Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Саратовский государственный технический университет

имени Гагарина Ю.А.»

Институт электронной техники и приборостроения

Кафедра Информационная безопасность автоматизированных систем

Направление 10.05.03 Информационная безопасность автоматизированных систем

**Практическая работа №2**

по дисциплине «Параллельные системы и их программирование»

по теме **«**Алгоритм поиска простых чисел»

|  |  |
| --- | --- |
|  | Выполнил: студент 4 курса  учебной группы с-ИБС42  очной формы обучения  Солодилов В.В.  Проверил: профессор. каф. ИБС  Кондратов Д.В. |

Саратов 2022

**Задание**

Реализовать последовательный и параллельные алгоритмы поиска простых чисел; выполнить анализ быстродействия алгоритмов при разном объеме данных, разном числе потоков; рассчитать ускорение и эффективность выполнения алгоритмов; сделать выводы о целесообразности применения параллельных алгоритмов и необходимости использования синхронизации.

**Вопросы**

1. Какими достоинствами и недостатками обладает каждый вариант распараллеливания?
2. Какие средства синхронизации можно использовать вместо конструкции lock? Какой вариант будет более эффективным?
3. Какой вариант ожидания завершения работ, запущенных пулом потоков, более эффективный и почему?

**Ход работы**

Разработанная в ходе выполнения практической работы программа работает следующим образом: вводится число на проверку, после чего определяется количество потоков, используемых для распараллеливания задач. После чего пользователь может выбрать алгоритм, который можно использовать для получения итогового результата. Результатом выполнения программы является сообщение, сигнализирующее, является ли число простым или составным, а также время, затраченное на выполнение данного алгоритма.

Проведем анализ быстродействия программы для последовательного и параллельных алгоритмов.

*Анализ эффективности многопоточной обработки*

Для анализа эффективности сравним время, затрачиваемое на выполнения задачи для последовательного и двух параллельных алгоритмов.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **M\N** | **1000** | **10000** | **100000** | **1000000** |
| **1** | 730 мкс | 3935 мкс | 29991 мкс | 378702 мкс |
| **2** | 1274 мкс | 4186 мкс | 25480 мкс | 261999 мкс |
| **3** | 1708 мкс | 4715 мкс | 30037 мкс | 173013 мкс |
| **4** | 1890 мкс | 4717 мкс | 28014 мкс | 175097 мкс |
| **5** | 2086 мкс | 5114 мкс | 23073 мкс | 232194 мкс |
| **10** | 2700 мкс | 2760 мкс | 28486 мкс | 266454 мкс |

Таблица 1. Время выполнения операции при последовательном алгоритме и параллельном с декомпозицией по данным

Для анализа эффективности в таблице 1 заданы четыре числа, которые анализируются с помощью последовательного алгоритма и параллельного, осуществляющего декомпозицию по данным.

В основе последовательного алгоритма лежит последовательный перебор переборе уже известных простых чисел, начиная с двойки, и проверке разложимости всех чисел диапазона (m, n] на найденное простое число m.

Идея распараллеливания с декомпозицией по данным заключается в разбиении диапазона \sqrt{n}…nна равные части. Каждый поток обрабатывает свою часть чисел, проверяя на разложимость по каждому базовому простому числу.

Сравним полученные результаты, проанализировав те же числа, но используя параллельный алгоритм с декомпозицией набора простых чисел.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **M\N** | **1000** | **10000** | **100000** | **1000000** |
| **1** | 730 мкс | 3935 мкс | 29991 мкс | 378702 мкс |
| **2** | 1233 мкс | 3276 мкс | 23918 мкс | 313986 мкс |
| **3** | 1375 мкс | 3397 мкс | 23780 мкс | 170982 мкс |
| **4** | 1568 мкс | 3497 мкс | 24722 мкс | 121505 мкс |
| **5** | 1601 мкс | 3704 мкс | 25011 мкс | 190501 мкс |
| **10** | 2578 мкс | 4580 мкс | 27720 мкс | 256935 мкс |

Таблица 2. Время выполнения операции при последовательном алгоритме и параллельном с декомпозицией набора простых чисел

Суть данного метода заключается в том, что в этом алгоритме разделяются базовые простые числа. Каждый поток работает с ограниченным набором простых чисел и проверяет весь диапазон  \sqrt{n}…n.

На основе данных таблицы 1 и таблицы 2, можно сделать вывод, что параллельный алгоритм с декомпозицией набора простых чисел оказался эффективнее при сравнении всех четырех чисел. Для значений N = 1000, 10000 разница несущественна, в увеличение числа потоков не ведёт к уменьшению времени работы программы. Для более крупных значений разница между методами более значительна, а увеличение числа потоков положительно сказывается на быстродействии вплоть до M = 4 для обоих методов. При дальнейшем увеличении числа потоков быстродействие падает из-за необходимости дополнительной обработки средств, используемых на организацию распараллеливания.

**Ответы на вопросы**

1. Все варианты распараллеливания наиболее эффективны только при большом количестве вычисляемых данных, так как при малом объеме элементов массива, накладываются расходы, связанные с организацией многопоточной обработки, которые существенно превышают выигрыш от параллельности обработки. Анализируя эффективность от распараллеливания задачи по частям и от параллельного выполнения цикла, можно сказать, что при первом варианте наибольшая эффективность достигается в случае, если число потоков будет совпадать с числом секций, а второй вариант – более универсальный, но сильно зависит от средств синхронизации потоков.
2. Помимо конструкции *lock* возможно использование конструкции *flush*, которая позволяет синхронизировать состояние памяти. Её выполнение предполагает следующие действия:

* значения всех переменных, временно хранящиеся в регистрах и кэш-памяти текущего потока, будут занесены в основную память;
* все изменения переменных, сделанные потоком во время работы, станут видимы остальным потокам;
* информация, хранящаяся в буферах вывода, будет удалена, т.е. буферы будут сброшены.

Выполнение данной директивы связано со значительными дополнительными расходами, и, если нужна гарантия согласованного представления не всех переменных, их можно явно перечислить в директиве списком. До полного завершения операции никакие действия с перечисленными в ней переменными не могут начаться. В связи с этим, эффективнее всего будет выполнение директивы *lock*.

1. Для ожидания завершения работ, запущенных пулом потоков, можно использовать конструкцию *barrier*. Данный вариант будет наиболее эффективным, поскольку позволяет явно указать точку, где задача, запущенная пулом потоков, завершится.

**Вывод**

В рамках данной практической работы была реализована программа на языке Java, позволяющая определить простое или составное число с применением последовательного алгоритма, а также методов распараллеливания. Анализ эффективности показал, что с увеличением количества потоков производительность параллельного алгоритма возрастает вплоть до 4 потоков, но только с учётом большого числа обрабатываемых элементов. При дальнейшем увеличении их числа производительность падает из-за дополнительных затрат на распараллеливание. В связи с этим применять элементы использование большего числа потоков нецелесообразно как для большого массива обрабатываемых данных, так и для малого, где эффективность распараллеливания даже ниже, чем у последовательного алгоритма.

**Листинг программы**

import java.util.Scanner;

import java.util.concurrent.atomic.AtomicInteger;

public class Main {

static Scanner scanner = new Scanner(System.in);

public static void main(String[] args) throws InterruptedException {

while (true) {

System.out.println("Введите число для проверки: ");

int startValue = scanner.nextInt();

if (startValue == 0) break;

System.out.println("Введите число потоков: ");

int M = scanner.nextInt();

int[] startArray = createStartArray(startValue);

int[] endArray = new int[0];

while (true) {

System.out.print("""

Выберите метод обработки:

1. Последовательный алгоритм

2. Параллельный алгоритм: декомпозиция по данным

3. Параллельный алгоритм: декомпозиция набора простых чисел

4. Параллельный алгоритм: последовательный перебор простых чисел

Введите соответствующее число:

""");

int chkValue = scanner.nextInt();

long time1 = System.nanoTime() / 1000;

if (chkValue == 0) break; else {

switch (chkValue) {

case 1 -> endArray = consistentAlgorithm(startArray, startArray.length);

case 2, 3, 4 -> endArray = parallelAlgorithm(startArray, M, chkValue - 1);

}

}

for (int i = 0; i < endArray.length; i++) {

//System.out.print(endArray[i] + " ");

if (endArray[i] == startValue) {

System.out.println("\nДанное число - простое");

break;

}

if (i == endArray.length - 1) System.out.println("\nДанное число - составное");

}

System.out.println("Время выполнения = " + (System.nanoTime() / 1000 - time1) + " мкс\n");

}

}

}

public static int[] createStartArray(int N) {

int[] startArray = new int[N];

for (int i = 0; i < startArray.length; i++) {

startArray[i] = i + 2;

}

return startArray;

}

public static int[] convertArray(int[] startArray, int[] modifiedArray) {

int[] tempArray = new int[modifiedArray.length];

int j = 0;

for (int i = 0; i < modifiedArray.length; i++) {

if (modifiedArray[i] == 0) {

tempArray[j] = startArray[i];

j++;

}

}

int[] endArray = new int[j];

System.arraycopy(tempArray, 0, endArray, 0, j);

return endArray;

}

public static int[] consistentAlgorithm(int[] startArray, int N) {

int[] sieveArray = new int[N];

for (int i = 2; i <= (int) Math.round(Math.sqrt(N)); i++) {

if (sieveArray[i - 2] == 1) continue;

for (int j = 0; j < sieveArray.length; j++) {

if (sieveArray[j] == 1) continue;

if (startArray[j] == i) sieveArray[j] = 0;

else if ((startArray[j] % i) == 0) {

sieveArray[j] = 1;

} else sieveArray[j] = 0;

}

}

return convertArray(startArray, sieveArray);

}

public static int[] parallelAlgorithm(int[] startArray, int M, int value) throws InterruptedException {

AtomicInteger atomicInteger = new AtomicInteger();

AtomicInteger atomicInteger1 = new AtomicInteger();

AtomicInteger atomicInteger2 = new AtomicInteger();

Object lock = new Object();

int round = (int) Math.round(Math.sqrt(startArray.length));

int[] basicArray = consistentAlgorithm(startArray, round);

int[] modifiedArray = new int[startArray.length];

atomicInteger.set(0);

atomicInteger1.set(round - 2);

atomicInteger2.set(0);

Runnable task1 = () -> { //Декомпозиция по данным

int length = (int) Math.ceil((double)(startArray.length - (round - 2)) / M);

for (int i = atomicInteger1.get(); i <= atomicInteger1.get() + length; i++) {

if (i >= startArray.length) break;

for (int k : basicArray) {

if (startArray[i] % k == 0) {

modifiedArray[i] = 1;

break;

} else modifiedArray[i] = 0;

}

}

atomicInteger1.set(atomicInteger1.get() + length);

};

Runnable task2 = () -> { //Декомпозиция набора базовых простых чисел

int length;

if (M > basicArray.length) length = 1;

else length = (basicArray.length) / M;

for (int i = atomicInteger.get(); i <= atomicInteger.get() + length; i++) {

if (i >= basicArray.length) break;

for (int j = round - 2; j < startArray.length; j++) {

if (modifiedArray[j] == 1) continue;

if (startArray[j] % basicArray[i] == 0) {

modifiedArray[j] = 1;

} else modifiedArray[j] = 0;

}

}

atomicInteger.set(atomicInteger.get() + length);

};

Runnable task3 = () -> { //Последовательный перебор

synchronized (lock) {

for (int i = atomicInteger2.get(); i <= atomicInteger2.get() + 1; i++) {

if (i >= basicArray.length) break;

else for (int j = 0; j < startArray.length; j++) {

if (modifiedArray[j] == 1) continue;

if (startArray[j] % basicArray[i] == 0) {

modifiedArray[j] = 1;

} else modifiedArray[j] = 0;

}

}

atomicInteger2.set(atomicInteger2.get() + 1);

}

};

Thread[] threads = new Thread[M];

for (int i = 0; i < threads.length; i++) {

switch (value) {

case 1 -> {

threads[i] = new Thread(task1);

threads[i].start();

threads[i].join();

}

case 2 -> {

threads[i] = new Thread(task2);

threads[i].start();

threads[i].join();

}

case 3 -> {

threads[i] = new Thread(task3);

threads[i].start();

if (atomicInteger2.get() == basicArray.length) break;

else if ((i + 1) == threads.length) i = 0;

}

}

}

for (int i = 0; i <= (round - 2); i++) {

for (int k : basicArray) {

if (startArray[i] == k) {

modifiedArray[i] = 0;

break;

} else modifiedArray[i] = 1;

}

}

return convertArray(startArray, modifiedArray);

}

}