Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

**Комп’ютерний практикум № 5**

з дисципліни «Технології паралельних обчислень»

**Тема:** «Застосування високорівневих засобів паралельного програмування для побудови алгоритмів імітації та дослідження їх ефективності»

|  |  |
| --- | --- |
| **Виконав:**  студент групи ІТ-04  Чапча Святослав  Дата здачі \_\_\_\_\_\_\_\_  Захищено з балом \_\_\_\_\_\_\_ | **Перевірила:**  ас. кафедри ІПІ  Дифучина О.Ю. |

Київ 2023

**Завдання:**

1. З використанням пулу потоків побудувати алгоритм імітації багатоканальної системи масового обслуговування з обмеженою чергою, відтворюючи функціонування кожного каналу обслуговування в окремій підзадачі. Результатом виконання алгоритму є розраховані значення середньої довжини черги та ймовірності відмови. **40 балів.**

Лістинг програми:

QueueCallable.java

import java.util.concurrent.Callable;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.TimeUnit;  
  
public class QueueCallable implements Callable<Analyst> {  
 private boolean isObserving;  
 private int numOfConsumers;  
  
 QueueCallable() {  
 this.isObserving = false;  
 this.numOfConsumers = 5;  
 }  
 QueueCallable(int queueCapacity) {  
 this.isObserving = false;  
 this.numOfConsumers = queueCapacity;  
 }  
 QueueCallable(boolean isObserving) {  
 this.isObserving = isObserving;  
 this.numOfConsumers = 5;  
 }  
 QueueCallable(boolean isObserving, int queueCapacity) {  
 this.isObserving = isObserving;  
 this.numOfConsumers = queueCapacity;  
 }  
  
 public Analyst call() {  
  
 Manager manager = new Manager();  
 ExecutorService executor = Executors.*newFixedThreadPool*(Runtime.*getRuntime*().availableProcessors());  
  
 for (int i = 0; i < numOfConsumers; i++) {  
 Consumer thread = new Consumer(manager);  
 executor.execute(thread);  
 }  
  
 if(isObserving) {  
 Observer observer = new Observer(manager);  
 executor.execute(observer);  
 }  
  
 Analyst analyst = new Analyst(manager);  
 executor.execute(analyst);  
  
 Producer producer = new Producer(manager);  
 executor.execute(producer);  
  
 executor.shutdown();  
  
 System.*out*.println("Queue in process...");  
  
 try {  
 boolean ok = executor.awaitTermination(30, TimeUnit.*SECONDS*);  
 } catch (InterruptedException e) {  
 System.*out*.println("Something Happened");  
 throw new RuntimeException();  
 }  
  
 return analyst;  
 }  
}

Manager.java

import java.util.ArrayDeque;  
import java.util.Queue;  
  
public class Manager {  
 private int queueSize = 100;  
 public int skipped;  
 public int completed;  
 private Queue<Integer> queue;  
 public boolean isBusy;  
  
 Manager() {  
 this.completed = 0;  
 this.skipped = 0;  
 this.isBusy = false;  
 this.queue = new ArrayDeque<>();  
 }  
  
 public synchronized void addItem(int item) {  
 if(this.queue.size() >= this.queueSize) {  
 this.skipped += 1;  
 return;  
 }  
  
 this.queue.add(item);  
 notifyAll();  
 }  
  
 public synchronized int getItem() {  
 while(this.queue.size() == 0) {  
 try {  
 wait();  
 } catch (InterruptedException ignored) {}  
 }  
 return this.queue.poll();  
 }  
  
 public synchronized void incCompleted() {  
 this.completed += 1;  
 }  
  
 public synchronized int getQueueSize() {  
 return this.queue.size();  
 }  
}

Producer.java

import java.util.Random;  
  
public class Producer extends Thread {  
 private Manager manager;  
  
 Producer(Manager manager) {  
 this.manager = manager;  
 }  
  
 @Override  
 public void run() {  
 Random random = new Random();  
 long startTime = System.*currentTimeMillis*();  
 long currentTime = 0;  
  
 while (currentTime < 10\_000) {  
 this.manager.addItem(random.nextInt(100));  
  
 try {  
 Thread.*sleep*(random.nextInt(15));  
 } catch (InterruptedException ignored) {}  
  
 currentTime = System.*currentTimeMillis*() - startTime;  
 }  
  
 this.manager.isBusy = true;  
 }  
}

Consumer.java

import java.util.Random;  
  
public class Consumer extends Thread {  
 private Manager manager;  
  
 Consumer(Manager manager) {  
 this.manager = manager;  
 }  
  
 @Override  
 public void run() {  
 Random random = new Random();  
  
 while(!manager.isBusy) {  
 int newItem = manager.getItem();  
 try {  
 Thread.*sleep*(random.nextInt(100));  
 } catch (InterruptedException ignored) {}  
  
 manager.incCompleted();  
 }  
 }  
}

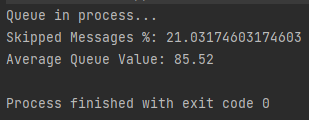
Analyst.java

public class Analyst extends Thread {  
 private Manager manager;  
 private int queueSize;  
 private int tasksNum;  
  
 Analyst(Manager manager) {  
 this.manager = manager;  
 this.queueSize = 0;  
 this.tasksNum = 0;  
 }  
  
 @Override  
 public void run() {  
 while(!this.manager.isBusy) {  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException ignored) {}  
  
 this.queueSize += this.manager.getQueueSize();  
 this.tasksNum += 1;  
 }  
 }  
  
 public double getAvgQueueValue() {  
 return (double) this.queueSize / this.tasksNum;  
 }  
 public double skippedChance(){  
 return (double) manager.skipped / (manager.skipped + manager.completed);  
 }  
}

Main.java

import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.\*;  
  
public class Main {  
 public static void main(String[] args) throws InterruptedException, ExecutionException{  
 task1();  
 }  
 public static void task1() {  
 QueueCallable task = new QueueCallable();  
 Analyst analyst = task.call();  
  
 System.*out*.println("Skipped Messages %: " + analyst.skippedChance()\*100);  
 System.*out*.println("Average Queue Value: " + analyst.getAvgQueueValue());  
 }

Результат виконання програми:



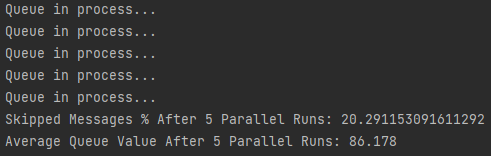
1. З використанням багатопоточної технології організувати паралельне виконання прогонів імітаційної моделі СМО для отримання статистично значимої оцінки середньої довжини черги та ймовірності відмови. **20 балів.**

Лістинг програми:

Main.java

public static void task2() throws InterruptedException, ExecutionException {  
 int poolsCount = 5;  
 ExecutorService executor = Executors.*newFixedThreadPool*(Runtime.*getRuntime*().availableProcessors());  
 List<Callable<Analyst>> tasks = new ArrayList<>();  
  
 for (int i = 0; i < poolsCount; i++) {  
 QueueCallable task = new QueueCallable();  
 tasks.add(task);  
 }  
  
 List<Future<Analyst>> result = executor.invokeAll(tasks);  
 executor.shutdown();  
  
 double totalMessages = 0;  
 double totalChance = 0;  
 for(Future<Analyst> taskResult : result) {  
 Analyst analyst = taskResult.get();  
 totalMessages += analyst.getAvgQueueValue();  
 totalChance += analyst.skippedChance();  
 }  
  
 System.*out*.println("Skipped Messages % After " + poolsCount + " Parallel Runs: " + (totalChance / result.size())\*100);  
 System.*out*.println("Average Queue Value After " + poolsCount + " Parallel Runs: " + totalMessages / result.size());  
}

Результат виконання програми:



1. Виводити результати імітаційного моделювання (стан моделі та чисельні значення вихідних змінних) в окремому потоці для динамічного відтворення імітації системи. **20 балів.**

Лістинг програми:

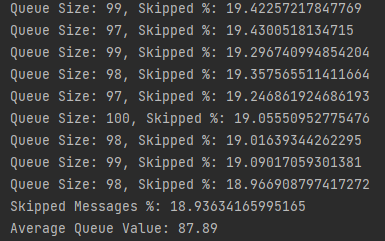
Observer.java

public class Observer extends Thread {  
 private Manager manager;  
 Observer(Manager manager) {  
 this.manager = manager;  
 }  
  
 @Override  
 public void run() {  
 while(!manager.isBusy) {  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException e) {}  
  
 System.*out*.println("Queue Size: " + this.manager.getQueueSize() + ", Skipped %: " + ((double) manager.skipped / (manager.skipped + manager.completed))\*100);  
 }  
 }  
}

Main.java

public static void task3() {  
 QueueCallable task = new QueueCallable(true);  
 Analyst analyst = task.call();  
  
 System.*out*.println("Skipped Messages %: " + analyst.skippedChance()\*100);  
 System.*out*.println("Average Queue Value: " + analyst.getAvgQueueValue());  
}

Результат виконання програми:



1. Побудувати теоретичні оцінки показників ефективності для одного з алгоритмів практичних завдань 2-5. **20 балів.**

Для виконання цього завдання я візьму алгоритм Фокса з другої практичної роботи. Оцінка показників ефективності буде проходити за наступними формулами:

, де - прискорення, - час виконання послідовної версії, а - час виконання паралельної версії на p процесорах.

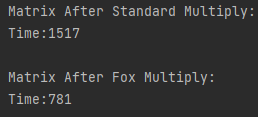
, де - ефективність, - прискорення, а p – кількість процесорів.

*, де* - вартість обчислень, p – кількість процесорів*,* а - час виконання паралельної версії на p процесорах.

Отже, перейдемо до замірів та обчислень.

1. 2 потоки

а) Розмір матриці – 1000

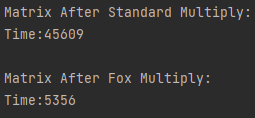


1.94

0.97

1.562

б) Розмір матриці – 2000

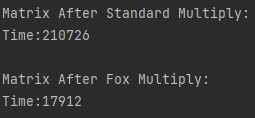


8.51

4.25

10.712

в) Розмір матриці – 3000



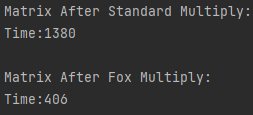
11.76

5.88

35.824

1. 4 потоки

а) Розмір матриці – 1000

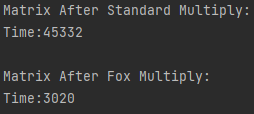


3.4

0.85

0.812

б) Розмір матриці – 2000

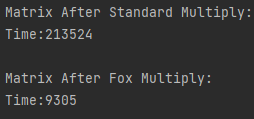


15.01

3.75

12.08

в) Розмір матриці - 3000

**

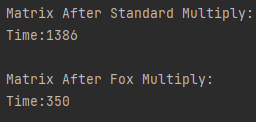
22.95

5.74

37.22

1. 6 потоків

а) Розмір матриці – 1000

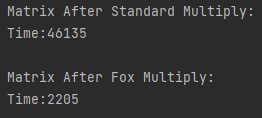


3.96

0.66

2.1

б) Розмір матриці – 2000

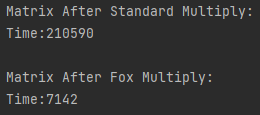


20.92

3.49

13.23

в) Розмір матриці - 3000

**

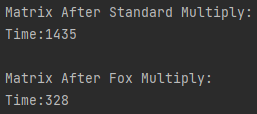
29.47

4.91

42.852

1. 8 потоків

а) Розмір матриці – 1000

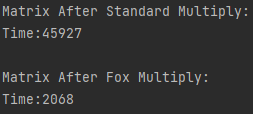


4.375

0.547

2.624

б) Розмір матриці – 2000

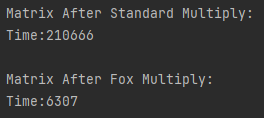


22.21

2.78

16.544

в) Розмір матриці - 3000

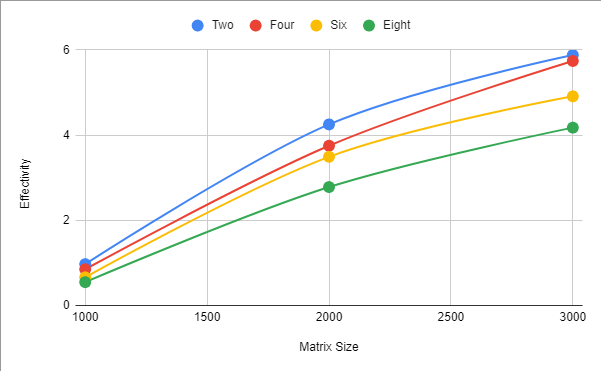
**

33.4

4.175

50.456

Побудуємо графік ефективності завдяки отриманим розрахункам:



Якщо ми використовуємо послідовний алгоритм для перемноження матриць, то кількість кроків, необхідних для цього, зростає в порядку O(n3). Однак, при певних умовах загальна кількість операцій, які потрібно виконати, може бути виражена у порядку n3/p. Це впливає на показники прискорення та ефективності алгоритму, які матимуть вигляд:

Загальний аналіз складності дає ідеальні показники ефективності паралельних обчислень.

Для підрахунку всіх аспектів необхідно враховувати інші характеристики.

Час паралельного виконання алгоритму, що відповідає розрахункам процесора:



На кожній ітерації один із процесорів у кожному рядку передає свій блок матриці A іншим процесорам у рядку:



Після множення матриці кожен процесор передає свій блок матриці Б до свого сусіда у вертикальному стовпці:



Тому, загальний час роботи паралельного алгоритма Фокса:

, де a – латентність, b – пропускна здатність мережі передачі, а w є розмір елемента матриці в байтах, параметр q визначає розмір процесорних ґрат і q = sqrt (p).

Програмний код до всіх завдань наведений в репозиторії:

<https://github.com/whitetark/multithreading/tree/main/lab05>

**Висновок**

У ході виконання моєї лабораторної роботи я використовував пул потоків для реалізації алгоритму імітації системи масового обслуговування. Крім того, я порівняв ефективність виконання алгоритму з використанням одного потоку з виконанням з використанням кількох потоків.