Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили

В. В. Старченко

Паралельне програмування

Методичні рекомендації
до виконання практичних та самостійних робіт

Випуск 327

Миколаїв – 2021

УДК 004.272.26
С 77

Рекомендовано до друку вченою радою Чорноморського національного університету імені Петра Могили (витяг з протоколу №11 від 27 серпня 2020 р.).

Рецензент:
Обрубов А. В., кандидат технічних наук, доцент кафедри «Суднових електроенергетичних систем» НУК ім. адм. Макарова.

С 77 **Старченко В. В.** Паралельне програмування : методичні рекомендації до виконання практичних та самостійних робіт : методичні рекомендації / В. В. Старченко. – Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2021. – 40 с. – (Методична серія; вип. 327).

Методичні рекомендації до виконання практичних та самостійних робіт призначені для студентів спеціальності 123 «Комп'ютерна інженерія», що вивчають курс «Паралельне програмування» на факультеті комп'ютерних наук Чорноморського національного університету ім. Петра Могили.

УДК 004.272.26

ISSN 1811-492X

ЗМІСТ

1. Модель обчислень у вигляді графа «операції-операнди»

Мета роботи: Опанувати методику побудови моделі обчислень у вигляді графа «операції-операнди».

1.1. Теоретичні відомості

1.1.1. Вступ

Модель обчислень у вигляді графа «операції-операнди» дуже часто використовується для опису наявних інформаційних залежностей у паралельних алгоритмах. Уявімо множину операцій, які виконуються в досліджуваному алгоритмі рішення обчислювальної задачі, і наявні між операціями інформаційні залежності у вигляді ациклічного орієнтованого графа $G=(V, E)$, де $V=\{1, ..., v\}$ – є множина вершин графа, що уявляє виконувані операції алгоритму, а E – множина дуг графа (при цьому дуга $e = (i, j)$ належить графу тільки, якщо операція j використовує результат виконання операції i). На рис. 2 наведено граф алгоритму обчислення площі прямокутника, заданого координатами двох кутів (Рис. 1) за формулою $s=(x_2 \cdot y_2 - x_2 \cdot y_1)+(x_1 \cdot y_1 - x_1 \cdot y_2)$.

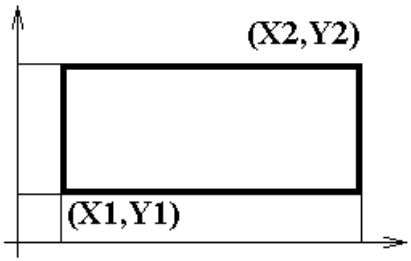

Рис. 1. Прямокутник, заданий координатами своїх кутів

1.1.2. Приклад паралельного алгоритму для обчислення площі прямокутника

Представлена тут програма ілюструє простий, але цікавий приклад використання обчислювальної моделі алгоритму у вигляді графа «операції-операнди» для побудови конвеєра команд. Розглянемо задачу обчислення площі прямокутника, заданого координатами своїх кутів (Рис. 1). Площу такого прямокутника можна обрахувати за наступною формулою:

$$s=(x_2-x_1)\cdot(y_2-y_1).$$

Розпаралелимо її:

$$s=(x_2-x_1)\cdot(y_2-y_1)=(x_2\cdot y_2-x_2\cdot y_1)+(x_1\cdot y_1-x_1\cdot y_2).$$

Побудуємо граф «операції-операнди» (рис. 2).

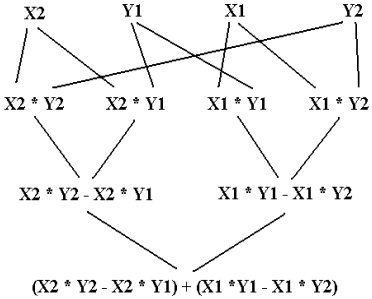

Рис. 2. Граф «операції-операнди»
для обчислення площі прямокутника

У верхньому рядочку графа розташовані операнди, значення яких подаються на вхід процесів, що виконують операцію множення. Результати цих операцій передаються процесам, які виконують операцію віднімання. На кінцевому етапі роботи результати віднімання збирає останній процес і виконує операцію їх складання. Для реалізації такого алгоритму створимо клас **Calculator**, який буде виконувати елементарні математичні дії. Для того, щоб цей клас можна було запускати як окремий процес, він повинен імплементувати інтерфейс **Runnable**. Математичні операції, які необхідно виконати визначаються константами **opSum**, **opSub**, **opMult** та **opDiv**. У об'єкт класу **Calculator** код операції передається за допомогою параметру **cop**. Початкові дані, проміжні та кінцеві результати зберігаються у лінійному масиві **data** класу **RectangleSquareCalc**. Цей масив є розподіленим ресурсом. Класу **Calculator** під час створення передаються лише індекси відповідних комірок у цьому масиві. А саме:

- op1 – індекс першого операнду;
- op2 – індекс другого операнду;
- res – індекс результату виконання операції.

Крім того, кожному примірнику класу **Calculator** передається його власне ім'я **name** та посилання на батьківський об'єкт-менеджер **RectangleSquareCalc**. Програмна реалізація класу **Calculator** на алгоритмічній мові **Java** наведена у лістингу 1.

```
import java.util.concurrent.*;

class Calculator implements Runnable {
  public final static int opSum = 1;
  public final static int opSub = 2;
  public final static int opMult = 3;
  public final static int opDiv = 4;
  int codeOp;
  int operator1;
  int operator2;
  int result;
  String threadName;
  RectangleSquareCalc manager;

  Calculator( String name, int cop, int op1, int op2, int res,
RectangleSquareCalc rsc) {
    threadName = name;
    codeOp = cop;
    operator1 = op1;
    operator2 = op2;
    result = res;
    manager = rsc;
    System.out.println( threadName + " - Created");
    }

  public void run() {
    System.out.println( threadName + " - Start of Work");
    while (!manager.stopFlag) {
      if (manager.flags[operator1] && manager.flags[operator2])
{
        switch(codeOp) {
          case opSum:
            manager.data[result] = manager.data[operator1] +
manager.data[operator2];
            System.out.println( threadName + ": Sum: " +
operator1 + " + " + operator2 + " ==> " + result);
            break;
          case opSub:
            manager.data[result] = manager.data[operator1] -
manager.data[operator2];
            System.out.println( threadName + ": Sub: " +
operator1 + " + " + operator2 + " ==> " + result);
            break;
          case opMult:
            manager.data[result] = manager.data[operator1] *
manager.data[operator2];
            System.out.println( threadName + ": Mult: " +
operator1 + " + " + operator2 + " ==> " + result);
            break;
```

```
          case opDiv:
            manager.data[result] = manager.data[operator1] /
manager.data[operator2];
            System.out.println( threadName + ": Div: " +
operator1 + " " + operator2 + " ==> " + result);
            break;
          default:
            break;
          }
        manager.flags[operator1] = false;
        manager.flags[operator2] = false;
        manager.flags[result] = true;
        }
      Thread.yield();
      }
    }
  }
```

Лістинг 1. Програмна реалізація класу Calculator

У процесі роботи кожен примірник класу **Calculator** аналізує наданий йому код операції, бере операнди із загального масиву, виконує над ними вказану операцію та кладе результат у загальний масив за його індексом.

Налагодження портфеля задач згідно з наведеним вище графом «операції-операнди» виконаємо за допомогою класу **RectangleSquare Calc**. Його програмна реалізація наведена у лістингу 2.

```
public class RectangleSquareCalc {
  double data[];
  boolean flags[];
  boolean stopFlag = false;

  RectangleSquareCalc( double x1, double y1, double x2, double
y2) {
    data = new double[15];
    flags = new boolean[15];
    int pos = 0;
    flags[pos] = true; data[pos++] = x2; flags[pos] = true;
data[pos++] = y2; flags[pos] = false; data[pos++] = 0.0;
    flags[pos] = true; data[pos++] = x2; flags[pos] = true;
data[pos++] = y1; flags[pos] = false; data[pos++] = 0.0;
    flags[pos] = true; data[pos++] = x1; flags[pos] = true;
data[pos++] = y2; flags[pos] = false; data[pos++] = 0.0;
    flags[pos] = true; data[pos++] = x1; flags[pos] = true;
data[pos++] = y1; flags[pos] = false; data[pos++] = 0.0;
    }


  public static void main( String argc[]) {
```

```
    System.out.println( "Main process started");
    RectangleSquareCalc rsc = new RectangleSquareCalc( 1.0, 2.0,
3.0, 4.0);

    ExecutorService execSvc = Executors.newFixedThreadPool( 7);

    execSvc.execute( new Calculator( "Mult_1",
Calculator.opMult, 0, 1, 2, rsc));
    execSvc.execute( new Calculator( "Mult_2",
Calculator.opMult, 3, 4, 5, rsc));
    execSvc.execute( new Calculator( "Mult_3",
Calculator.opMult, 6, 7, 8, rsc));
    execSvc.execute( new Calculator( "Mult_4",
Calculator.opMult, 9, 10, 11, rsc));
    execSvc.execute( new Calculator( "Sub_5", Calculator.opSub,
2, 5, 12, rsc));
    execSvc.execute( new Calculator( "Sub_6", Calculator.opSub,
11, 8, 13, rsc));
    execSvc.execute( new Calculator( "Sum_7", Calculator.opSum,
12, 13, 14, rsc));
    while (!rsc.flags[14]) {
      try {
        Thread.sleep( 2);
        }
      catch(InterruptedException e) {}
      }
    rsc.stopFlag = true;
    System.out.println();
    for (int i = 0; i < 4; ++i) {
      for (int j = 0; j < 3; ++j) {
        System.out.print( "( " + (3 * i + j) + " ): " +
rsc.data[3 * i + j] + ", ");
        }
      System.out.println();
      }
    System.out.println( "( " + 12 + " ): " + rsc.data[12] + ", ");
    System.out.println( "( " + 13 + " ): " + rsc.data[13] + ", ");
    System.out.println( "( " + 14 + " ): " + rsc.data[14] + ", ");
    execSvc.shutdown();
    System.out.println( "Main process ended");
    }
  }
```

Лістинг 2. Програмна реалізація класу **RectangleSquareCalc**

Примірник класу **RectangleSquareCalc** є об’єктом-менеджером, який керує усім процесом обчислення. На нього покладені наступні завдання:

– розміщення та зберігання початкових, проміжних та кінцевих результатів обчислень. Для цього призначено масив **data** подвійної точності;

– зберігання статусів початкових, проміжних та кінцевих результатів обчислень. Для цього призначено масив **flags** логічного типу;

– організація портфеля задач та призначення процесів для його обслуговування;

– створення та налагодження задач відповідно до графу «операції-операнди» та розміщення їх у портфелі задач;

– очікування результатів обрахунку та виведення їх на консоль;

– вивільнення ресурсів, зайнятих при організації портфеля задач.

Портфель задач, примірник класу **ExecutorService**, створимо виділивши для цього фіксований пул процесів з об'єкту **Executors**. У якості параметра, при цьому, вкажемо кількість необхідних процесів. Після створення портфеля задач, за допомогою його методу **execute**, додамо у нього робочі задачі. У нашому прикладі, такими задачами є примірники класу **Calculator**. Кожен такий примірник створюється оператором **new**, і у якості параметрів приймає свою власну назву, код виконуваної ним операції, індекси її двох операндів, індекс результату, та посилання на батьківський об'єкт. Згідно з наповненням портфеля задач, його робочі процеси виконують обчислення. Об'єкту-менеджеру **RectangleSquare Calc** залишається лише дочекатися результату, прийняти і вивести його на консоль.

Наведений приклад програмного коду належить до класу **програм, що керуються даними**. Тобто порядок обрахунків залежить виключно від структури графу «операції-операнди». При цьому, операції визначаються під час створення робочих задач, а операнди – розташовуються у лінійному загальнодоступному масиві. Відповідно до готовності даних, кожна комірка цього масиву може мати два стани: дані у комірці не готові, та дані готові до початку обчислення. Таким чином робоча задача може бути виконана тільки якщо дані обох операндів отримують стан готовності. Якщо, хоча б один з операндів не готовий до обрахунку, задача залишається у стані чекання. У наведеному прикладі, за стан готовності даних відповідає булевий масив **flags**. Початковий стан масивів **data** та **flags** наведено у таблиці 1.

Таблиця 1

Стан масивів data та flags перед початком обчислення

№	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
data	x2	y2	0	x2	y1	0	x1	y1	0	x1	y2	0	0	0	0
flags	T	T	F	T	T	F	T	T	F	T	T	F	F	F	F

1.2. Завдання

1. Розглянути, відкомпілювати та запустити на виконання наведений приклад.

2. З'ясувати принципи взаємодії та синхронізації процесів під час використання моделі обчислень у вигляді графа «операції-операнди».

3. З'ясувати особливості програмування та застосування моделі обчислень у вигляді графа «операції-операнди».

1.3. Індивідуальні варіанти завдань до захисту роботи

Побудувати граф «операції-операнди» та реалізувати його програмно для обрахунку такого виразу:

1. $y = \frac{(a+b+c+d)\frac{e+f}{g+h}}{\frac{\frac{i-j}{k\cdot l}}{m\cdot n+\frac{q}{r}}}$;
2. $y = \frac{\frac{a}{b}-(c+d)+\frac{e+f}{g\cdot k}}{i\cdot j\cdot\frac{k}{l}-(m\cdot n-p+q)}$;
3. $y = \frac{\left(\frac{a}{b}+c+d\right)\cdot(e\cdot f-(g+h))}{\left(i\cdot j-\frac{k}{l}\right)+(m+n)\cdot\frac{p}{q}}$;
4. $y = \frac{\frac{a+b}{c-d}+(e-f)\cdot(g-h)}{\left(i\cdot j+\frac{k}{l}\right)\cdot((m-n)+(p-q))}$;
5. $y = \frac{\frac{a\cdot b}{c+d}\cdot\left((e-f)+\frac{g}{h}\right)}{\frac{i+j}{k\cdot l}-\left(\frac{m}{n}-\frac{p}{q}\right)}$;
6. $y = \frac{\left(\frac{a}{b}+\frac{c}{d}\right)\cdot\left(\frac{e}{f}+\frac{g}{h}\right)}{\frac{i-j}{k-l}\cdot\frac{m\cdot n}{p-q}}$;
7. $y = \frac{\frac{a\cdot b}{c+d}+\frac{e-f}{g\cdot h}}{\left(\frac{i}{j}\frac{k}{l}\right)\cdot\frac{m-n}{p-q}}$;
8. $y = \frac{\frac{a+b}{c\cdot d}}{\frac{e\cdot f}{g-h}}\cdot\frac{\frac{i-j}{k-l}}{\frac{m}{n}-\frac{p}{q}}$;
9. $y = \frac{a\cdot b-c\cdot d-e\cdot f+\frac{g}{h}}{(i+j)\cdot\frac{k}{l}\left(\frac{m}{n}-\frac{p}{q}\right)}$;
10. $y = \frac{\frac{a}{b}+\frac{c}{d}}{(e-f)\cdot\frac{g}{h}}+\frac{\frac{i}{j}-k\cdot l}{\frac{m-n}{p\cdot q}}$;
11. $y = \frac{(a-b)\cdot\frac{c}{d}-\frac{e}{f}\frac{g}{h}}{\frac{i+j}{k\cdot l}\frac{m-n}{p+q}}$;

12. $y = \left(\frac{a+b}{c+d} - \frac{e}{f} \cdot (g+h)\right) \cdot \left((i \cdot j + k \cdot l) + \left(\frac{m}{n} - (p-q)\right)\right);$
13. $y = \left(((a+b) + c \cdot d) - (e+f) + (g+h)\right) - \left(i \cdot j + \frac{k}{l} - (m \cdot n - p \cdot q)\right);$
14. $y = \left(\left(a \cdot b - \frac{c}{d}\right) - (e \cdot f - g \cdot h)\right) + \left(i \cdot j + k \cdot l + \frac{m \cdot n}{p \cdot q}\right);$
15. $y = \left((a-b) \cdot c \cdot d\right) + \frac{e+f}{\frac{g}{h}} + (i \cdot j + k + l) \cdot \frac{m+n}{p-q};$
16. $y = \left(a \cdot b - \frac{c}{d} \cdot \frac{e+f}{g-h}\right) \cdot \left((i+j) \cdot (k+l) - \left(\frac{m}{n} + p \cdot q\right)\right);$
17. $y = \frac{(a+b) \cdot (c-d)}{(e+f) \cdot (g-h)} \cdot \left(i \cdot j - (k+l)\right) \cdot \frac{m+n}{p-q};$
18. $y = \frac{(a-b) \cdot \frac{c}{d}}{\frac{e}{f} - g \cdot h} + \frac{i-j}{k \cdot l} \cdot \left((m+n) - p \cdot q\right);$
19. $y = \frac{a+b-c \cdot d}{\frac{e}{f} + g \cdot h} \cdot \left(\frac{i+j}{k \cdot l} + m \cdot n \cdot \frac{p}{q}\right);$
20. $y = \frac{a \cdot b}{c-d} - \frac{e-f}{g \cdot h} + \left(\frac{i}{j} + k \cdot l\right) \cdot \frac{m \cdot n}{p \cdot q};$
21. $y = \frac{a \cdot b + c \cdot d}{e \cdot f - g \cdot h} \cdot \frac{\frac{i}{j} - \frac{k}{l}}{(m-n) \cdot (p+q)};$
22. $y = \frac{a-b}{c-d} \cdot \left(\frac{e}{f} + \frac{g}{h}\right) \cdot \left(\frac{i \cdot j}{k+l} + \frac{m}{p} \cdot \frac{n}{q}\right);$
23. $y = \frac{a \cdot b}{c+d} \cdot \frac{e \cdot f}{g-h} - \frac{i}{j} \cdot \frac{k}{l} \cdot \left(\frac{m}{n} + \frac{p}{q}\right);$
24. $y = \left(\frac{a+b}{c-d} + \frac{e \cdot f}{g \cdot h}\right) \cdot \left((i+j) \cdot \frac{k}{l} + \frac{m}{n} \cdot \frac{p}{q}\right);$
25. $y = \left(\frac{a+b}{c \cdot d} - \left(\frac{e}{f} - \frac{g}{h}\right)\right) \cdot \left(\frac{i-j}{k-l} + \frac{m}{n} \cdot \frac{p}{q}\right).$

2. Використання портфеля задач

Мета роботи: Опанувати методику створення розподілених програм з використанням портфеля задач.

2.1. Теоретичні відомості

Портфель задач – неупорядковане сховище задач, що чекають на виконання. Задачі кладуться у портфель, що розподілений між декількома робочими процесами. Кожен робочий процес виконує такий основний код:

```
while (true) {
    отримати задачу з портфеля;
    if (задач більше нема)
        break; // вихід з циклу while
    виконати задачу, можливо, породжуючи нові задачі;
    }
```

Лістинг 3. Алгоритм робочого процесу
для парадигми портфеля задач

Цей підхід можна використовувати для:

– реалізації рекурсивного паралелізму;

– вирішення ітеративних проблем з фіксованою кількістю незалежних задач.

Парадигма портфеля задач має наступні корисні властивості:

– вона дуже проста у використанні. Достатньо визначити представлення задачі, реалізувати портфель, запрограмувати виконання задачі і визначити умови завершення роботи алгоритму;

– програми, що використовують портфель задач є масштабованими у тому сенсі, що їх можна використовувати з будь-якою кількістю процесорів. Для цього достатньо просто змінити кількість робочих процесів;

– спрощується реалізація балансування навантаження. Якщо час виконання різних задач різний, то деякі з них будуть виконуватися довше інших. Але поки задач більше ніж робочих процесів (у 2 – 3 рази), загальні об'єми обчислень, що виконуються робочими процесорами, будуть приблизно однаковими.

2.2. Приклад використання портфеля задач за допомогою ExecutorService

У лістингу 4, за допомогою виклику **Executors.newFixedThread Pool** було створено пул на 5 процесів. Тепер, у разі появи великої кількості задач, вони будуть виконуватися вже наявними процесами. Тим самим, буде економитися час на створення нових процесів за рахунок більш ефективного використання вже наявних процесів з цього пулу.

```
import java.util.concurrent.*;
import java.util.Random;

class CallableImpl implements Callable <Integer> {
  private static final Random rand = new Random();
  int threadNumber = 0;

  public void setThreadNumber( int num) {
    threadNumber = num;
    }

  public Integer call() {
    System.out.println( "Callable task(" + threadNumber +
      ") begin");
    busy();
    System.out.println( "Callable task(" + threadNumber +
      ") end");
    return new Integer( threadNumber);
    }

  private void busy() {
    try {
      Thread.sleep( rand.nextInt( 500));
      }
    catch (InterruptedException e) {}
    }
  }

public class ExecutorServiceDemo {
  public static void main( String args[]) {
    CallableImpl callable[] = new CallableImpl[10];
    Future future[] = new Future[10];

    ExecutorService executor =
        Executors.newFixedThreadPool( 5);
    for (int i = 0; i < 10; ++i) {
      callable[i] = new CallableImpl();
      callable[i].setThreadNumber( i + 1);
      future[i] = executor.submit( callable[i]);
```

```
    }

  for (int i = 0; i < 10; ++i) {
    try {
      System.out.println( "Future value: " +
        future[i].get());
      }
    catch (Exception e) {
      e.printStackTrace();
      }
    }
  executor.shutdown();
  }
}
```

Лістинг 4. Приклад використання ExecutorService

Задача від процесу відрізняється тим, що може повертати результат обчислення до батьківського процесу. Для цього її треба імплементувати від інтерфейсу **Callable**. Цей інтерфейс вимагає, щоб усі дії, пов'язані з обчисленнями задачі виконувалися у публічному методі **call**. Такий метод, на відміну від методу **run** може повертати значення – результат обчислення.

Для того, щоб батьківський процес отримав результат обчислення від задачі, використовується синхронний об'єкт класу **Future**. Кожен примірник цього класу пов'язується з відповідною задачею за допомогою методу **submit** класу **ExecutorService** під час додавання цієї задачі до портфеля.

Після того, як задача додана до портфеля, її може виконати будь-який з його робочих процесів. Це може відбутися у будь-який момент часу. Тому виклик методу **get** з відповідного об'єкта типу **Future** змушує поточний процес очікувати результат виконання цієї задачі робочим процесом.

2.3. Завдання

1. Розглянути, відкомпілювати та запустити на виконання наведений приклад.
2. З'ясувати принципи взаємодії та синхронізації процесів під час використання портфеля задач.
3. З'ясувати особливості програмування та застосування портфеля задач.

2.4. Індивідуальні варіанти завдань до захисту роботи

1. Написати паралельну програму для генерації послідовності чисел, які менше 10000 та діляться на 3, 5, 7, 9. Використати два процеси та чотири задачі.

2. Обчислити визначений інтеграл для функції $y = x^2/4$ у діапазоні [2, 10]. Використати два процеси та чотири задачі.

3. Обчислити довжину шляху, пройденого матеріальною точкою за $t = 48$ секунд, яка рухається за таким законом:

$$\begin{cases} y = 16 \cdot sin(2 \cdot t), \\ x = 4 \cdot t. \end{cases}$$

Використати три процеси та дев'ять задач.

4. Гра «Хто більше». Використовуючи механізм синхронізації «портфель задач», створити два процеси та десять задач. Кожна задача генерує випадкове ціле число у діапазоні від 0 до 100 та закінчує свою роботу. Після вичерпання «портфеля задач» головна програма збирає згенеровані дані, та з'ясовує, яка із задач згенерувала найбільше число та виводить його у консоль разом із номером цієї задачі.

5. Маємо масив, заповнений випадковими цілими числами від 0 до 10000. Використовуючи механізм синхронізації «портфель задач», створити три процеси та десять задач для пошуку у цьому масиві заданого числа. Довжина масиву повинна бути кратна кількості задач. Забезпечити обробку ситуацій, коли шукане число не буде знайдене, або буде знайдено декілька таких чисел.

6. Маємо масив, заповнений цілими числами від 0 до 10000. Використовуючи механізм синхронізації «портфель задач», з'ясувати, чи відсортований він за зростанням. Створити чотири процеси та десять задач. Забезпечити вивід у консоль з головної програми повідомлення про те у якому з сегментів масиву порушується порядок (якщо таке є).

7. Маємо масив, заповнений випадковими цілими числами від 0 до 15000. Використовуючи механізм синхронізації «портфель задач», підрахувати кількість парних чисел у ньому. Створити три процеси та п'ятнадцять задач. Забезпечити вивід діагностики роботи програми у консоль.

8. Маємо кардіоїду вписану у квадрат. Обрахувати співвідношення площин кардіоїди та квадрата методом Монте-Карло. Використати чотири процеси та шістнадцять задач.

9. Маємо коло вписане у квадрат. Обрахувати співвідношення площин кола та квадрата методом Монте-Карло. Використати два процеси та вісім задач.

10. Маємо квадрат вписаний у коло. Обрахувати співвідношення площин кола та квадрата методом Монте-Карло. Використати чотири процеси та шістнадцять задач.

11. Маємо трикутник вписаний у коло. Обрахувати співвідношення площин кола та трикутника методом Монте-Карло. Використати три процеси та шістнадцять задач.

12. Маємо коло вписане у трикутник. Обрахувати співвідношення площин кола та трикутника методом Монте-Карло. Використати три процеси та шістнадцять задач.

13. Маємо зірку вписану у коло. Обрахувати співвідношення площин кола та зірки методом Монте-Карло. Використати три процеси та дев'ять задач.

14. Маємо сферу вписану у куб. Обрахувати співвідношення об'ємів сфери та куба методом Монте-Карло. Використати три процеси та дев'ять задач.

15. Маємо куб вписаний у сферу. Обрахувати співвідношення об'ємів сфери та куба методом Монте-Карло. Використати чотири процеси та шістнадцять задач.

16. Маємо сферу вписану у тетраедр[1]. Обрахувати співвідношення об'ємів сфери та тетраедра методом Монте-Карло. Використати три процеси та дев'ять задач.

17. Маємо тетраедр вписаний у сферу. Обрахувати співвідношення об'ємів сфери та тетраедра методом Монте-Карло. Використати чотири процеси та шістнадцять задач.

[1] Тетраедр описати як об'ємну фігуру, що обмежена такими площинами: $\sqrt{2}a + 4\sqrt{3}z \geq 0$, $4\sqrt{3}z \leq 3\sqrt{2}a + 8\sqrt{6}x$, $4(\sqrt{6}x - 3\sqrt{2}y + \sqrt{3}z) \leq 3\sqrt{2}a$, $4(\sqrt{6}x + 3\sqrt{2}y + 3z) \leq 3\sqrt{2}a$, де a — довжина сторони тетраедра.

3. Використання бар'єрної синхронізації

Мета роботи: Опанувати методику створення розподілених програм з використанням бар'єрної синхронізації.

3.1. Теоретичні відомості

Бар'єр – це засіб синхронізації, який використовується для того, щоб деяка кількість процесів очікувала одне одного у деякому місці програми, яке називається бар'єром або **точкою синхронізації**. Після того, як усі процеси досягли точки синхронізації, вони розблокуються і зможуть продовжувати виконання. На практиці бар'єри використовуються для збору результатів виконання деякої розпаралеленої задачі.

```
import java.util.concurrent.*;
import java.util.Random;

class Worker extends Thread {
  CyclicBarrier barrier;
  int workerNumber = 0;

  Worker( CyclicBarrier b, int n) {
    barrier = b;
    workerNumber = n;
    }

  public void run() {
    System.out.println( "Worker(" + workerNumber + "): start");
    try {
      barrier.await();
      System.out.println( "Worker(" + workerNumber + "): end");
      }
    catch (InterruptedException e) {
      e.printStackTrace();
      }
    catch (BrokenBarrierException e) {
      e.printStackTrace();
      }
    }
  }

public class CyclicBarrierDemo {

  public static void main( String args[]) {
    int workersNumber = 5;
    CyclicBarrier barrier = new CyclicBarrier( workersNumber,
new Runnable() {
```

```
      public void run() {
          System.out.println( "CyclicBarrier: run");
          }
        });

    for (int i = 0; i < workersNumber; i++) {
      new Worker( barrier, i).start();
      }
    }
  }
```

Лістинг 5. Приклад паралельного множення двох матриць з використанням бар'єрної синхронізації

Розглянемо приклад використання класу **CyclicBarrier**, який є реалізацією бар'єра. Він є складовою пакета **java.util.concurrent**. У якості такого прикладу можна розглянути задачу множення двох матриць (лістинг 5). Під час розпаралелювання цієї задачі кожному процесу буде доручено множення визначених рядків на визначені стовпці. У точці синхронізації отримані результати збираються від усіх процесів, і будується кінцева матриця.

Матриці *MB* і *MC* містять початкові дані. Матриця *MA*, після завершення обрахунків буде містити результат. Для того, щоб вони були доступні усім процесам, їх розміщено у публічному об'єкті, породженому від класу **MatrixCalcCyclicBarrier**, і продекларовано за допомогою модифікатора доступу **static**. Це означає, що матриці є статичними даними. Тобто вони повинні існувати протягом усього процесу обрахунку.

Процес налагодження програми складається з чотирьох етапів (див. функцію **main** лістингу 5). На першому етапі матриці *MA, MB* і *MC* розміщуються у статичній пам'яті. Їм виділяється об'єм пам'яті відповідно до їх розмірності. На другому – матриці *MB* і *MC* ініціалізуються одиницями, а матриці *MA* – нулями. На третьому – відбувається створення та налагодження бар'єра. Бар'єр створюється як примірник класу **CyclicBarrier**. При ініціалізації йому передається кількість робочих процесів (**workersNumber**) та анонімний клас типу **Runnable** з визначеним обробником події, яка відбудеться коли до цього бар'єра підійдуть усі його **workersNumber** робочих процесів. На четвертому етапі створюються та запускаються на виконання дочірні робочі процеси. Їх кількість повинна точно дорівнювати кількості процесів, вказаної під час створення бар'єра. Кожному процесу у якості параметрів передаються посилання на матриці даних та результату, початковий та кінцевий індекси частин матриць, які будуть оброблятися цим процесом, та посилання на примірник бар'єра.

Кожен дочірній процес є примірником класу **MatrixThread** і складається з конструктора та методу **run**. У конструкторі відбувається прийняття та переприсвоєння формальних параметрів. У методі **run** – процес обчислення. Кожен процес обраховує лише свою ділянку матриці *MA* та зупиняється перед бар'єром за допомогою методу **await**. Оскільки під час чекання інших процесів може виникнути немасковане переривання або аварійний прорив бар'єра, то метод **await** може викинути переривання **InterruptedException** та/або **BrokenBarrier Exception**. Їх треба перехопити та обробити. Для цього використовується конструкція **try – catch**.

3.2. Завдання

1. Розглянути, відкомпілювати та запустити на виконання наведений приклад.
2. З'ясувати принципи взаємодії та синхронізації процесів під час використання бар'єрної синхронізації.
3. З'ясувати особливості програмування та застосування бар'єрної синхронізації.

3.3. Індивідуальні варіанти завдань до захисту роботи

1. Написати паралельну програму для транспонування матриці розміром $4n \times 4n$ елементів n процесами. Для збору даних від робочих процесів та виводу кінцевого результату використати метод бар'єрної синхронізації.
2. Ввести загальну змінну з початковим значенням 0. Зробити так, щоб кожен процес після свого старту збільшував її на 1. Після зупинки усіх процесів перед бар'єром, він повинен вивести її значення на екран. Перед закінченням роботи кожен процес повинен зменшити її значення на 1 і вивести результат на екран.
3. Створити загальний масив розмірності N. Зробити так, щоб кожен процес після свого старту збільшував відповідний елемент цього масиву на одиницю. Після зупинки усіх процесів перед бар'єром він повинен вивести у консоль усі елементи цього масиву. Перед закінченням роботи кожен процес повинен зменшити на одиницю значення відповідної комірки масиву та вивести результат у консоль. Розмірність масиву задати як параметр у командному рядку під час запуску програми.
4. Розробити паралельну програму з двома робочими процесами та двома бар'єрами. Головний процес створює два дочірніх робочих процеси A і B. Процес A генерує матрицю A та ініціалізує її одиницями. Процес B генерує матрицю B та ініціалізує її двійками.

Після чого обидва процеси зупиняються біля першого бар'єра. Перший бар'єр виводить значення обох матриць у консоль. Після закінчення виводу процеси *A* і *B* обраховують матрицю *C*, яка є поелементною сумою матриць *A* і *B* (тобто $C = A + B$). Причому процес *A* обраховує першу половину матриці *C*, а процес *B* – другу. Після закінчення обрахунку обидва процеси зупиняються біля другого бар'єра, який виводить значення матриці *C* у консоль. Після цього процеси припиняють роботу. Розмірність матриць задати як параметр у командному рядку під час запуску програми.

5. Розробити паралельну програму з двома робочими процесами та двома бар'єрами. Головний процес створює два дочірніх робочих процеси *A* і *B*. Перший процес генерує матрицю відношення R_1: $x \geq y$ (більше чи дорівнює). Другий генерує матрицю відношення R_2: $x \leq y$ (менше чи дорівнює). Після чого обидва процеси зупиняються біля першого бар'єра. Перший бар'єр виводить значення обох матриць у консоль. Після закінчення виводу процеси *A* і *B* обраховують композицію цих відношень. Причому перший процес обраховує першу половину матриці відношення, а другий процес – другу. Після закінчення обрахунку обидва процеси зупиняються перед другим бар'єром. Другий бар'єр виводить значення отриманої матриці у консоль. Після цього процеси припиняють роботу. Розмірність матриць задати як параметр у командному рядку під час запуску програми.

6. Розробити паралельну програму з двома робочими процесами та двома бар'єрами. Головний процес створює два дочірніх робочих процеси *A* і *B*. Процес *A* генерує множину *A*. Процес *B* генерує множину *B*. Після чого обидва процеси зупиняються перед першим бар'єром. Перший бар'єр виводить значення обох матриць у консоль. Після закінчення виводу процеси *A* і *B* обраховують декартовій добуток цих множин. Причому процес *A* обраховує першу половину матриці декартового добутку, а процес *B* – другу. Після закінчення обрахунку обидва процеси зупиняються перед другим бар'єром. Другий бар'єр виводить значення матриці декартового добутку у консоль. Після цього процеси припиняють роботу. Потужність множин задати як параметр у командному рядку під час запуску програми.

7. Розробити паралельну програму з двома робочими процесами та двома бар'єрами. Головний процес створює два дочірніх робочих процеси *A* і *B*. Процес *A* генерує множину *A*. Процес *B* генерує множину *B*. Після чого обидва процеси зупиняються у першого бар'єра. Перший бар'єр виводить значення обох матриць у консоль.

Після закінчення виводу процеси *A* і *B* обраховують переріз цих множин. Причому процес *A* обраховує першу половину перерізу, а процес *B* – другу. Після закінчення обрахунку обидва процеси зупиняються перед другим бар'єром. Другий бар'єр виводить значення множини перерізу у консоль. Після цього процеси припиняють роботу. Потужність множин задати як параметр у командному рядку під час запуску програми.

8. Розробити паралельну програму з трьома робочими процесами та двома бар'єрами. Головний процес створює три дочірніх робочих процеси *A, B* і *C*. Процес *A* генерує матрицю *MA* та ініціалізує її одиницями. Процес *B* генерує матрицю *MB* та ініціалізує її двійками. Процес *C* генерує матрицю *MC* та ініціалізує її трійками. Після чого усі процеси зупиняються перед першим бар'єром. Перший бар'єр виводить значення усіх матриць у консоль. Після закінчення виводу процеси *A, B* і *C* обраховують матрицю *MD*, яка є результатом такого виразу: $MD=MA+MB \cdot MC$. Матриця *MC* – загальний ресурс, а матриці *MA* і *MB* розбиваються на три частини та розподіляються між робочими процесами. Після закінчення обрахунку усі робочі процеси зупиняються у другого бар'єра. Другий бар'єр виводить значення матриці *MD* у консоль. Після цього процеси припиняють роботу. Розмірність матриць задати як параметр у командному рядку під час запуску програми.

9. Розробити паралельну програму з двома робочими процесами та бар'єром. Головний процес створює два дочірніх робочих процеси *A* і *B*. Процес *A* генерує множину *A*. Процес *B* генерує множину *B*. Після чого обидва процеси зупиняються перед бар'єром. Бар'єр виконує об'єднання цих множин. Після цього процес *A* виводить у консоль першу половину об'єднаної множини, а процес *B* – другу. Потужність множин задати як параметр у командному рядку під час запуску програми.

10. Розробити паралельну програму з трьома робочими процесами та бар'єром. Головний процес створює три дочірніх робочих процеси *A, B* і *C*. Процес *A* генерує матрицю *MA* та ініціалізує її одиницями. Процес *B* генерує матрицю *MB* та ініціалізує її двійками. Процес *C* генерує матрицю *MC* та ініціалізує її трійками. Після чого усі процеси зупиняються перед першим бар'єром. Перший бар'єр збільшує кожен елемент матриць *MA*, *MB*, та *MC* на одиницю. Після закінчення роботи першого бар'єра процеси *A, B* і *C* здійснюють вивід матриць у відповідні дискові файли у форматі CSV програми Microsoft Excel. Тобто:

– процес *A* здійснює вивід матриці *MA* у файл MA.CSV;

– процес *B* здійснює вивід матриці *MB* у файл MB.CSV;
– процес *C* здійснює вивід матриці *MC* у файл MC.CSV

Після цього процеси припиняють роботу. Розмірність матриць задати як параметр у командному рядку під час запуску програми. Файли результату, які були утворені під час роботи програми (тобто MA.CSV, MB.CSV, MC.CSV) переглянути у програмі Microsoft Excel.

11. Розробити паралельну програму з двома робочими процесами та двома бар'єрами. Головний процес створює два дочірніх робочих процеси *A* і *B*. Процес *A* генерує матрицю *MA* та ініціалізує її одиницями. Процес *B* генерує матрицю *MB* та ініціалізує її двійками. Після чого обидва процеси зупиняються перед першим бар'єром. Перший бар'єр виконує введення значень матриці *MC* з дискового файлу (можна використати файл у форматі CSV). Після закінчення вводу процеси *A, B* і *C* обраховують матрицю *MD*, яка є результатом такого виразу $MD=MA+MB \cdot MC$. Матриця *MC* – загальний ресурс, а матриці *MA* і *MB* розбиваються на дві частини та розподіляються між робочими процесами. Після закінчення обрахунку усі робочі процеси зупиняються перед другим бар'єром. Другий бар'єр виводить значення матриці *MD* у консоль. Після цього процеси припиняють роботу. Розмірність матриць та назву файлу даних з матрицею *MC* задати як параметр у командному рядку під час запуску програми.

4. Використання обмінників

Мета роботи: Опанувати методику створення розподілених програм з використанням обмінників.

4.1. Теоретичні відомості

Обмінники – це засоби в синхронізації, які використовуються для **гарантованого** обміну інформацією між **двома** процесами. Обмінники є, також, **синхронними** засобами комунікації між двома процесами. Це означає, що процес-відправник даних буде знаходитися у стані чекання доти, доки процес-отримувач даних їх не отримає. Обмінник є дуже простим класом, який може пов'язати тільки два процеси, та за один сеанс зв'язку може передати лише один об'єкт даних.

У мові Java обмінники представлені класом **Exchanger**. Обмін даними відбувається під час виклику його методу **exchange**. При цьому, обмінник може працювати у таких режимах:

- режим передачі даних;
- режим приймання даних;
- режим одночасного приймання та передачі даних.

У режимі передачі даних у якості параметра методу **exchang**е треба передати дані у вигляді примірника **інтерфейсного клас**у (DataClass). Наприклад:

```
exchanger.exchange( new DataClass( parametersList ));
```

або так:

```
DataClass data = new DataClass( parametersList );
exchanger.exchange( data);
```

Значення, що повертає метод **exchange**, при цьому можна не приймати.

У режимі приймання даних, навпаки важливе саме значення, яке повертає метод exchange:

```
DataClass newData = exchanger.exchange( null);
```

У режимі одночасного приймання та передачі даних метод **exchange** можна застосувати наступним чином:

```
DataClass data = new DataClass( new DataClass( parametersList));
```

або

```
DataClass data = new DataClass( parametersList );
DataClass newData = exchanger.exchange( data);
```

Розглянемо приклад організації синхронного обміну даними між двома процесами **loop1** і **loop2**. Вони є примірниками класу **Loop** (лістинг 6). Особливість полягає у тому, що спочатку треба визначитися з даними, які підлягають обміну та форматом їх передачі. Для цього треба створити інтерфейсний клас. У наведеному лістингу це **DataClass**. Під час обміну буде передаватися числове значення (`int value;`) та рядок тексту (`String message;`). Примірнику інтерфейсного класу вони будуть передаватися під час його створення за допомогою конструктора (`new DataClass(value, textString);`). Після отримання інтерфейсного класу процес-приймач зможе отримати ці дані за допомогою методів **getValue** та **getMassage**.

```
import java.util.concurrent.*;

class DataClass {
  int value;
  String message;

  DataClass( int v, String s) {
    value = v; message = s;
    }
  int getValue() { return( value); }
  String getMassage() { return( message); }
  }

class Loop implements Runnable {
  int counter;
  String name;
  Exchanger<DataClass> exchanger;

  Loop( int startValue, String id, Exchanger<DataClass> ex) {
    counter = startValue;
    name = id; exchanger = ex;
    System.out.println( name + ": created");
    }

  public void run() {
    System.out.println( name + ": started");
    DataClass data = new DataClass( counter, name);
    for (int i = 0; i < 3; ++i) {
      try {
        DataClass newData = exchanger.exchange( data);
        counter += newData.getValue();
        System.out.println( name + ": from " +
          newData.getMassage() + ": data: " +
          newData.getValue() + ": state = " + counter);
        }
```

```
      catch (InterruptedException e) {
        System.err.println( e.toString());
        }
      }
    System.out.println( name + ": ended");
    }
  }

public class ExchangerDemo {
  public static void main( String args[]) {
    System.out.println( "Main process started");
    Exchanger<DataClass> exchanger = new Exchanger<DataClass>();
    Loop loop1 = new Loop( 1, "First", exchanger);
    Loop loop2 = new Loop( 2, "Second", exchanger);
    new Thread( loop1).start();
    new Thread( loop2).start();
    System.out.println( "Main process ended");
    }
  }
```

Лістинг 6. Приклад використання обмінника

Робота головного процесу (**ExchangerDemo**) складається з трьох головних етапів. На першому етапі створюється примірник обмінника **exchanger** для інтерфейсного класу **DataClass**. На другому – створюються два робочі процеси **loop1** і **loop2**, які є примірниками класу **Loop**. При ініціалізації у якості параметрів їм надається деяке числове значення (у лістингу 6 це номер процесу, який є одночасно і початковим значенням рахівника сеансів обміну даними) та текстовий рядок (у лістингу 6 це власна назва процесу), а також посилання на примірник обмінника, створений раніше. На третьому – процеси **loop1** і **loop2** запускаються на виконання.

Під час роботи програми відбуваються три сеанси обміну даними між робочими процесами **loop1** і **loop2**. Кожен з них створює свої власні примірники інтерфейсного класу **DataClass** для приймання та передачі даних та тричі викликає метод **exchange** для синхронного обміну даними з іншим процесом.

4.2. Завдання

1. Розглянути, відкомпілювати та запустити на виконання наведений приклад.
2. З'ясувати принципи взаємодії та синхронізації процесів під час використання обмінників.
3. З'ясувати особливості програмування та застосування обмінників.

4.3. Індивідуальні варіанти завдань до захисту роботи

1. Створити три процеси та організувати обмін даними між ними за топологією «Лінійка». Тобто перший процес передає дані другому, а другий – третьому.
2. Створити три процеси та організувати циклічний обмін даними між ними. Тобто перший процес повинен передавати дані до другого, другий – до третього, а третій – до першого.
3. Створити чотири процеси та організувати обмін даними між ними за топологією «Зірка». Тобто один (головний) процес повинен роздати дані іншим трьом (підлеглим) процесам.
4. Розробити паралельну програму – конвеєр даних для табулювання функції $y=1+1/(1+1/(1+1/(1+1/x)))$.
5. Розробити паралельну програму – конвеєр даних для табулювання функції $y=a\cdot(1+a\cdot(1+a\cdot(1+a\cdot(1+a))))$.
6. Розробити паралельну програму – конвеєр даних для табулювання функції $y=a\cdot x/(a\cdot x/(a\cdot x/(a\cdot x)))$.
7. Розробити паралельну програму – конвеєр даних для табулювання функції $y=a/(x+a/(x+a/(x+a/(x+a/x))))$.
8. Розробити паралельну програму – конвеєр даних для табулювання функції $y=a\cdot b/(x+a\cdot b/(x+a\cdot b/(x+a\cdot b/(x+a\cdot b/x))))$.
9. Розробити паралельну програму – конвеєр даних для табулювання функції $y=1+(1+(1+(1+x)/x)/x)/x$.
10. Розробити паралельну програму – конвеєр даних для табулювання функції $y=2\cdot(2\cdot(2\cdot x-1)\cdot x-1)\cdot x-1$.
11. Розробити паралельну програму – конвеєр даних для табулювання функції $y=a/(a/(a/(a/x-b)-b)-b)-b$.
12. Розробити паралельну програму – конвеєр даних для табулювання функції $y=a\cdot(b/(a\cdot(b/(a\cdot(b/x)+c))+c))+c$.
13. Написати паралельну програму для побайтового читання вхідного файлу *filename1* і запису його у стандартний вихідний потік, а також у файл *filename2*, тобто створення двох копій вхідних даних. Розпаралелити програму так, щоб вона використовувала три процеси:
   - читання із файлу *filename1*;
   - запису в стандартний вихідний потік;
   - запису у файл *filename2*.

   Для передачі даних між процесами використати обмінники.

5. *Часова синхронізація паралельних процесів*

Мета роботи: Опанувати методику створення розподілених програм з використанням часової синхронізації паралельних процесів.

5.1. Теоретичні відомості

Клас **CountDownLatch** пакету **java.util.concurrent** використовується для того, щоб один чи декілька процесів змогли дочекатися виконання певної кількості операцій у інших процесах. Цей клас працює за принципом таймера. Виконується ініціалізація його деяким початковим значенням і зворотній відлік. Під час виклику методу **await** цього класу деяким процесом, він переходить у стан чекання моменту досягнення рахівником таймера значення 0. На практиці цей клас зручно використовувати для координації моменту початку та закінчення певної кількості процесів. Це дозволяє зробити наступне:

- запускати декілька процесів в один і той самий момент часу;
- відстежувати момент закінчення роботи декількох процесів.

У наведеному прикладі (Лістинг 7) розглянемо початок виконання декількох процесів в один і той самий момент часу. Під час створення об'єкту **CountDownLatch**, його рахівник ініціалізується значенням 1. Усі процеси чекають моменту переходу цього рахівника у стан 0, після чого починають виконання свого коду.

```
import java.util.concurrent.*;

class LatchedThread extends Thread {
  private final CountDownLatch startLatch;
  private int procNumber;

  public LatchedThread( CountDownLatch s, int n) {
    startLatch = s;
    procNumber = n;
    System.out.println( "Thread(" + procNumber + "): created");
    }

  public void run() {
    System.out.println( "Thread(" + procNumber + "): started");
    try {
      startLatch.await();
      System.out.println( "Thread(" + procNumber + "): running");
      }
    catch(InterruptedException e) {
      System.err.println( e.toString());
      }
```

```
    System.out.println( "Thread(" + procNumber + "): stoped");
    }
  }

public class CountDownLatchDemo {
  public static void main( String args[]) {
    System.out.println( "Main process started");
    CountDownLatch startLatch = new CountDownLatch( 1);
    for (int i = 0; i < 4; ++i) {
      LatchedThread t = new LatchedThread(startLatch, i+1);
      t.start();
      }
    try {
      Thread.sleep( 200);
      }
    catch( InterruptedException e) {
      System.err.println( e.toString());
      }
    startLatch.countDown();
    System.out.println( "Main process stoped");
    }
  }
```

Лістинг 7. Приклад використання класу **CountDownLatch** для запуску декількох процесів в один і той самий момент часу

У наведеному прикладі (Лістинг 8) розглянемо процес відстеження моменту закінчення роботи декількох процесів. Під час створення об'єкту **CountDownLatch**, його рахівнику привласнюється значення кількості процесів. Після цього головний процес переводиться до стану чекання за допомогою методу **await**. Усі дочірні процеси перед закінченням виконання свого коду за допомогою методу **countDown** зменшують значення рахівника на одиницю. Після того, як значення рахівника стане дорівнювати 0 головний процес виходить зі стану чекання та продовжує виконання свого коду.

```
import java.util.concurrent.*;

class StopLatchedThread extends Thread {
  private final CountDownLatch stopLatch;
  private int procNumber;

  public StopLatchedThread( CountDownLatch s, int n) {
    stopLatch = s;
    procNumber = n;
    System.out.println( "Thread(" + procNumber + "): created");
    }
```

```
  public void run() {
    System.out.println( "Thread(" + procNumber + "): started");
    System.out.println( "Thread(" + procNumber + "): running");
    stopLatch.countDown();
    System.out.println( "Thread(" + procNumber + "): stoped");
    }
  }

public class StopLatchedThreadDemo {
  public static void main( String args[]) {
    CountDownLatch stopLatch = new CountDownLatch( 3);
    for (int i = 0; i < 3; ++i) {
      StopLatchedThread t = new StopLatchedThread( stopLatch, i + 1);
      t.start();
      }
    try {
      stopLatch.await();
      }
    catch( InterruptedException e) {
      System.err.println( e.toString());
      }
    System.out.println( "Main process stoped");
    }
  }
```

Лістинг 8. Приклад використання класу **CountDownLatch** для відстеження моменту закінчення роботи декількох процесів

5.2. Завдання

1. Розглянути, відкомпілювати та запустити на виконання наведені приклади.
2. З'ясувати принципи взаємодії та часової синхронізації процесів під час використання класу **CountDownLatch**.
3. З'ясувати особливості програмування та застосування класу **CountDownLatch**.

5.3. Індивідуальні варіанти завдань до захисту роботи

Примітка. Для синхронізації робочих процесів використати клас **CountDownLatch**. Кількість робочих процесів *n* ввести як параметр у командному рядку під час запуску програми.

1. Заповнити одновимірний масив значенням *True*, використовуючи *n* робочих процесів.
2. Підрахувати кількість елементів, які мають значення істина, в одновимірному булевому масиві, використовуючи *n* робочих процесів.
3. Підрахувати кількість елементів, які мають значення істина, в двовимірній булевій матриці, використовуючи *n* робочих процесів.

4. Підрахувати кількість елементів, які мають значення істина, в тривимірному булевому кубі даних, використовуючи *n* робочих процесів.

5. У цілочисельному масиві знайти середнє арифметичне, використовуючи *n* робочих процесів.

6. У цілочисельному масиві знайти медіану (середнє найменшого й найбільшого значень), використовуючи *n* робочих процесів.

7. Виконати інверсію елементів одновимірного масиву за допомогою *n* паралельних процесів.

8. Підрахувати суму елементів цілочисельної матриці за допомогою *n* паралельних процесів.

9. У двовимірному масиві знайти задане число за допомогою *n* паралельних процесів.

10. Маємо одновимірний масив цілих чисел. За допомогою *n* паралельних процесів здійснити у ньому такі зміни: 3 → 6, 8 → 12, 10 → 5. Вивести на екран початковий масив, модифікований масив і загальну кількість змін. Результат подати у вигляді таблиці.

11. За допомогою *n* паралельних процесів у послідовності цифр знайти номер першого елементу, який менше заданого числа.

12. Маємо масив натуральних чисел. За допомогою *n* паралельних процесів перевірити, чи не складається він з нулів.

13. Маємо послідовність упорядкованих за зростанням натуральних чисел. За допомогою *n* паралельних процесів визначити, чи можна вставити у неї елемент.

14. За допомогою *n* паралельних процесів у послідовності цифр знайти номер останнього негативного числа.

15. Маємо рядок якій складається зі слів (послідовність символів, які обмежені проміжками). За допомогою *n* паралельних процесів знайти усі слова, у яких літера «а» входить не менше двох разів.

16. За допомогою *n* паралельних процесів перевірити, чи виконується у зазначеному рядку баланс дужок.

17. Маємо рядок, за допомогою *n* паралельних процесів знайти найбільшу кількість цифр, які йдуть у ньому підряд.

18. За допомогою *n* паралельних процесів знайти найдовшу групу цифр у рядку. Якщо таку довжину мають декілька груп, то взяти першу за чергою.

19. Маємо рядок, який складається зі слів (послідовність символів, які обмежені проміжками). За допомогою *n* паралельних процесів у словах, які закінчуються на «ing» змінити закінчення на «ed» і розташувати їх у кінці рядка у черзі, яка зворотня їх появі у вхідному рядку.

20. Маємо рядок, який складається зі слів (послідовність символів, які обмежені проміжками). За допомогою *n* паралельних процесів знайти слово у якому кількість літер «a» та «b» максимальна.

21. Маємо рядок, який складається зі слів (послідовність символів, які обмежені проміжками). За допомогою *n* паралельних процесів знайти слово, яке містить найбільшу кількість голосних літер (на початку та в кінці рядка проміжків немає).

22. Маємо рядок. За допомогою *n* паралельних процесів видалити кожен символ, який зустрічається підряд більш, ніж один раз.

23. Маємо рядок у якому є однакові символи, що йдуть одне за одним. За допомогою *n* паралельних процесів визначити, скільки разів зустрічається таке поєднання?

24. За допомогою *n* паралельних процесів знайти суму цілих позитивних чисел кратних 3, що більше A і менше B ($A < B$).

25. За допомогою *n* паралельних процесів знайти суму цілих позитивних непарних чисел, менше 1000.

26. За допомогою *n* паралельних процесів знайти суму залишків від ділення на 5 усіх цілих позитивних чисел менше A ($A >> 5$).

27. Написати паралельну програму для транспонування матриці розміром $4n \times 4n$ елементів *n* процесами.

28. За допомогою *n* паралельних процесів обчислити периметр еліпсу заданого параметрично:

$$\begin{cases} X = A \cdot cos(T), \\ Y = A \cdot sin(T). \end{cases}$$

З кроком T = 1°C замінюючи проміжки кривої відрізками прямої.

6. Самостійна робота

Тема роботи: Архітектура багатопотокових додатків.

Мета роботи: Опанувати методику створення багатозадачних застосунків.

6.1. Теоретичні відомості

Розпаралелювання алгоритмів виконують для зменшення часу обрахунків. При цьому виникає питання, наскільки ефективно працює отриманий паралельний алгоритм.

Для оцінки ефективності паралельного алгоритму використовують так званий **коефіцієнт прискорення** (*K*). Коефіцієнт прискорення визначається як відношення часу виконання обрахунку за послідовним алгоритмом T_1 до часу виконання обрахунку за паралельним алгоритмом T_n. Тобто:

$$K = \frac{T_1}{T_n}.$$

Розрізняють **реальний** та **теоретичний** коефіцієнт прискорення. Реальний коефіцієнт прискорення можна визначити коли за паралельним алгоритмом алгоритмічною мовою побудована програма. Ця програма відлагоджена, виконана і отриманий результат. Тоді стає відомий і час обрахунку. Але на практиці так буває не завжди. Як правило, сучасні обчислювальні алгоритми є або дуже складними, або призначеними для обробки дуже великих об'ємів даних. Їх програмна реалізація вимагає багато робочого часу програмістів. Для їх виконання будуються так звані суперкомп'ютери. Усе це вимагає значного фінансування. Для оцінки обсягів цього фінансування і застосовують теоретичний коефіцієнт прискорення. Теоретичний коефіцієнт прискорення визначається як відношення кількості елементарних інструкцій у послідовному алгоритмі до кількості інструкцій у паралельному алгоритмі поділеної на кількість процесів, що їх виконують. Тобто:

$$K = \frac{S+P}{S+\frac{P}{n}}.$$

де S – кількість інструкцій у частині алгоритму, яка не може бути розпаралелена, P – кількість інструкцій у частині алгоритму, яка може бути розпаралелена та передана для виконання n робочим процесам (або процесорам).

При цьому вважається, що для виконання кожної інструкції потрібен однаковий час. Це значно зменшує точність обрахунку

теоретичного коефіцієнта прискорення. Тому його використовують лише для попередньої оцінки трудових та фінансових витрат.

Ще одна характеристика паралельного алгоритму – **ефективність використання** (E) робочих процесів (або процесорів). Вона визначається як відношення коефіцієнта прискорення K до кількості робочих процесів (або процесорів) n:

$$E = \frac{K}{n}.$$

6.2. Вимоги до звіту

Звіт з самостійної роботи повинен складатися з наступних частин:

1. Титульна сторінка.
2. Постановка задачі.
   1) схема обчислювальної системи;
   2) розрахункова формула;
   3) опис вхідних і вихідних даних;
   4) розподіл вхідних і вихідних даних за портами вводу/виводу.
3. Опис паралельного математичного алгоритму.
   1) кількість процесів (процесорів);
   2) перелік та опис початкових даних;
   3) перелік та опис результатів обчислень;
   4) перелік та опис загальних ресурсів;
   5) розрахунок розподілюваних ресурсів;
   6) розрахункові формули для кожного виду розрахунку (процесу);
   7) схема математичних розрахунків;
   8) діаграма паралельних процесів.
4. Структурна схема взаємодії процесів.
5. Лістинг паралельної програми.
6. Результати тестування паралельної програми.
   1) log – файл;
   2) початкові дані і результат обрахунку;
   3) час виконання (послідовної частини, паралельної частини, загальний).
7. Графік залежності часу виконання програми від кількості процесів, що її виконують.
8. Теоретичний обрахунок коефіцієнта прискорення.
9. Визначення реального коефіцієнта прискорення.
   1) загального (для усієї програми);
   2) для паралельної частини.
10. Висновок. Порівняння теоретичних коефіцієнтів прискорення з фактично отриманими під час тестування паралельної програми.

6.3. Індивідуальні завдання

1. **Паралельна черга** – це черга, у якій операції видалення і вставки можуть виконуватися паралельно. Нехай черга зберігається в масиві *queue[1:n]*. Дві змінні, *front* і *rear*, вказують відповідно на перший заповнений елемент, і на наступну порожню комірку. Операція видалення затримується доти, доки у комірці *queue[front]* не з'явиться елемент, потім видаляє його і збільшує значення *front* (за модулем *n*). Операція вставки зупиняється, поки немає порожньої комірки, потім новий елемент кладеться у комірку *queue[rear]*, і значення *rear* збільшується. Розробіть алгоритми і програму для вставки в чергу і видалення з неї з максимальним паралелізмом процесів. Зокрема, за винятком критичних точок, де відбувається звертання до розподілюваних змінних, вставки і видалення елементів повинні мати можливість виконуватися паралельно щодо відношення один до одного та внутрішнього паралелізму[2].

2. Розглянемо задачу визначення кількості слів у словнику, букви у яких не повторюються (унікальні). Прописні і малі літери не розрізняються[3]. Напишіть паралельну програму для вирішення цієї задачі. Використайте модель «портфель задач» і *w* робочих процесів. У кінці програми виведіть кількість слів, що складаються з унікальних літер, і найбільш довгі з них. Перед початком паралельних обчислень словник можна прочитати в розподілювані змінні.

3. Розглянемо задачу генерації усіх простих чисел до деякої межі *L* за допомогою моделі «портфель задач». Один зі способів рішення складається в імітації решета Ератосфена, у якому з масиву цілих чисел послідовно викреслюються числа, кратні простим: 2, 3, 5 і т. д. У цьому випадку портфель буде зберігати наступне просте число, яке треба використовувати. Цей метод легко програмується, але вимагає великих витрат пам'яті для збереження всього масиву з *L* цілих чисел. Напишіть паралельну програму реалізації цього алгоритму. Використовуйте *w* робочих процесів. У кінці програми виведіть 10 останніх знайдених простих чисел і час роботи обчислювальної частини програми для різних значень *L* і *w*.

4. Розглянемо задачу генерації усіх простих чисел до деякої межі *L* за допомогою моделі «портфель задач». Інший спосіб вирішення цієї задачі полягає у почерговій перевірці усіх непарних чисел. У цьому

[2] Вам знадобляться додаткові змінні. Можна припустити, що доступна інструкція «витягти і скласти».

[3] У більшості систем Unix є один або кілька словників, наприклад у файлі /usr/dict/words.

випадку «портфель» буде містити всі непарні числа[4]. Для кожного кандидата перевіряємо, чи просте це число. Якщо так, додаємо його у кінець списку відомих простих чисел. Цей список використовується для перевірки наступних кандидатів. Такий метод вимагає значно менше пам'яті, але виникає складна проблема синхронізації. Напишіть паралельну програму реалізації цього алгоритму. Використовуйте *w* робочих процесів. У кінці роботи програми виведіть 10 останніх знайдених простих чисел і час роботи обчислювальної частини програми для різних значень *L* і *w*.

5. Розглянемо послідовно зв'язаний список, елементи якого зв'язані у порядку зростання їхніх полів даних. Стандартний послідовний алгоритм вставки нових елементів у необхідне місце має лінійну оцінку часу роботи (у середньому повинна бути переглянута половина списку). Напишіть алгоритм, паралельний за даними, для вставки нового елемента у список за логарифмічний час.

6. Розглянемо два послідовно зв'язаних списки. Напишіть алгоритм, паралельний за даними, що ставить у відповідність елементи з однаковими номерами в послідовностях. Після завершення алгоритму ці елементи списків повинні вказувати один на одного. Якщо один список довший за іншого, то зайві елементи довшого списку повинні містити порожні вказівники. Побудуйте необхідні структури. Не змінюйте початкові списки. Замість цього збережіть відповіді у додаткових масивах.

7. Припустимо, що *P[1:n]* – масив процесів, а *b[1:n]* – розподілюваний булевий масив. Напишіть алгоритм і програму, паралельну за даними, для підрахунку кількості елементів *b[i]*, що мають значення «істина».

8. Припустимо, що є *N* процесів *P[1:n]*, і *P[1]* має деяке локальне значення *V*, яке потрібно передати іншим, зберігши у кожному елементі масиву *a[1:n]*. Очевидний послідовний алгоритм вимагає лінійного часу. Напишіть алгоритм, паралельний за даними, якому достатньо логарифмічного часу.

9. Один з можливих способів сортування *N* цілих чисел полягає у використанні обмінного сортування «парний – непарний» (або сортування перестановкою парних і непарних). Припустимо, що є *N* процесів *P[1:n]*, і *N* парне. Під час цього способу сортування кожен процес виконує серію циклів. На непарних циклах процеси *P[odd]* з непарними номерами обмінюються значеннями з *P[odd + 1]*, якщо значення не упорядковані. На парних циклах процеси *P[even]* з

[4] їх зберігати непотрібно, оскільки з поточного числа легко одержати наступне.

парними номерами обмінюються значеннями з *P[even + 1]*, якщо значення не упорядковані (на парних циклах процеси *P[1]* і *P[n]* нічого не роблять). Визначити, скільки циклів потрібно виконати у гіршому випадку для сортування *N* чисел. Напишіть алгоритм, паралельний за даними, для сортування цілочисельного масиву *A[1:n]* у порядку зростання.

10. Розробити паралельну програму-конвеєр для табулювання функції *y=1+1/(1+1/(1+1/(1+1/x)))*.

11. Розробити паралельну програму-конвеєр для обрахування детермінанту матриці розміром $N \times N$.

12. Генератори даних генерують вектори такої структури: (*колір, число*). Дані потрапляють до рахівника, який представляє набір комірок відповідного кольору. Кожне число додається до того, що вже є у комірці відповідного кольору. Дані будь-якого кольору можуть у будь-який момент часу бути згенерованими будь-яким генератором. Робота програми завершується, коли сума чисел будь-якого кольору перевищує заздалегідь встановлену межу. Написати паралельну програму, яка реалізує таку систему з 6 генераторами даних.

13. Розробити паралельну програму-конвеєр для вирішення системи лінійних рівнянь розмірності *N*.

14. Виріб складається з чотирьох складових частин «a», «b», «c», «d», які можуть надходити у цех у довільному порядку один за одним. За наявності усіх потрібних частин виріб збирається і відправляється у сховище. Скласти паралельну програму ведення роботи цеху та сховища.

15. Вироби складаються з деталей «a», «b», «c», «d»:

– виріб 1 з «a», «b», «c»;
– виріб 2 з «b», «c», «d».

Скласти паралельну програму комплектації виробів, яка б визначала кількість скомплектованих виробів. Вироби мають комплектуватися рівномірно, деталі надходять одна за одною у довільному порядку.

16. Розпаралелити задачу обчислення площі неправильного опуклого п’ятикутника, заданого координатами вершин, методом тріангуляції. Площу елементарних трикутників обраховувати за формулою Герона. Кількість процесів не обмежувати. Обов’язково навести граф «операції-операнди».

17. У процесі обробки зображень виникає наступна задача виділення області. Початкове зображення зберігається у масиві цілих чисел *image[1:n, 1:n]*. Значення кожного елемента – інтенсивність пікселя. Сусідами пікселя є чотири пікселя (ліворуч, праворуч, знизу і

зверху від нього). Два сусідніх належать до однієї області, якщо їхні значення рівні. Таким чином, область – це максимальна множина пікселів, що зв'язані (у сенсі транзитивно-рефлексивного замикання відносин сусідства) і мають однакові значення. Завдання полягає у тому, щоб знайти всі області і надати усім пікселям кожної області унікальну (для області) мітку. Точніше, нехай *label[1:n, 1:n]* – ще одна матриця, і початковим значенням елемента *label[i, j]* є *n * i + j*. Кінцеве значення елемента *label[i, j]* повинне бути максимальним з початкових міток області, якій належить піксель *[i, j]*. Опишіть сіткові обчислення, паралельні за даними, для визначення кінцевих значень елементів *label*. Обчислення повинні закінчуватися, якщо жоден з елементів *label* не змінюється. Напишіть програму, паралельну за даними, що реалізує цей алгоритм. Час її виконання повинен бути пропорційний *log(n)*.

18. У процесі обробки зображень виникає наступна задача виділення границь областей з однаковим кольором. Початкове зображення зберігається у масиві цілих чисел *image[1:n, 1:n]*. Значення кожного елемента – колір пікселя. Сусідами пікселя є чотири пікселя (ліворуч, праворуч, знизу і зверху від нього). Два сусідніх належать до однієї області, якщо їхні значення рівні. Таким чином, область – це максимальна множина пікселів, що зв'язані (у сенсі транзитивно-рефлексивного замикання відносин сусідства) і мають однакові значення. Завдання полягає у тому, щоб знайти всі границі між такими областями, підрахувати загальну кількість таких областей і загальну довжину границь (у знайдених пікселях). Напишіть програму, паралельну за даними, що реалізує цей алгоритм. Час її виконання повинен бути пропорційний *log(n)*.

19. Маємо два концентричних кола радіусом R_1 і R_2. Причому $R_1 < R_2$. Обчислити співвідношення площ цих кіл методом Монте-Карло. Тобто написати паралельну програму, яка підраховує наскільки площа меншого кола менше ніж площа більшого. Дослідити залежність точності результату від кількості випробувань. Побудувати відповідний графік (можна за допомогою MS Excel).

20. Маємо зірку, яка вписана у коло. Обчислити співвідношення площ зірки і кола методом Монте-Карло. Тобто написати паралельну програму, яка підраховує наскільки площа зірки менше площі кола. Дослідити залежність точності результату від кількості випробувань. Побудувати відповідний графік (можна за допомогою MS Excel).

ДЛЯ НОТАТОК

ДЛЯ НОТАТОК

Навчальне видання

В'ячеслав Володимирович
Старченко

Паралельне програмування

Методичні рекомендації
до виконання практичних та самостійних робіт

Випуск 327

Редактор *Р. Грубкіна.*
Технічний редактор *О. Петроченко*. Комп'ютерна верстка *Н. Кардаш*
Друк *С. Волинець.* Фальцювально-палітурні роботи *О. Мішалкіна.*

Підп. до друку 14.05.2021
Формат 60x84$^{1}/_{16}$. Папір офсет.
Гарнітура «Times New Roman». Друк ризограф.
Ум. друк. арк. 2,33. Обл.-вид. арк. 1,55.
Тираж 5 пр. Зам. № 6111.

54003, м. Миколаїв, вул. 68 Десантників, 10.
Тел.: 8 (0512) 50–03–32, 8 (0512) 76–55–81, e-mail: rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 6124 від 05.04.2018.