("\nTime: %g\n",stop-start);
}
```