aboba

Содержание

Введение

В современных вычислительных системах всё большее значение приобретают параллельные и асинхронные методы обработки данных [1]. Использование многопоточности позволяет выполнять независимые задачи одновременно, повышая общую эффективность программы и снижая время отклика.

Java предоставляет мощные средства для организации параллельных вычислений, включая такие высокоуровневые компоненты, как `ExecutorService`, `Callable` и `Future` [2]. Данные инструменты позволяют запускать задачи в отдельных потоках, получать результаты их выполнения и контролировать процесс работы без необходимости вручную управлять потоками.

В данной работе демонстрируется пример вычислительной задачи, выполняемой асинхронно: генерация массива случайных чисел и проверка наличия заданного элемента. Подобный подход может применяться для выполнения тяжёлых вычислений, анализа данных, поиска решений в больших пространствах или обработки потоков информации, не блокируя основной поток программы.

1 JAVA код

1.1 Общее назначение

Программа демонстрирует применение многопоточности в Java для параллельной обработки двумерного массива (матрицы). Задача заключается в заполнении матрицы случайными значениями и применении функции косинуса ко всем элементам с разделением вычислений между несколькими потоками.

1.2 Основные параметры

— `rows = 5000` — количество строк матрицы;

— `cols = 5000` — количество столбцов матрицы;

— `numThreads = 100` — количество потоков, между которыми распределяется работа.

1.3 Описание метода `coolMatrix()`

Метод `coolMatrix(int rows, int cols, int numThreads)` выполняет три основные операции:

а) Инициализация матрицы случайными числами;

б) Параллельное вычисление значений $\cos(x)$ для каждого элемента;

в) Ожидание завершения всех потоков.

1.4 Шаг 1. Генерация матрицы

Создаётся матрица типа `double[rows][cols]` и заполняется случайными числами в диапазоне $[0, 10)$:

$$M_{ij} = \text{rand}() \times 10$$

где rand() возвращает псевдослучайное число с плавающей запятой в диапазоне $[0, 1)$.

1.5 Шаг 2. Разделение работы на потоки

Создаётся массив потоков:

```
Thread[] threads = new Thread[numThreads];
```

Количество строк, обрабатываемых каждым потоком:

$$\texttt{rowsPerThread} = \frac{\texttt{rows}}{\texttt{numThreads}}$$

Для каждого потока t:

$$\texttt{startRow} = t \cdot \texttt{rowsPerThread}, \quad \texttt{endRow} = \begin{cases} \texttt{rows}, t = \texttt{numThreads} - 1 \\ \texttt{startRow} + \texttt{rowsPerThread}, \text{иначе} \end{cases}$$

1.6 Шаг 3. Обработка данных

Каждый поток выполняет следующий фрагмент:

```
for (int i = 0; i < rows; i++) {
 for (int j = 0; j < cols; j++) {
  matrix[i][j] = Math.cos(matrix[i][j]);
 }
}
```

Примечание: в текущей реализации присутствует ошибка — цикл по `i` должен выполняться в диапазоне `[startRow, endRow)`, чтобы каждый поток обрабатывал только свой участок данных. Иначе все потоки одновременно вычисляют косинус для всех строк, что не даёт прироста производительности и создаёт избыточные вычисления. Корректная версия:

```
for (int i = startRow; i < endRow; i++) {
 for (int j = 0; j < cols; j++) {
  matrix[i][j] = Math.cos(matrix[i][j]);
 }
}
```

1.7 Шаг 4. Синхронизация потоков

После запуска всех потоков вызывается метод `join()` для ожидания их завершения:

$$\forall t \in [0, numThreads - 1] : \texttt{threads[t].join()}$$

Это гарантирует, что матрица будет полностью преобразована до перехода к следующему этапу программы.

1.8 Измерение времени выполнения

В методе `main()` измеряется время выполнения вычислений:

$$\texttt{duration} = \texttt{endTime} - \texttt{startTime}$$

Результат выводится в миллисекундах:

$$\texttt{milliseconds} = \frac{\texttt{duration}}{10^6}$$

и отображается командой:

```
System.out.printf("Время выполнения: %.3f мс%n", milliseconds);
```

1.9 Java код методов нашего класса

```
public static void coolMatrix(int rows, int cols, int numThread) {
 double [][] matrix = new double[rows][cols];
 Random random = new Random();

 for (int i = 0 ; i < rows; i ++) {
  for (int j =0; j < cols ;j++) {
   matrix[i][j] = random.nextDouble() * 10;
  }
 }

 Thread[] threads = new Thread[numThread];
```

```
 int rowsPerThread = rows / numThread;

 for (int t = 0; t < numThread; t ++) {
  final int startRow = t * rowsPerThread;
  final int endRow = (t == numThread - 1) ?
  rows: startRow + rowsPerThread;

  threads[t] = new Thread(() -> {
   for (int i = 0 ; i < rows; i ++) {
    for (int j = 0; j < cols; j++) {
     matrix[i][j] = Math.cos(matrix[i][j]);
    }
   }
  });
  threads[t].start();
 }

 for (Thread thread: threads) {
  try{
   thread.join();
  } catch (InterruptedException e){
   e.printStackTrace();
  }
 }

}

public static void main(String[] args) {
 int rows = 5000;
 int cols = 5000;

 int numThreads = 100;
 long startTime = System.nanoTime();
```

```
 coolMatrix(rows, cols, numThreads);

 long endTime = System.nanoTime();

 long duration = endTime - startTime;
 double milliseconds = duration / 1_000_000.0;

 System.out.printf("Время выполнения: %.3f мс%n", milliseconds);
}
```

1.10 Результат выполнения

На выходе программа печатает только время выполнения в миллисекундах, например:

```
 Время выполнения: 154.382 мс
```

1.11 Ключевые концепции

— **Параллельная обработка данных** — распределение вычислений по нескольким потокам для ускорения работы;

— **Разделение данных (Data Partitioning)** — разделение матрицы на участки по строкам между потоками;

— **Синхронизация потоков** с помощью метода `join()`;

— **Потокобезопасность** — отсутствует взаимодействие между потоками, так как каждый работает со своей областью памяти (при исправленной версии кода);

— **Функция** `Math.cos()` — пример вычислительной нагрузки для моделирования CPU-интенсивных задач.

Заключение

Программа иллюстрирует базовый принцип параллельной обработки матриц в Java. При правильной реализации распределения строк между потоками достигается линейное ускорение вычислений при увеличении числа потоков, до тех пор, пока затраты на управление потоками не начинают преобладать. Таким образом, пример демонстрирует основы оптимизации вычислений на многоядерных системах.

Список использованных источников

1. *Зубкова, В. В.* Анализ актуальности закона Мура / В. В. Зубкова // *Перспективы развития информационных технологий*. — 2014. — no. 21. — Дата обращения: 06.10.2025. URL: https://cyberleninka.ru/article/n/analiz-aktualnosti-zakona-mura.

2. *Сьерра, К.* Изучаем Java: [пер. с англ.] / К. Сьерра, Б. Бейтс. Мировой компьютерный бестселлер. — Эксмо, 2012. URL: https://books.google.ru/books?id=qEZ1kQEACAAJ.