**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Федеральное государственное автономное   
образовательное учреждение

Высшего образования

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Инженерная школа информационных технологий и робототехники

Направление: «Программная инженерия»

Отчет по лабораторной работе №8 по дисциплине

**«Параллельное, распределенное программирование»**

Выполнил:

Студент группы 8К61 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Исламов Е.Р.

Принял:

Доцент ОИТ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Мокина Е.Е.

Томск 2019

**Содержание**

[Цель работы 3](#_Toc23454460)

[Ход работы 4](#_Toc23454461)

[Вывод 15](#_Toc23454462)

# Цель работы

Познакомиться с параллельными алгоритмами сортировки.

Разработать программы, реализующие последовательный и параллельный алгоритм сортировки с использованием регулярного набора образцов. Сравнить результаты на наборах данных с разным количеством. Построить график.

# Ход работы

В ходе работы была разработана программа, позволяющая осуществлять быструю сортировку и параллельную сортировку с использованием регулярного набора образцов. Было проведено тестирование алгоритмов для разного набора данных.

Класс QuickSort предоставляет алгоритм быстрой сортировки.

**package** Lab8;  
  
**import** java.util.Collections;  
**import** java.util.List;  
  
*/\*\*  
 \* Быстрая сортировка.  
 \*/***public class** QuickSort {  
 */\*\*  
 \* Быстрая сортировка.  
 \** ***@param list*** *Список, который нужно отсортировать.  
 \** ***@param begin*** *Нижняя граница сортировки.  
 \** ***@param end*** *Верхняя граница сортировки.  
 \*/* **public static void** quickSort(List<Integer> list, **int** begin, **int** end) {  
 **if** (begin < end) {  
 **int** partitionIndex = *partition*(list, begin, end);  
  
 *quickSort*(list, begin, partitionIndex-1);  
 *quickSort*(list, partitionIndex+1, end);  
 }  
 }  
  
 */\*\*  
 \* Разделить на две части и обменяться элементами соответственно.  
 \** ***@param list*** *Список.  
 \** ***@param begin*** *Нижняя граница.  
 \** ***@param end*** *Верхняя граница.  
 \** ***@return*** *Центральная точка.  
 \*/* **private static int** partition(List<Integer> list, **int** begin, **int** end) {  
 **int** pivot = list.get(end);  
 **int** i = (begin-1);  
  
 **for** (**int** j = begin; j < end; j++) {  
 **if** (list.get(j) <= pivot) {  
 i++;  
  
 Collections.*swap*(list, i, j);  
 }  
 }  
  
 Collections.*swap*(list, i+1, end);  
 **return** i+1;  
 }  
}

В классе PSRS реализован параллельный алгоритм сортировки с использованием регулярного набора образцов с произвольным числом потоков.

**package** Lab8;  
  
**import** java.util.\*;  
**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
  
*/\*\*  
 \* Параллельная сортировка с использованием регулярного набора образцов.  
 \*/***final class** PSRS {  
 */\*\*  
 \* Барьер для синхронизации потоков.  
 \*/* **private** CyclicBarrier **barrier**;  
  
 */\*\*  
 \* Словарь вида "НОМЕР ПОТОКА - СПИСОК ГРАНИЦ".  
 \*/* **private** Map<Integer, List<Bound>> **threadBoundMap** = **new** HashMap<Integer, List<Bound>>();  
  
 */\*\*  
 \* Исходный список элементов.  
 \*/* **private** List<Integer> **sourceList**;  
  
 */\*\*  
 \* Результирующий список элементов.  
 \*/* **private** Integer[] **resultList**;  
  
 */\*\*  
 \* Образцы.  
 \*/* **private** List<Integer> **samples**;  
  
 */\*\*  
 \* Ведущие элементы.  
 \*/* **private** List<Integer> **pivots**;  
  
 */\*\*  
 \* Число блоков (разделений) для каждого потока.  
 \*/* **private** Integer[] **partitionsSize**;  
  
 */\*\*  
 \* Число потоков (процессоров).  
 \*/* **private int threadsCount**;  
  
 */\*\*  
 \* Конструктор.  
 \*  
 \** ***@param threadsCount*** *Число потоков.  
 \*/* **private** PSRS(**int** threadsCount) {  
 **this**.**threadsCount** = threadsCount;  
 **samples** = Collections.*synchronizedList*(**new** ArrayList<Integer>());  
 **barrier** = **new** CyclicBarrier(threadsCount);  
 **partitionsSize** = **new** Integer[threadsCount];  
 }  
  
 */\*\*  
 \* Отсортировать массив.  
 \*  
 \** ***@param threadsCount*** *Число потоков  
 \** ***@param source*** *Исходный массив.  
 \** ***@return*** *Отсортированный массив.  
 \*/* @SuppressWarnings(**"SameParameterValue"**)  
 **static** List<Integer> sort(**int** threadsCount, List<Integer> source) {  
 PSRS psrs = **new** PSRS(threadsCount);  
 **return** psrs.parentSort(source);  
 }  
  
 */\*\*  
 \* Запуск потоков сортировки и ожидание их выполнения.  
 \*  
 \** ***@param sourceList*** *Исходный массив.  
 \** ***@return*** *Отсортированный массив.  
 \*/* **private** List<Integer> parentSort(**final** List<Integer> sourceList) {  
 **this**.**sourceList** = sourceList;  
 **this**.**resultList** = **new** Integer[sourceList.size()];  
  
 List<Thread> threads = **new** ArrayList<>();  
 **for** (**int** p = 0; p < **threadsCount**; p++) {  
 **final int** pTemp = p;  
 Thread t = **new** Thread(() -> childSort(pTemp));  
 t.start();  
 threads.add(t);  
 }  
  
 **for** (Thread t : threads) {  
 **try** {  
 t.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 **return** Arrays.*asList*(**resultList**);  
 }  
  
 */\*\*  
 \* Непосредственная сортировка. Выполняется несколькими потоками одновременно.  
 \*  
 \** ***@param p*** *Номер потока.  
 \*/* **private void** childSort(**int** p) {  
 *// ЭТАП 1.  
 // Получаем границы сортировки для потока.* Bound localBound = getBounds(p);  
 *// Осуществляем быструю сортировку.* QuickSort.*quickSort*(**sourceList**, localBound.**low**, localBound.**high**);  
  
 *// Формируем для каждого потока наборы образцов и добавляем в общий список.* List<Integer> sample = getSample(localBound.**low**, localBound.**high**);  
 **samples**.addAll(sample);  
  
 *// Ждем, пока все потоки завершат этап подготовки образцов.* barrierAwait(**barrier**);  
  
 *// ЭТАП 2.  
 // Сортируем регулярный набор образцов  
 // и формируем список ведущих элементов.* **if** (p == 0) {  
 QuickSort.*quickSort*(**samples**, 0, **samples**.size() - 1);  
 **pivots** = getPivots(**samples**);  
 }  
  
 barrierAwait(**barrier**);  
  
 *// ЭТАП 3.  
 // Разделение блока потока на несколько частей с использованием  
 // полученного набора опорных точек.* partitionAndBroadcast(localBound);  
 barrierAwait(**barrier**);  
  
 *// ЭТАП 4. Слияние полученных частей в отсортированный блок.  
 // Формируем число блоков для каждого потока.* **if** (p == 0) {  
 **for** (**int** i = 0; i < **threadsCount**; i++) {  
 **partitionsSize**[i] = findLocalListSize(i);  
 }  
 }  
 barrierAwait(**barrier**);  
  
 *// Сливаеем полученные части в отсортированный блок.* mergePartitions(p);  
 }  
  
 */\*\*  
 \* Получить границы сортировки для потока.  
 \*  
 \** ***@param p*** *Номер потока.  
 \** ***@return*** *Границы сортировки.  
 \*/* **private** Bound getBounds(**int** p) {  
 Bound b = **new** Bound();  
 **int** n = **sourceList**.size();  
 **int** elemsPerProc = (**int**) Math.*ceil*((**float**) n / (**float**) **threadsCount**);  
 b.**low** = p \* elemsPerProc;  
 b.**high** = b.**low** - 1 + elemsPerProc;  
 **if** (b.**high** > **sourceList**.size() - 1) {  
 b.**high** = **sourceList**.size() - 1;  
 }  
 **return** b;  
 }  
  
 */\*\*  
 \* Формирование регулярного набора образцов по формуле m=n/p^2  
 \* для 0, m, 2m, ..., (p-1)\*m.  
 \*  
 \** ***@param low*** *Нижняя граница.  
 \** ***@param high*** *Верхняя граница.  
 \** ***@return*** *Набор образцов.  
 \*/* **private** List<Integer> getSample(**int** low, **int** high) {  
 List<Integer> sample = **new** ArrayList<>();  
 **int** n = **sourceList**.size();  
 **for** (**int** i = 0; i < **threadsCount**; i++) {  
 **int** index = ((i \* n) / (**threadsCount** \* **threadsCount**)) + 1 + low;  
 **if** (index > high) {  
 index = high;  
 }  
 sample.add(**sourceList**.get(index - 1));  
 }  
 **return** sample;  
 }  
  
 */\*\*  
 \* Получить набор ведущих элементов списка по формуле  
 \* (p-1) \* p + p / 2, где p - число потоков.  
 \*  
 \** ***@param list*** *Список элементов.  
 \** ***@return*** *Список ведущих элементов списка.  
 \*/* **private** List<Integer> getPivots(List<Integer> list) {  
 List<Integer> pivots = **new** ArrayList<>();  
 **for** (**int** i = 1; i < **threadsCount**; i++) {  
 **int** index = (i \* **threadsCount** + (**int**) Math.*floor*(**threadsCount** >> 1));  
 **if** (index > list.size() - 1) {  
 index = list.size() - 1;  
 }  
 pivots.add(list.get(index - 1));  
 }  
 **return** pivots;  
 }  
  
 */\*\*  
 \* Ожидание потоков.  
 \*  
 \** ***@param barrier*** *Барьер.  
 \*/* **private void** barrierAwait(CyclicBarrier barrier) {  
 **try** {  
 barrier.await();  
 } **catch** (InterruptedException | BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* Выделить части своего блока и разослать всем остальным потокам системы.  
 \*  
 \** ***@param b*** *Границы потока.  
 \*/* **private void** partitionAndBroadcast(Bound b) {  
 Bound newBound = **new** Bound();  
 newBound.**low** = b.**low**;  
 **for** (**int** pivotIndex = 0; pivotIndex <= **pivots**.size() - 1; pivotIndex++) {  
 **int** i = newBound.**low**;  
 i = findNextHighBound(pivotIndex, i, b.**high**);  
 newBound.**high** = i - 1;  
 addBound(pivotIndex, newBound);  
 newBound = **new** Bound();  
 newBound.**low** = i;  
 }  
  
 **if** (newBound.**low** < b.**high**) {  
 **int** i = newBound.**low**;  
 **while** (i < b.**high**) {  
 i++;  
 }  
 newBound.**high** = i;  
 addBound(**pivots**.size(), newBound);  
 }  
 }  
  
 */\*\*  
 \* Найти следующую верхнюю границу.  
 \*  
 \** ***@param pivotIndex*** *Индекс опорного элемента.  
 \** ***@param currIndex*** *Текущий индекс.  
 \** ***@param high*** *Максимальная верхняя граница блока потока.  
 \** ***@return*** *Индекс верхней границы.  
 \*/* **private int** findNextHighBound(**int** pivotIndex, **int** currIndex, **int** high) {  
 Integer pivotValue = **pivots**.get(pivotIndex);  
 **while** (**sourceList**.get(currIndex) <= pivotValue && currIndex <= high) {  
 currIndex++;  
 }  
 **return** currIndex;  
 }  
  
 */\*\*  
 \* Слияние(запись) маленьких отсортированных массивов в один большой отсортированный массив.  
 \*  
 \** ***@param p*** *Номер потока.  
 \*/* **private void** mergePartitions(**int** p) {  
 List<Bound> boundList = getBoundList(p);  
 **int** currIndex = findStartIndex(p);  
 **while** (boundList.size() > 0) {  
 Bound lowest = findNextLowest(boundList);  
 **resultList**[currIndex] = **sourceList**.get(lowest.**low**);  
 lowest.**low**++;  
 currIndex++;  
 **if** (lowest.**low** > lowest.**high**) {  
 boundList.remove(lowest);  
 }  
 }  
 }  
  
 */\*\*  
 \* Найти стартовый индекс слияния для потока.  
 \*  
 \** ***@param p*** *Номер потока.  
 \** ***@return*** *Начальный индекс.  
 \*/* **private int** findStartIndex(**int** p) {  
 **int** startIndex = 0;  
 **for** (**int** i = 0; i < p; i++) {  
 startIndex += **partitionsSize**[i];  
 }  
 **return** startIndex;  
 }  
  
 */\*\*  
 \* Получить количество частей, полученных после  
 \* разделения блоков на несколько частей для конкретного потока.  
 \*  
 \** ***@param p*** *Индекс потока.  
 \** ***@return*** *Количество частей.  
 \*/* **private int** findLocalListSize(**int** p) {  
 List<Bound> boundList = getBoundList(p);  
 **int** size = 0;  
 **for** (Bound b : boundList) {  
 size += (b.**high** - b.**low**) + 1;  
 }  
 **return** size;  
 }  
  
 */\*\*  
 \* Найти элемент списка границ с самой высокой нижней границей.  
 \*  
 \** ***@param boundList*** *Список границ.  
 \** ***@return*** *Элемент списка с самой высокой нижней границей.  
 \*/* **private** Bound findNextLowest(List<Bound> boundList) {  
 Bound lowest = **null**;  
 **for** (Bound b : boundList) {  
 Integer currValue = **sourceList**.get(b.**low**);  
 **if** (lowest == **null** || currValue < **sourceList**.get(lowest.**low**))  
 lowest = b;  
 }  
 **return** lowest;  
 }  
  
 */\*\*  
 \* Добавить границы (часть блока) в список границ потока.  
 \*  
 \** ***@param p*** *Номер потока.  
 \** ***@param b*** *Границы.  
 \*/* **private synchronized void** addBound(**int** p, Bound b) {  
 List<Bound> boundList = getBoundList(p);  
 boundList.add(b);  
 }  
  
 */\*\*  
 \* Получить список границ (частей блока) потока.  
 \*  
 \** ***@param p*** *Номер потока.  
 \** ***@return*** *Список границ.  
 \*/* **private** List<Bound> getBoundList(**int** p) {  
 **return threadBoundMap**.computeIfAbsent(p, k -> **new** ArrayList<>());  
 }  
}

Класс Bound представляет собой вспомогательный класс интервала.

**package** Lab8;  
  
*/\*\*  
 \* Границы.  
 \*/***class** Bound {  
 */\*\*  
 \* Нижняя граница.  
 \*/* **int low**;  
  
 */\*\*  
 \* Верхняя граница.  
 \*/* **int high**;  
}

Класс Main обладает рядом вспомогательных методов, таким как засечка времени, генерация и клонирование случайных списков и др, а также позволяет протестировать вышеуказанные алгоритмы для разного размера исходного списка.

**package** Lab8;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.concurrent.ThreadLocalRandom;  
**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.atomic.AtomicReference;  
  
*/\*\*  
 \* Входная точка.  
 \*/***public class** Main {  
 */\*\*  
 \* Количество потоков.  
 \*/* **private static final int *THREADS\_COUNT*** = 4;  
  
 */\*\*  
 \* Входная точка.  
 \*  
 \** ***@param args*** *Параметры запуска.  
 \*/* **public static void** main(String[] args) {  
 *benchmark*(0, 5000, 10000);  
 *benchmark*(0, 5000, 50000);  
 *benchmark*(0, 5000, 100000);  
 *benchmark*(0, 5000, 200000);  
 *benchmark*(0, 5000, 500000);  
 *benchmark*(0, 5000, 1000000);  
 *benchmark*(0, 5000, 2000000);  
 *benchmark*(0, 5000, 5000000);  
 }  
  
 */\*\*  
 \* Замерить время выполнения сортировки с использованием регулярного набора образцов и быстрой сортировки.  
 \** ***@param minValue*** *Минимальное значение элемента списка.  
 \** ***@param maxValue*** *Максимальное значение элемента списка.  
 \** ***@param listSize*** *Количество элементов списка.  
 \*/* **public static void** benchmark(**int** minValue, **int** maxValue, **int** listSize) {  
 System.***out***.printf(**"Размер списка: %d, минимальное значение: %d, максимальное значение: %d\n"**,  
 listSize, minValue, maxValue);  
 List<Integer> unsorted = *generateRandomList*(minValue, maxValue, listSize);  
 List<Integer> unsortedClone = *cloneList*(unsorted);  
  
 AtomicReference<List<Integer>> PSRSTemp = **new** AtomicReference<>();  
 **long** msPSRS = *timer*(() -> {  
 PSRSTemp.set(PSRS.*sort*(***THREADS\_COUNT***, unsorted));  
 }, TimeUnit.***MILLISECONDS***);  
 List<Integer> PSRSResult = PSRSTemp.get();  
  
 **long** msQS = *timer*(() -> {  
 QuickSort.*quickSort*(unsortedClone, 0, unsortedClone.size() - 1);  
 }, TimeUnit.***MILLISECONDS***);  
  
 System.***out***.printf(**"Время выполнения PSRS: %d.\n"**, msPSRS);  
 System.***out***.printf(**"Время выполнения быстрой сортировки: %d\n"**, msQS);  
  
 System.***out***.println(**"Сортировка выполнена корректно для обоих алгоритмов: "** + (*isOrdered*(PSRSResult) && *isOrdered*(unsortedClone)) + **"\n\n\n\n"**);  
 }  
  
 */\*\*  
 \* Установить, является ли список отсортированным.  
 \*  
 \** ***@param list*** *Список.  
 \** ***@return*** *Отсортирован ли список.  
 \*/* **private static boolean** isOrdered(List<Integer> list) {  
 **int** prev = list.get(0);  
 **for** (**int** val : list) {  
 **if** (val < prev)  
 **return false**;  
 prev = val;  
 }  
 **return true**;  
 }  
  
 */\*\*  
 \* Сгенерировать список случайных чисел.  
 \*  
 \** ***@param min*** *Минимальное значение числа.  
 \** ***@param max*** *Максимальное значение числа.  
 \** ***@param count*** *Количество чисел.  
 \** ***@return*** *Список случайных чисел.  
 \*/* **private static** List<Integer> generateRandomList(**int** min, **int** max, **int** count) {  
 **return** ThreadLocalRandom  
 .*current*()  
 .ints(min, max)  
 .limit(count)  
 .collect(ArrayList::**new**, ArrayList::add, ArrayList::addAll);  
 }  
  
  
 */\*\*  
 \* Глубокое копирование списка чисел.  
 \*  
 \** ***@param list*** *Список.  
 \** ***@return*** *Копию списка.  
 \*/* **private static** List<Integer> cloneList(List<Integer> list) {  
 List<Integer> clone = **new** ArrayList<Integer>(list.size());  
 **for** (Integer item : list)  
 *//noinspection UnnecessaryUnboxing* clone.add(item.intValue());  
 **return** clone;  
 }  
  
 */\*\*  
 \* Засечь время выполнения метода.  
 \*  
 \** ***@param method*** *Метод.  
 \** ***@param timeUnit*** *Единица измерения.  
 \** ***@return*** *Время выполнения метода.  
 \*/* @SuppressWarnings(**"SameParameterValue"**)  
 **private static long** timer(Runnable method, TimeUnit timeUnit) {  
 **long** time = System.*nanoTime*();  
 method.run();  
 time = System.*nanoTime*() - time;  
 **return** timeUnit.convert(time, TimeUnit.***NANOSECONDS***);  
 }  
}

На рисунке 1 приведен результат работы программы.

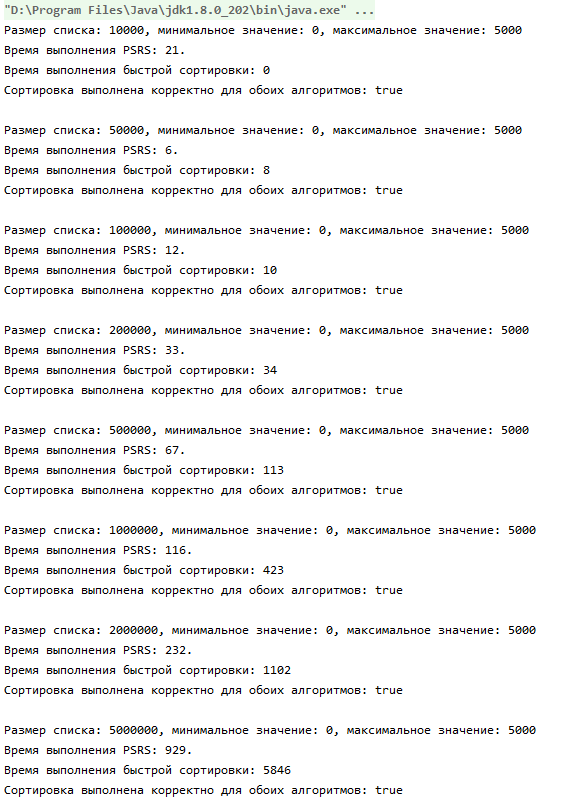


Рисунок 1 - результат работы программы

На рисунке 2 изображен график времени выполнения от длины исходных данных.



Рисунок 2 – график

Из графика видно, что алгоритм параллельной сортировки (синий график) с использованием регулярного набора образцов дает серьезный выигрыш в скорости на больших наборах данных. На небольших наборах данных имеет смысл использован быструю сортировку.

# Вывод

В результате выполненной работы было проведено знакомство с параллельными алгоритмами сортировки. Были разработаны программы, реализующие последовательный и параллельный алгоритм сортировки с использованием регулярного набора образцов.