**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

**Сергиенко Лев Эдуардович**

Отчет по

Лабораторная работа 4

Параллельные вычисления в Java Модели создания и функционирования потоков Пулы потоков в Java

|  |  | **Преподаватель** |
| --- | --- | --- |
|  | ***Кондратьева О.М.*** |
|  | | |

**2025**

# **Задание 1**

Изучить ExecutorService в Java. Решить задачу «Простые числа». Выполнить эксперименты. Сравнить реализации и результаты экспериментов с Модель делегирования 2 из ЛР 3.

## **Текст многопоточной программы.**

| package com.example;  import java.util.ArrayList; import java.util.List; import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors; import java.util.concurrent.Future; import java.util.concurrent.Callable;  public class **PrimeCounterExecutor** {  public static class **PrimeTask** implements **Callable**<**Integer**> {  private final int start;  private final int end;   public **PrimeTask**(int start, int end) {  this.start = start;  this.end = end;  }   @Override  public Integer **call**() {  int count = 0;  for (int i = start; i <= end; i++) {  if (isPrime(i)) {  count++;  }  }  return count;  }   private boolean **isPrime**(int n) {  if (n < 2)  return false;  for (int i = 2; i \* i <= n; i++) {  if (n % i == 0)  return false;  }  return true;  }  }   private int N = 1\_000\_000; // интервал поиска: [2, N]  private int CHUNK\_SIZE = 1000; // размер батча  private int NUM\_WORKERS = Runtime.getRuntime().availableProcessors(); // число воркеров по умолчанию   public **PrimeCounterExecutor**(int N, int CHUNK\_SIZE, int NUM\_WORKERS) {  this.N = N;  this.CHUNK\_SIZE = CHUNK\_SIZE;  this.NUM\_WORKERS = NUM\_WORKERS;  }   public int **countPrimes**() throws Exception {  System.out.println("Подсчет простых чисел в диапазоне [2, " + N + "] с CHUNK\_SIZE = " + CHUNK\_SIZE + " и "  + NUM\_WORKERS + " воркерами.");  long startTime = System.currentTimeMillis();   ExecutorService executor = Executors.newFixedThreadPool(NUM\_WORKERS);  List<Future<Integer>> futures = new ArrayList<>();   for (int i = 2; i <= N; i += CHUNK\_SIZE) {  int end = Math.min(i + CHUNK\_SIZE - 1, N);  futures.add(executor.submit(new PrimeTask(i, end)));  }   int totalPrimes = 0;  for (Future<Integer> future : futures) {  totalPrimes += future.get();  }   executor.shutdown();  long endTime = System.currentTimeMillis();  long elapsed = endTime - startTime;   System.out.println("Найдено простых чисел в диапазоне [2, " + N + "]: " + totalPrimes);  System.out.println("Время выполнения: " + elapsed + " мс");   return totalPrimes;  } } |
| --- |

## **Таблица с результатами экспериментов.**

**Модель делегирования 2 из лаб 3**

| Размерность задачи | Время выполнения последовательной программы | Параллельная программа - 2 потока | | | Параллельная программа - 4 потока | | | Параллельная программа - 16 потока | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность |
| 1 000 000 | 0,132 | 0,08 | 1,65 | 0,825 | 0,051 | 2,588235294 | 0,6470588235 | 0,041 | 3,219512195 | 0,2012195122 |
| 10 000 000 | 2,831 | 1,647 | 1,718882817 | 0,8594414086 | 1,06 | 2,670754717 | 0,6676886792 | 0,473 | 5,985200846 | 0,3740750529 |
| 100 000 000 | 104,632 | 52,277 | 2,001492052 | 1,000746026 | 28,327 | 3,693719773 | 0,9234299432 | 9,038 | 11,57689754 | 0,7235560965 |

**Execulor**

| размер чанка = 500 |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Размерность задачи | Время выполнения последовательной программы, мс | Параллельная программа - 2 потока | | | Параллельная программа - 4 потока | | | Параллельная программа - 16 потока | | |
| Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность |
| 1 000 000 | 108 | 50 | 2,16 | 1,08 | 28 | 3,857142857 | 0,9642857143 | 16 | 6,75 | 0,421875 |
| 10 000 000 | 2279 | 1187 | 1,919966302 | 0,9599831508 | 570 | 3,998245614 | 0,9995614035 | 287 | 7,940766551 | 0,4962979094 |
| 100 000 000 | 58012 | 29346 | 1,976828188 | 0,9884140939 | 14950 | 3,880401338 | 0,9701003344 | 7642 | 7,59120649 | 0,4744504057 |
|  |  |  |  |  |  |  |  |  |  |  |
| размер чанка = 1000 |  |  |  |  |  |  |  |  |  |  |
| Размерность задачи | Время выполнения последовательной программы, мс | Параллельная программа - 2 потока | | | Параллельная программа - 4 потока | | | Параллельная программа - 16 потока | | |
| Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность |
| 1 000 000 | 97 | 48 | 2,020833333 | 1,010416667 | 25 | 3,88 | 0,97 | 13 | 7,461538462 | 0,4663461538 |
| 10 000 000 | 2270 | 1172 | 1,936860068 | 0,9684300341 | 579 | 3,920552677 | 0,9801381693 | 288 | 7,881944444 | 0,4926215278 |
| 100 000 000 | 57861 | 29229 | 1,97957508 | 0,9897875398 | 14894 | 3,884852961 | 0,9712132402 | 7637 | 7,576404347 | 0,4735252717 |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
| размер чанка = 5000 |  |  |  |  |  |  |  |  |  |  |
| Размерность задачи | Время выполнения последовательной программы, мс | Параллельная программа - 2 потока | | | Параллельная программа - 4 потока | | | Параллельная программа - 16 потока | | |
| Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность |
| 1 000 000 | 94 | 49 | 1,918367347 | 0,9591836735 | 24 | 3,916666667 | 0,9791666667 | 13 | 7,230769231 | 0,4519230769 |
| 10 000 000 | 2241 | 1170 | 1,915384615 | 0,9576923077 | 569 | 3,938488576 | 0,9846221441 | 288 | 7,78125 | 0,486328125 |
| 100 000 000 | 56527 | 29292 | 1,929776048 | 0,964888024 | 14776 | 3,82559556 | 0,9563988901 | 7599 | 7,43874194 | 0,4649213712 |
|  |  |  |  |  |  |  |  |  |  |  |
| размер чанка = 10000 |  |  |  |  |  |  |  |  |  |  |
| Размерность задачи | Время выполнения последовательной программы, мс | Параллельная программа - 2 потока | | | Параллельная программа - 4 потока | | | Параллельная программа - 16 потока | | |
| Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность |
| 1 000 000 | 96 | 49 | 1,959183673 | 0,9795918367 | 25 | 3,84 | 0,96 | 13 | 7,384615385 | 0,4615384615 |
| 10 000 000 | 2231 | 1169 | 1,908468777 | 0,9542343884 | 569 | 3,920913884 | 0,980228471 | 288 | 7,746527778 | 0,4841579861 |
| 100 000 000 | 56228 | 28977 | 1,940435518 | 0,9702177589 | 14763 | 3,808710967 | 0,9521777417 | 7573 | 7,424798627 | 0,4640499142 |

## **Сравнительный анализ.**

**Выводы из сравнений:** В целом, ExecutorService показал лучшие результаты по сравнению с Моделью делегирования 2, особенно при увеличении числа потоков. На больших задачах разница в скорости становится особенно заметной. Разница во времени между ExecutorService и Моделью делегирования 2 связана с более эффективным управлением потоками и планированием задач в ExecutorService.

Ускорение и эффективность в ExecutorService демонстрируют лучшие результаты по сравнению с Моделью делегирования 2, особенно при увеличении потоков. На 16 потоках ExecutorService достигает ускорения до 7.4-7.9 раз, в то время как у Модели делегирования 2 оно редко превышает 5.9 раз. При 2 и 4 воркерах ускорение ExecutorService почти линейно.

С увеличением размера чанка от 500 до 10000 производительность улучшается, но прирост после 5000-10000 становится минимальным. Чанки размером 1000 и 5000 демонстрируют лучшую сбалансированность между накладными расходами и загрузкой потоков.

Исходя из анализа результатов:

* Размер чанка 1000 - 5000 дает оптимальный баланс между распределением нагрузки и накладными расходами на управление задачами.
* Использование 16 потоков позволяет максимально раскрыть потенциал ExecutorService, но эффективность начинает снижаться из-за накладных расходов.
* ExecutorService обгоняет Модель делегирования 2 везде, но особенно заметно на больших размерах задач.

# **Задание 2**

Выполнить Exercise 12.3 [https://math.hws.edu/javanotes/c12/exercises.html]. Выполнить Exercise 12.4 [https://math.hws.edu/javanotes/c12/exercises.html]. Провести вычислительные эксперименты. Сравнить реализации и результаты экспериментов.

## **Тексты многопоточных программ.**

**Exercise 12.3**

| package com.example;  import java.util.concurrent.ConcurrentLinkedQueue; import java.util.concurrent.LinkedBlockingQueue; import java.util.Scanner;  public class **CountDivisorsUsingThreadPool** {  private final static int MAX = 25\_000\_000;  private final static int CHUNK\_SIZE = 1000;   private static ConcurrentLinkedQueue<Task> pendingTasks;   private record **Task**(int start, int end) {  public void **compute**() {  int highestDivisorCount = 0;  int bestNumber = 0;  for (int i = start; i < end; i++) {  int divCount = calculateDivisorCount(i);  if (divCount > highestDivisorCount) {  highestDivisorCount = divCount;  bestNumber = i;  }  }  taskResults.add(new Result(highestDivisorCount, bestNumber));  }   private static int **calculateDivisorCount**(int number) {  int count = 0;  int limit = (int) Math.sqrt(number);  for (int i = 1; i <= limit; i++) {  if (number % i == 0) {  if (i \* i == number) {  count++;  } else {  count += 2;  }  }  }  return count;  }  }   private static LinkedBlockingQueue<Result> taskResults;   private record **Result**(int taskMax, int taskNumber) {  }   private static class **DivisorCounterThread** extends **Thread** {  public void **run**() {  while (true) {  Task nextTask = pendingTasks.poll();  if (nextTask == null) {  break;  }  nextTask.compute();  }  }  }   private static void **executeDivisorCounting**(int threadCount) {  System.out.println("\nВычисление делителей с использованием " + threadCount + " потоков...");  long startTime = System.currentTimeMillis();   taskResults = new LinkedBlockingQueue<>();  pendingTasks = new ConcurrentLinkedQueue<>();   DivisorCounterThread[] workers = new DivisorCounterThread[threadCount];  for (int i = 0; i < threadCount; i++) {  workers[i] = new DivisorCounterThread();  }   int numberOfTasks = (MAX + (CHUNK\_SIZE - 1)) / CHUNK\_SIZE;  for (int i = 0; i < numberOfTasks; i++) {  int taskStart = i \* CHUNK\_SIZE + 1;  int taskEnd = (i + 1) \* CHUNK\_SIZE;  if (taskEnd > MAX) {  taskEnd = MAX;  }   pendingTasks.add(new Task(taskStart, taskEnd));  }   for (int i = 0; i < threadCount; i++)  workers[i].start();   int globalMaxCount = 0;  int globalBestNumber = 0;  for (int i = 0; i < numberOfTasks; i++) {  try {  Result result = taskResults.take();  if (result.taskMax() > globalMaxCount) {  globalMaxCount = result.taskMax();  globalBestNumber = result.taskNumber();  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }   long elapsed = System.currentTimeMillis() - startTime;   System.out.println("\nНаибольшее количество делителей для чисел от 1 до " + MAX + " равно " + globalMaxCount);  System.out.println("Число с таким количеством делителей: " + globalBestNumber);  System.out.println("Общее время выполнения: " + (elapsed / 1000.0) + " секунд.\n");  }   public static void **main**(String[] args) {  Scanner scanner = new Scanner(System.in);  int threadCount = 0;  while (threadCount < 1 || threadCount > 16) {  System.out.println("Сколько потоков вы хотите использовать (1 до 16)?");  threadCount = scanner.nextInt();  if (threadCount < 1 || threadCount > 16) {  System.out.println("Пожалуйста, введите число от 1 до 16!");  }  }   executeDivisorCounting(threadCount);  }  } |
| --- |

**Exercise 12.4**

| **package com.example;  import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors; import java.util.concurrent.Callable; import java.util.concurrent.Future; import java.util.ArrayList; import java.util.Scanner;  public class CountDivisorsUsingExecutor {  private final static int MAX = 25\_000\_000;  private final static int TASK\_SIZE = 1000;   public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  int threadCount = 0;  while (threadCount < 1 || threadCount > 16) {  System.out.println("Сколько потоков вы хотите использовать (1 до 16)?");  threadCount = scanner.nextInt();  if (threadCount < 1 || threadCount > 16) {  System.out.println("Пожалуйста, введите число от 1 до 16!");  }  }   executeDivisorCounting(threadCount);  }   private static void executeDivisorCounting(int threadCount) {  System.out.println("\nПодсчет делителей с использованием " + threadCount + " потоков...");   long startTime = System.currentTimeMillis();  ExecutorService executor = Executors.newFixedThreadPool(threadCount);   ArrayList<Future<DivisorResult>> futureResults = new ArrayList<>();   int taskCount = (MAX + TASK\_SIZE - 1) / TASK\_SIZE;  for (int i = 0; i < taskCount; i++) {  int start = i \* TASK\_SIZE + 1;  int end = Math.min((i + 1) \* TASK\_SIZE, MAX);  Future<DivisorResult> taskFuture = executor.submit(new DivisorTask(start, end));  futureResults.add(taskFuture);  }   int maxDivisorCount = 0;  int intWithMaxDivisors = 0;  for (Future<DivisorResult> taskFuture : futureResults) {  try {  DivisorResult result = taskFuture.get();  if (result.maxDivisorCount > maxDivisorCount) {  maxDivisorCount = result.maxDivisorCount;  intWithMaxDivisors = result.numberWithMaxDivisors;  }  } catch (Exception e) {  System.out.println("Ошибка при обработке задачи: " + e.getMessage());  executor.shutdownNow();  return;  }  }   long elapsedTime = System.currentTimeMillis() - startTime;  System.out.println("\nНаибольшее количество делителей " + "для чисел от 1 до " + MAX + " равно "  + maxDivisorCount);  System.out.println("Число с таким количеством делителей: " + intWithMaxDivisors);  System.out.println("Общее затраченное время: " + (elapsedTime / 1000.0) + " секунд.\n");   executor.shutdown();  }   private record DivisorResult(int maxDivisorCount, int numberWithMaxDivisors) {  }   private record DivisorTask(int min, int max) implements Callable<DivisorResult> {  public DivisorResult call() {  int maxDivisors = 0;  int numberWithMax = 0;  for (int i = min; i <= max; i++) {  int divisors = countDivisors(i);  if (divisors > maxDivisors) {  maxDivisors = divisors;  numberWithMax = i;  }  }  return new DivisorResult(maxDivisors, numberWithMax);  }   private int countDivisors(int num) {  int count = 0;  int limit = (int) Math.sqrt(num);  for (int i = 1; i <= limit; i++) {  if (num % i == 0) {  count++;  if (i \* i != num) {  count++;  }  }  }  return count;  }  } }** |
| --- |

## **Таблицы с результатами экспериментов.**

**Exercise 12.3**

| Размерность задачи | последовательная программа, c | 2 потока | | | 4 потока | | | 16 потока | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность |
| 1 000 000 | 0,834 | 0,438 | 1,90 | 0,9520547945 | 0,233 | 3,579399142 | 0,8948497854 | 0,125 | 6,672 | 0,417 |
| 10 000 000 | 27,206 | 13,748 | 1,978906023 | 0,9894530113 | 7,174 | 3,792305548 | 0,948076387 | 3,532 | 7,702718007 | 0,4814198754 |
| 25 000 000 | 103,644 | 55,933 | 1,8530027 | 0,9265013498 | 28,974 | 3,577138124 | 0,894284531 | 13,322 | 7,779912926 | 0,4862445579 |

**Exercise 12.4**

| Размерность задачи | последовательная программа, c | 2 потока | | | 4 потока | | | 16 потока | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность |
| 1 000 000 | 0,853 | 0,442 | 1,929864253 | 0,9649321267 | 0,23 | 3,708695652 | 0,927173913 | 0,126 | 6,76984127 | 0,4231150794 |
| 10 000 000 | 28,343 | 13,164 | 2,153068976 | 1,076534488 | 7,026 | 4,03401651 | 1,008504128 | 3,372 | 8,40539739 | 0,5253373369 |
| 25 000 000 | 106,463 | 51,417 | 2,070579769 | 1,035289885 | 29,983 | 3,550778775 | 0,8876946937 | 13,782 | 7,724785953 | 0,482799122 |

## 

## 

## **Сравнительный анализ.**

Обе реализации делят задачу на множество небольших подзадач и выполняют их параллельно, но отличаются способом управления потоками:

1. **Реализация с использованием собственного пула потоков:**

Задачи помещаются в очередь, результаты – в блокирующую очередь. Потоки забирают задачи из очереди до её опустошения и сразу же помещают результат в очередь результатов.

1. **Реализация с использованием ExecutorService и Future:**

Для управления пулом потоков используется встроенный механизм ExecutorService. Каждая задача реализована как Callable, возвращающая объект-результат. При отправке задачи в исполнитель возвращается Future, через который впоследствии можно получить результат.

Этот подход обеспечивает более высокоуровневый и удобный способ управления потоками, снижая вероятность ошибок при синхронизации.

**Сравнение результатов экспериментов:**

* Обе программы показывают хорошие ускорения при использовании 2 и 4 потоков (скоростной прирост около 1.9–3.6×) с эффективностью, близкой к 90–95%.
* При 16 потоках ускорение заметно ниже (примерно 7.77×), эффективность падает до 45–50%. Это связано с дополнительными накладными расходами.

**Вывод:**

* **Функционально:** Оба подхода корректно решают задачу и дают практически идентичные результаты по времени выполнения, ускорению и эффективности. Разница между ними минимальна и может быть связана скорее с методами синхронизации и накладными расходами встроенных механизмов, нежели с качеством алгоритма.
* **Управление потоками:** ExecutorService предоставляет более удобный и безопасный способ управления пулом потоков, что снижает вероятность ошибок, связанных с синхронизацией и завершением потоков.
* **Масштабируемость:** Для небольшого количества потоков (2–4) оба решения демонстрируют почти линейное ускорение. При использовании большого количества потоков эффективность падает.

Таким образом, выбор между собственным пулом потоков и ExecutorService в основном сводится к удобству разработки и поддержке кода: ExecutorService является более высокоуровневым и предпочтительным решением, особенно если не требуется тонкая настройка управления потоками.