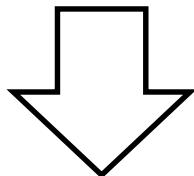


Поняття паралельних, одночасних (concurrent) та розподілених обчислень

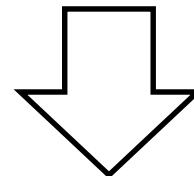
Інна Вячеславівна Стеценко
професор кафедри АСОІУ НТУУ «КПІ ім. Ігоря Сікорського»,
д.т.н., доцент

Структура курсу

Паралельні та розподілені обчислення



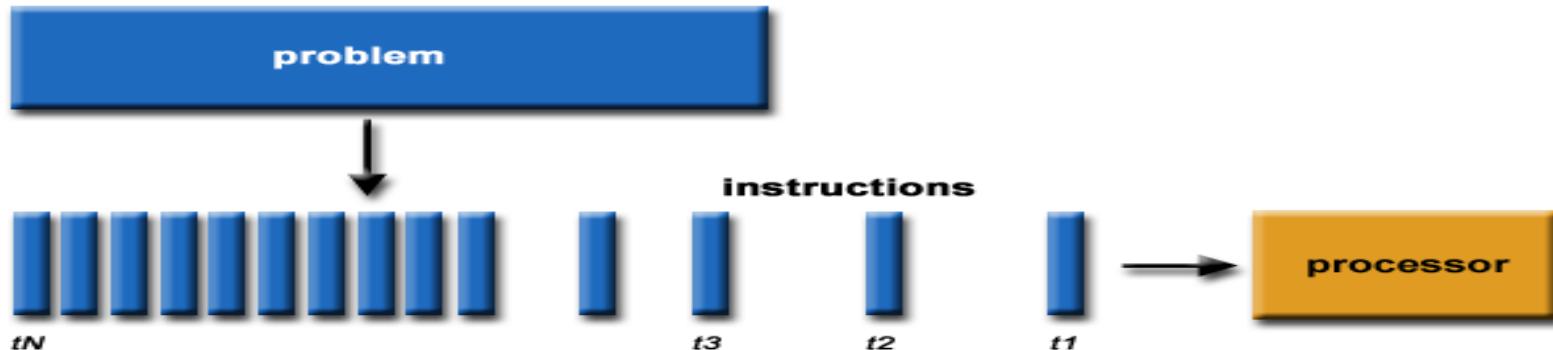
Паралельне
програмування



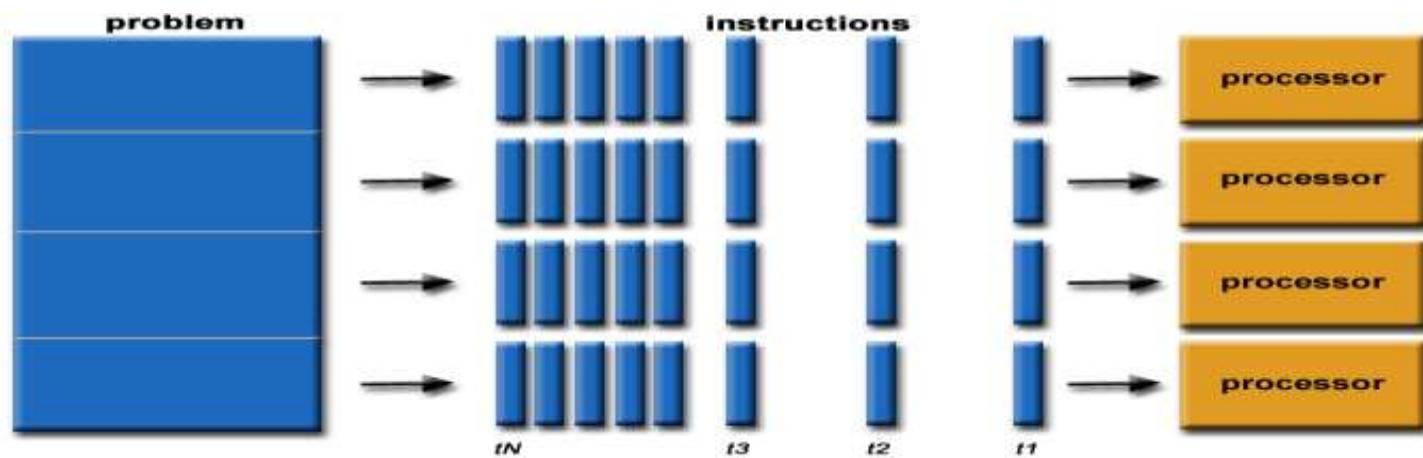
Паралельні обчислення
в розподілених системах

Послідовні та паралельні обчислення

Послідовні



Паралельні



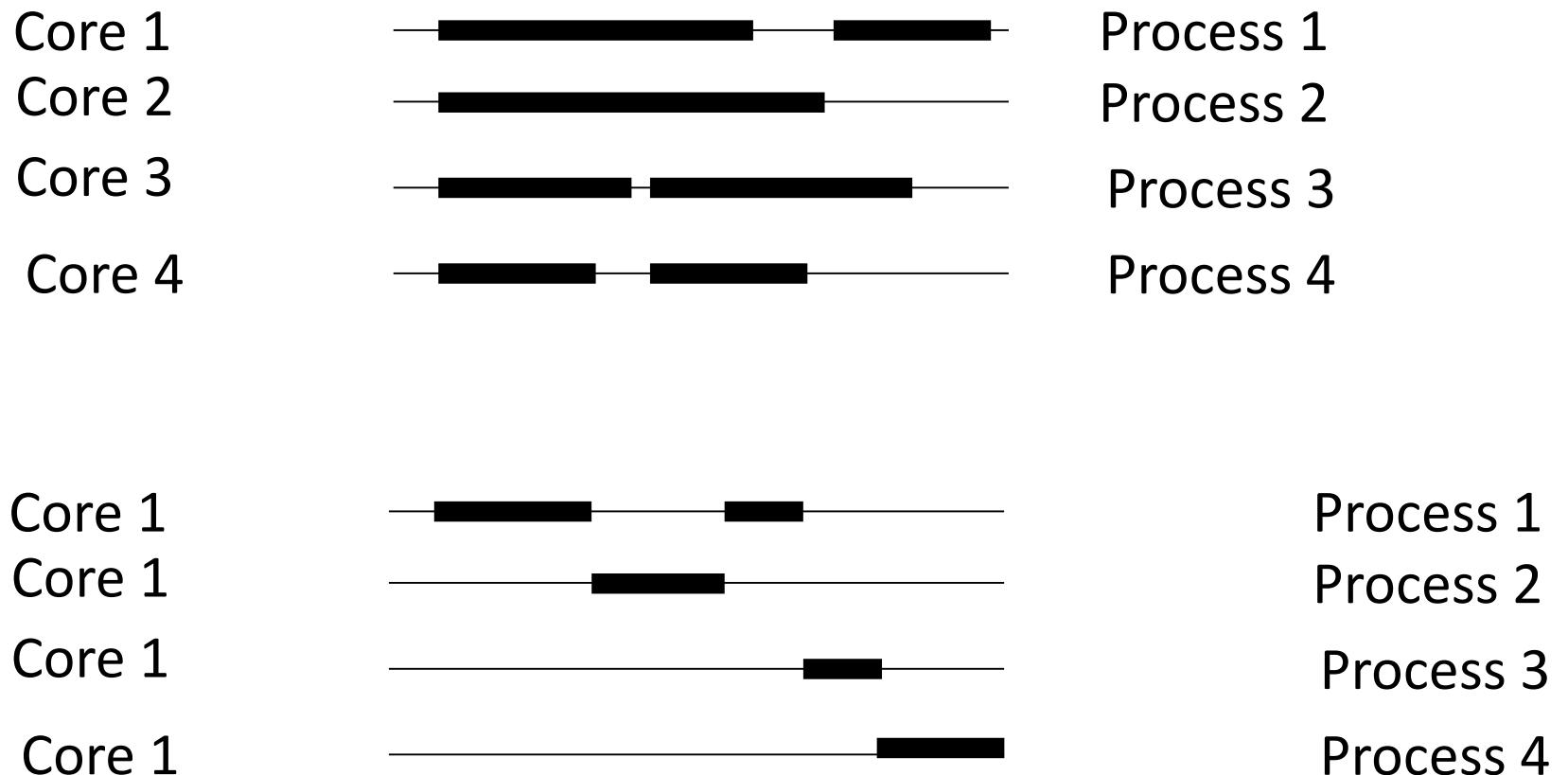
Визначення паралельних обчислень

Паралельні обчислення = одночасне використання кількох комп'ютерних ресурсів для розв'язання однієї обчислювальної задачі

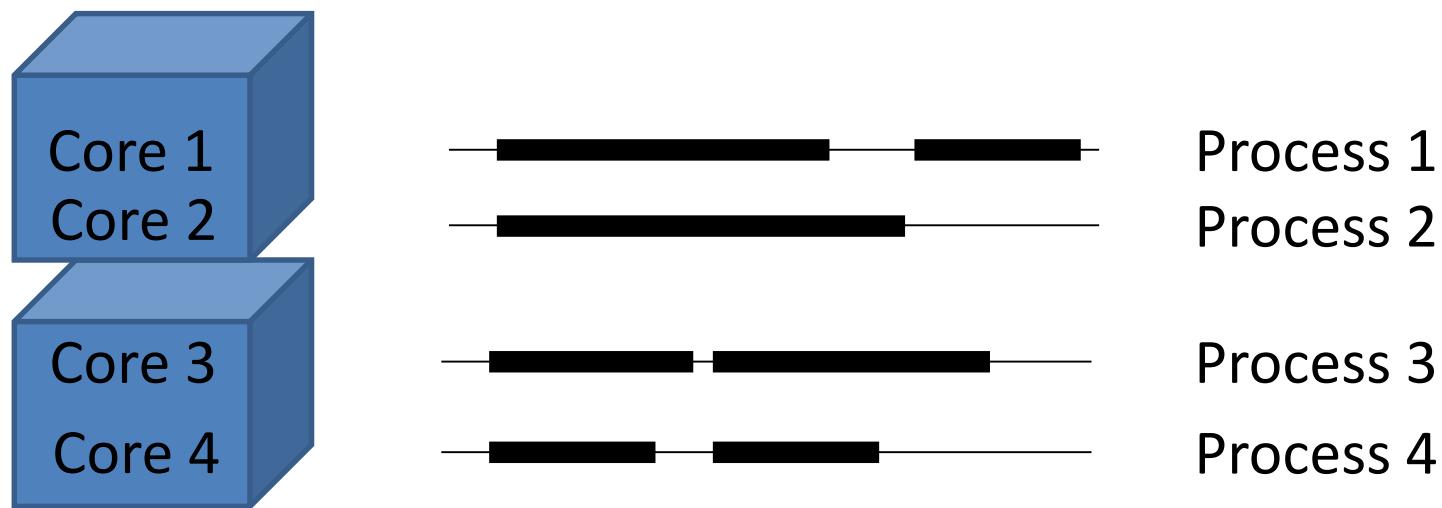
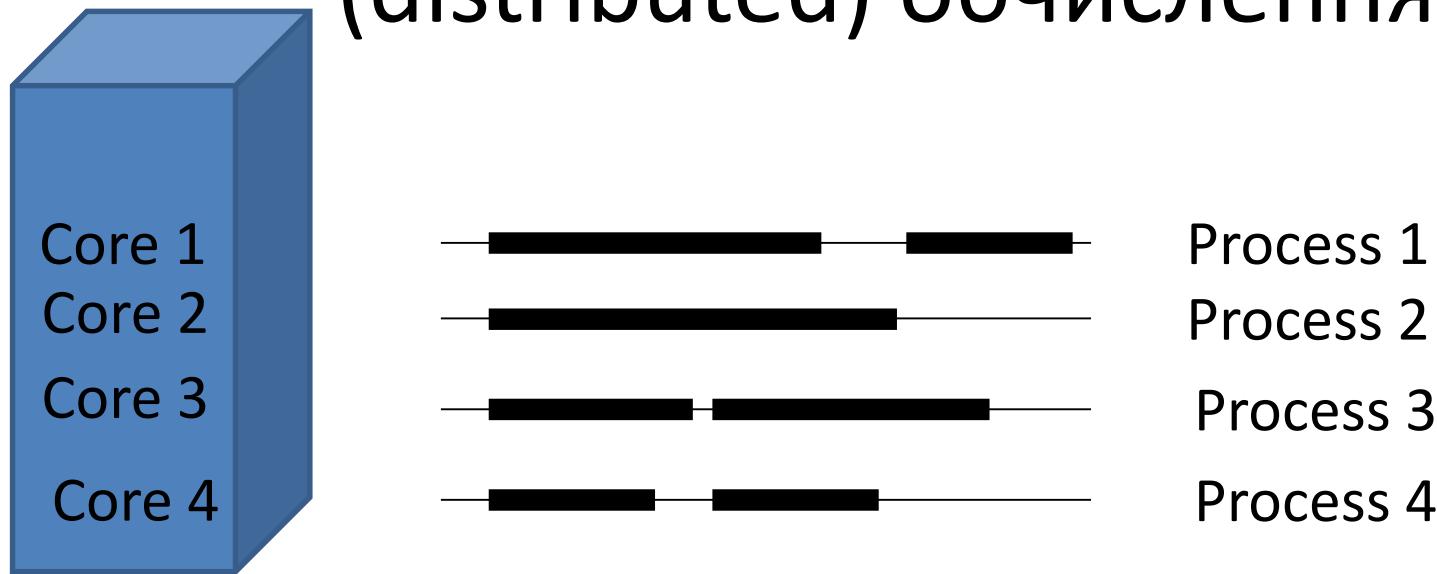
Властивості задачі, яка може обчислюватись паралельно:

- може бути розбита на частини, які можуть обчислюватись незалежно одна від одної у будь які моменти часу;
- одночасне виконання частин задачі зменшує загальний час її обчислення.

Паралельні (parallel) та одночасні (concurrent) обчислення



Паралельні (parallel) та розподілені (distributed) обчислення

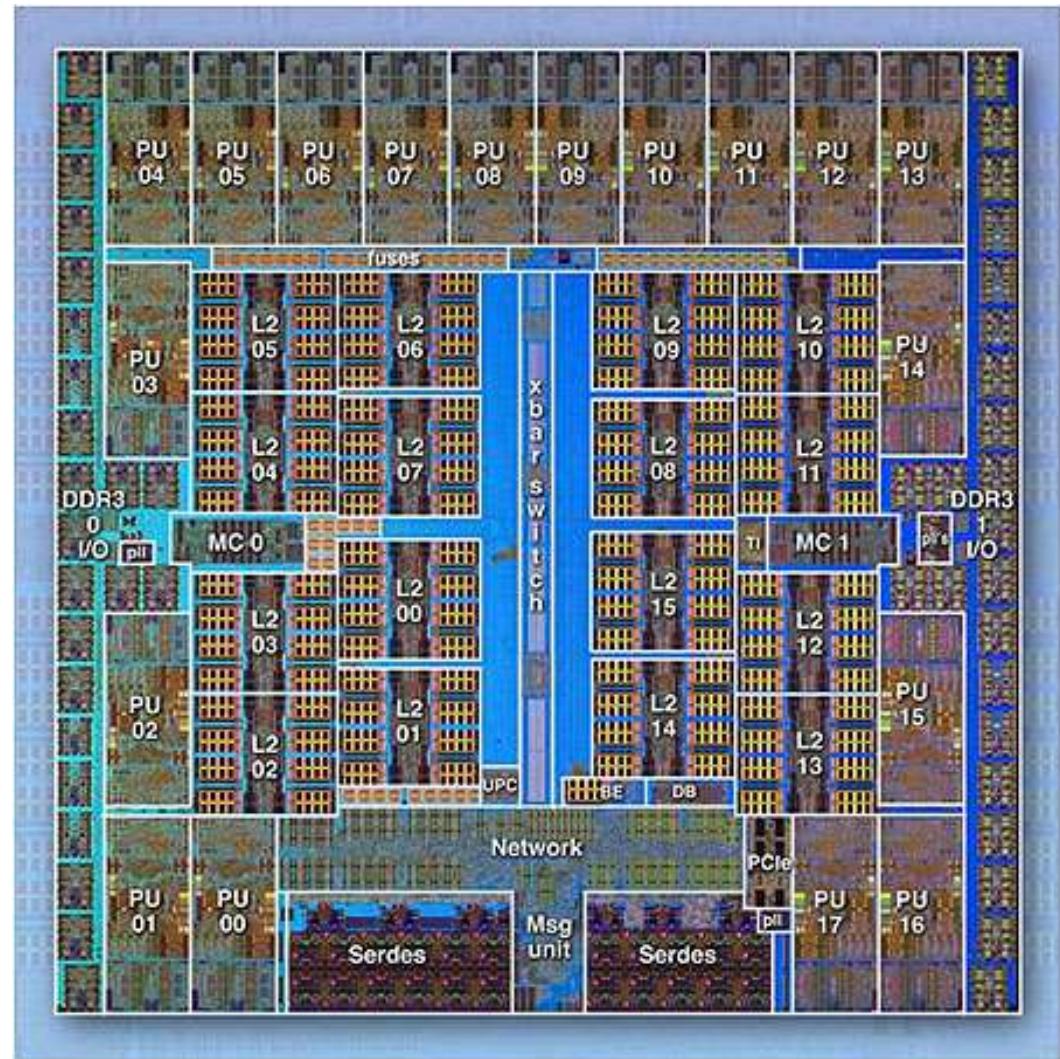


Комп'ютерні ресурси в паралельних обчисленнях

- Однопроцесорний комп'ютерний ресурс
(Single processor)
- Багатопроцесорний комп'ютерний ресурс
(Multiple processor)
- Розподілена комп'ютерна система
(Distributed system)

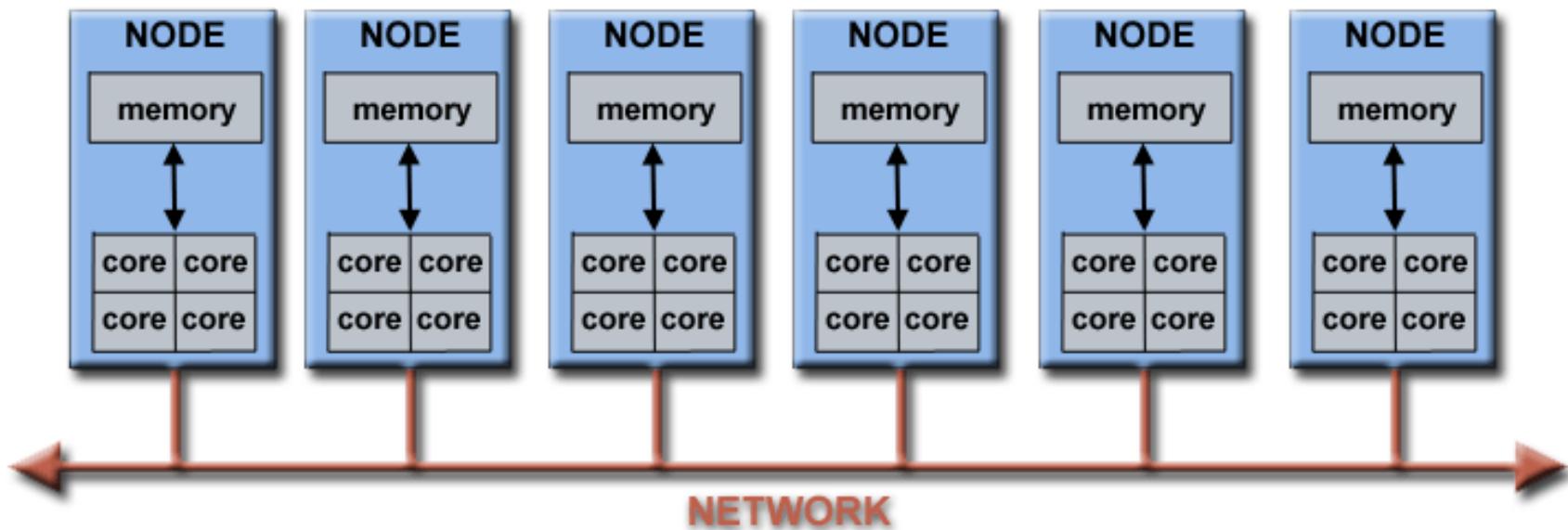
Багатопроцесорний комп'ютерний ресурс

IBM BG/Q
Computer Chip
with 18 cores (PU)
and 16 L2 Cache
units (L2)



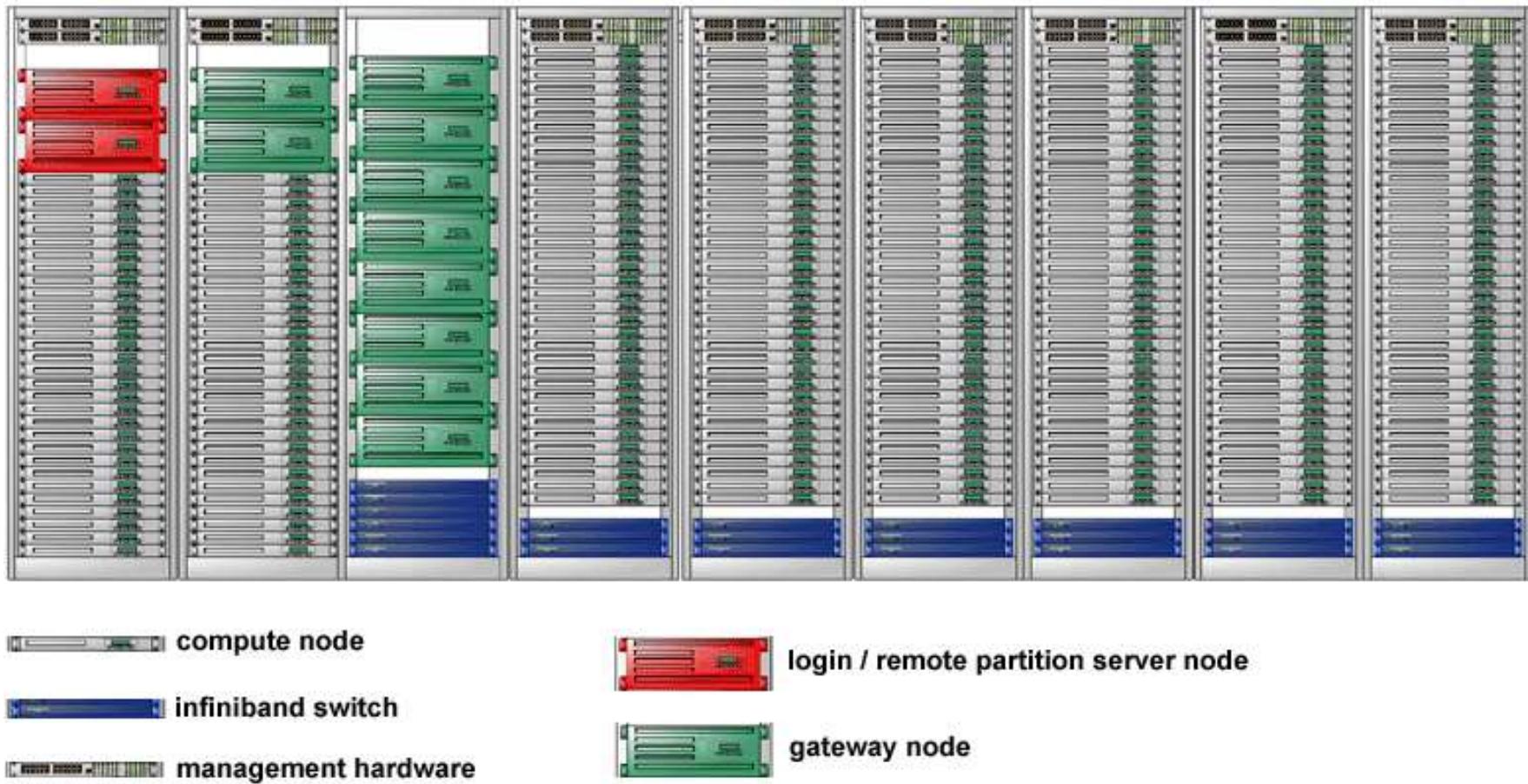
Розподілені системи. Комп'ютерний кластер

- Кожний вузол є багатопроцесорним комп'ютерним ресурсом
- Вузли з'єднані мережевим зв'язком
- окремі вузли мають спеціальне призначення



Приклад кластера:

LLNL(Lawrence Livermore National Laboratory) parallel computer cluster



Суперком'ютер

- Багатопроцесорний комп'ютер з продуктивністю набагато вищою ніж звичайний
- Найшвидші комп'ютери світу зареєстровані в TOP500 рейтингу
- Рекорд (пікової) продуктивності багатопроцесорного комп'ютерного ресурсу на листопад 2019 року = 148,6 petaflops належить суперком'ютеру [Summit](#) (US), другий у світовому рейтингу – суперком'ютер [Sierra](#) (US, Lawrence Livermore National Laboratory) з продуктивністю 94,6 petaflops, третій - [Sunway TaihuLight](#) (China)
 - PFLOPS (peta FLoating-point Operations Per Second)
- 206 суперком'ютерів зі списку TOP500 належать Китаю, 124 – Сполученим Штатам

Суперкомп'ютер Sunway TaihuLight (друге місце TOP500)



Кількість процесорів — 40 960. Кожен процесор містить 256 ядер загального призначення і 4 допоміжних ядра для керування. Загалом - 10 649 600 ядер

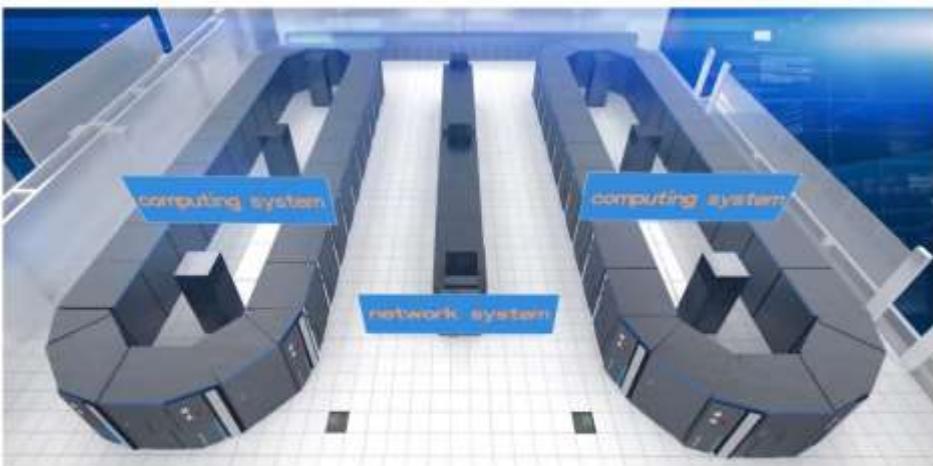


Figure 4: Overview of the Sunway TaihuLight System

TOP 5 Sites for November 2019

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Summit IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory <i>United States</i>	2 414 592	148 600	200 795	10,096
2	Sierra IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/NNSA/LLNL <i>United States</i>	1 572 480	94 640	125 712	7 438
3	Sunway TaihuLight Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi <i>China</i>	10 649 600	93 015	129 436	15 371
4	Tianhe-2A TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou <i>China</i>	4 981 760	61 444	100 679	18482
5	Frontera Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR , Dell EMCTexas Advanced Computing Center/Univ. of Texas <i>United States</i>	448 448	23 516	38 745	

Динаміка зростання продуктивності суперкомп'ютерів



Динаміка зростання продуктивності суперкомп'ютерів



Суперком'ютер СКІТ-4

Інститут кібернетики ім. В.М.Глушкова

- Продуктивність (пікова) – 25,6 Тфлопс
- 12 вузлів
- 192 ядра



Цілі використання паралельних обчислень

- Більш точне відтворення реального світу.
Моделювання фізичних, екологічних, логістичних, виробничих процесів
- Економія витрат часу/грошей
- Вирішення великих / складних задач
- Забезпечення одночасного вирішення задач (проектів, завдань)
- Ефективне використання територіально віддалених комп'ютерних ресурсів (грід-системи)
- Ефективне використання апаратного забезпечення комп'ютера

Області застосування паралельних обчислень



Galaxy Formation



Planetary Movements



Climate Change



Rush Hour Traffic



Plate Tectonics



Weather



Auto Assembly



Jet Construction



Drive-thru Lunch

Промислове та комерційне використання паралельних та розподілених обчислень

- "Big Data", інтелектуальний аналіз даних
- Розвідка нафти
- Веб-пошукові системи, web based business services
- Медична візуалізація та діагностика
- Фармацевтичні розробки
- Фінансове та економічне моделювання
- Управління національними та транснаціональними корпораціями
- Сучасна графіка та віртуальна реальність, особливо в індустрії розваг
- Мульти-медійні технології
- Середовища співпраці

Класифікація обчислювальних ресурсів. Таксономія Флінна

S I S D

Single Instruction stream
Single Data stream

S I M D

Single Instruction stream
Multiple Data stream

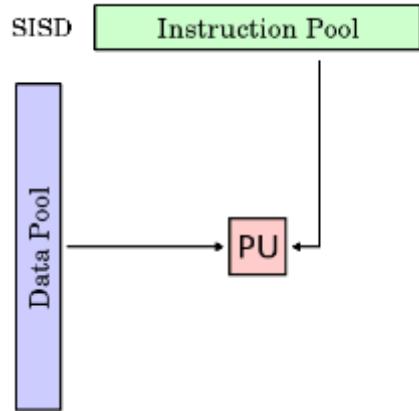
M I S D

Multiple Instruction stream
Single Data stream

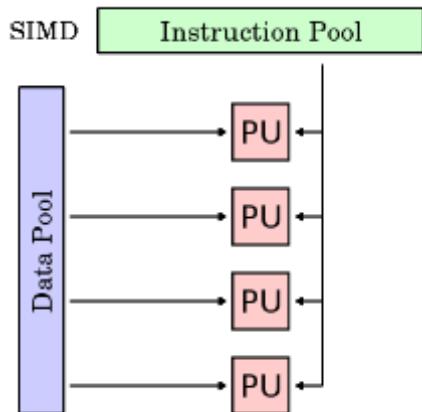
M I M D

Multiple Instruction stream
Multiple Data stream

Таксономія Фліна: SISD, SIMD

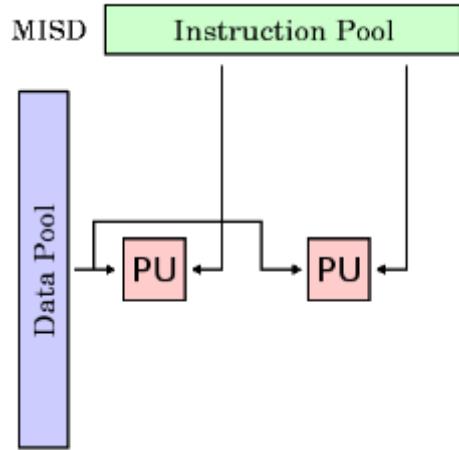


Приклад: мінікомп'ютери, робочі станції, одноядерний ПК.

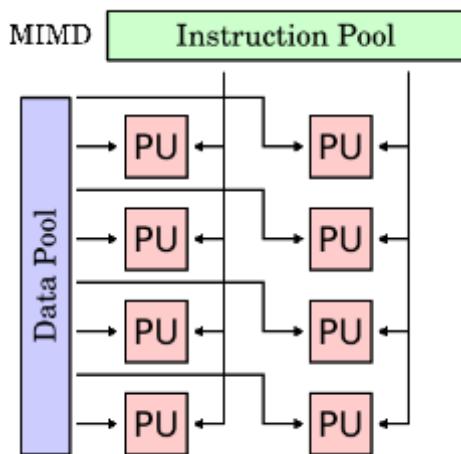


Приклад: обчислення з використанням графічного процесора (GPUs); обробка мультимедійної інформації

Таксономія Фліна: MISD, MIMD



Приклад: криптографічні алгоритми, що розшифровують одне повідомлення



Приклад: кластери, багатоядерні ПК

Програмне забезпечення паралельних обчислень

Багатопоточна технологія (Java, C#, Python, Go)

Бібліотеки паралельних обчислень (OpenMP)

Програмне забезпечення розподілених
обчислень (OpenMPI, RMI, Apache Hadoop)

Навчальний матеріал

- В цій лекції використаний навчальний матеріал курсу паралельних обчислень *Blaise Barney*

Клас Thread

Створення та запуск потоків

Інна Вячеславівна Стеценко
професор кафедри ІПІ НТУУ «КПІ ім. Ігоря Сікорського»,
д.т.н., професор

Поняття процесу і потоку

- Процес – це самостійне середовище виконання обчислень, як правило, з окремою виділеною пам'яттю. Програма, яка запускається віртуальною машиною java, є за замовчуванням одним процесом. Java застосунок може створювати додаткові процеси. Для цього передбачений клас `java.lang.ProcessBuilder`. Кожний процес містить хоч один потік.
- Потік – це «легкий» процес, що запускається в межах певного процесу. Потоки, що виконуються в межах одного процесу, мають спільні ресурси процесу.
- Отже, розрізняються багатопроцесне виконання програм та багатопоточне. У наступних 9 лекціях розглядаємо багатопоточність.

Клас Thread

- Для створення потоків в Java використовують клас `java.lang.Thread`. Кожний об'єкт типу `Thread` є потоком.
- Документація класу
<https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#Thread-javax.lang.Runnable->

← → C https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Apps Яндекс Поста Google Translate Avast Account Multithreaded, Paral... Лекції - Паралельні... Library | Scribd » Other bookmarks

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.lang

Class Thread

java.lang.Object

java.lang.Thread

All Implemented Interfaces:

Runnable

Direct Known Subclasses:

ForkJoinWorkerThread

```
public class Thread  
extends Object  
implements Runnable
```

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also

Інтерфейс java.lang.Runnable

- Клас імплементує інтерфейс `java.lang.Runnable`. Метод `run()` цього інтерфейсу призначений для опису дій, що будуть виконуватись потоком.
- `Runnable`-об'єкти можуть використовуватися для створення потоків, але не є потоками

Інтерфейс Runnable

https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

Apps Яндекс Пошта Google Translate Avast Account Multithreaded, Paral... Лекції - Паралельні... Library | Scribd Other bookmarks

Method Summary

All Methods Instance Methods Abstract Methods

Modifier and Type	Method and Description
void	run() When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

Method Detail

run

```
void run()
```

When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

The general contract of the method run is that it may take any action whatsoever.

See Also:

`Thread.run()`

Способи створення потоку: спадкування Thread

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {  
    long minPrime;  
    PrimeThread(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        ...  
    }  
}
```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);  
p.start();
```

The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

Способи створення потоку: використання Runnable-об'єкта

The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

```
class PrimeRun implements Runnable {  
    long minPrime;  
    PrimeRun(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run(){  
        // compute primes larger than minPrime  
        . . .  
    }  
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);  
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Найпростіші дії з потоками

Потік	Створення потоку	Запуск потоку	Призупинка потоку	Завершення дії потоку	Переривання дії потоку
об'єкт типу Runnable	new Thread (new Runnable) new MyThread(), де MyThread extends Thread	thread.start()	Thread.sleep()	Завершення методу run()	thread.interrupt()

Стани потоку

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Apps Яндекс Пошта Google Translate Avast Account Multithreaded, Paral... Лекції - Паралельні... Library | Scribd Other bookmarks

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
static class	Thread.State A thread state.
static interface	Thread.UncaughtExceptionHandler Interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception.

Field Summary

Fields

Modifier and Type	Field and Description
static int	MAX_PRIORITY The maximum priority that a thread can have.
static int	MIN_PRIORITY The minimum priority that a thread can have.
static int	NORM_PRIORITY

Стани потоку

Enum Class Thread.State

```
java.lang.Object
  java.lang.Enum<Thread.State>
    java.lang.Thread.State
```

All Implemented Interfaces:

Serializable, Comparable<Thread.State>, Constable

Enclosing class:

Thread

```
public static enum Thread.State
extends Enum<Thread.State>
```

A thread state. A thread can be in one of the following states:

- NEW
A thread that has not yet started is in this state.
- RUNNABLE
A thread executing in the Java virtual machine is in this state.
- BLOCKED
A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.

Основні методи класу Thread

- Змінюють стан або параметри потоку
 - start()
 - sleep()
 - interrupt()
 - yield()
 - join()
- Інформують про стан та параметри потоку
 - currentThread()
 - isAlive()
 - interrupted()

Метод sleep()

Виклик цього методу переводить потік у стан TIMED_WAITING.

Інші потоки отримують можливість захопити звільнений ресурс.

Після завершення часової затримки, вказаної в методі sleep(), потік переходить у стан RUNNABLE і разом з іншими потоками намагається знову захопити ресурс на виконання обчислень. Це означає, що наступна після виклику методу sleep() інструкція виконається не обов'язково через затримку, вказану в методі sleep(), а виявиться більшою. Тільки коли ресурс успішно захоплено, потік продовжує обчислення.

static void

sleep(long millis)

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

static void

sleep(long millis, int nanos)

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers.

Пріоритет потоку

За замовчуванням потік, який створюється має середнє значення пріоритету.

Пріоритет потоку можна змінити, застосовуючи метод `setPriority()`.

Вище значення пріоритету не гарантує першочергове виконання, проте забезпечує більшу ймовірність доступу до захоплення ресурсу. При цьому ресурс розподіляється між потоками з меншим пріоритетом і більшим пріоритетом операційною системи з позицій «справедливості». Тому, якщо потоків з низьким пріоритетом багато, вони можуть домінувати при захопленні ресурсу. Якщо потоків з високим пріоритетом багато, вони будуть конкурувати між собою за захоплення ресурсу.

Методи:

`getPriority()`
`setPriority()`

Поля:

`MAX_PRIORITY`
`MIN_PRIORITY`
`NORM_PRIORITY`

Приклад розробки багатопоточної програми «Більярдні кульки»

[Cay Horstmann Core Java Volume I--Fundamentals (Core Series) 11th Edition]

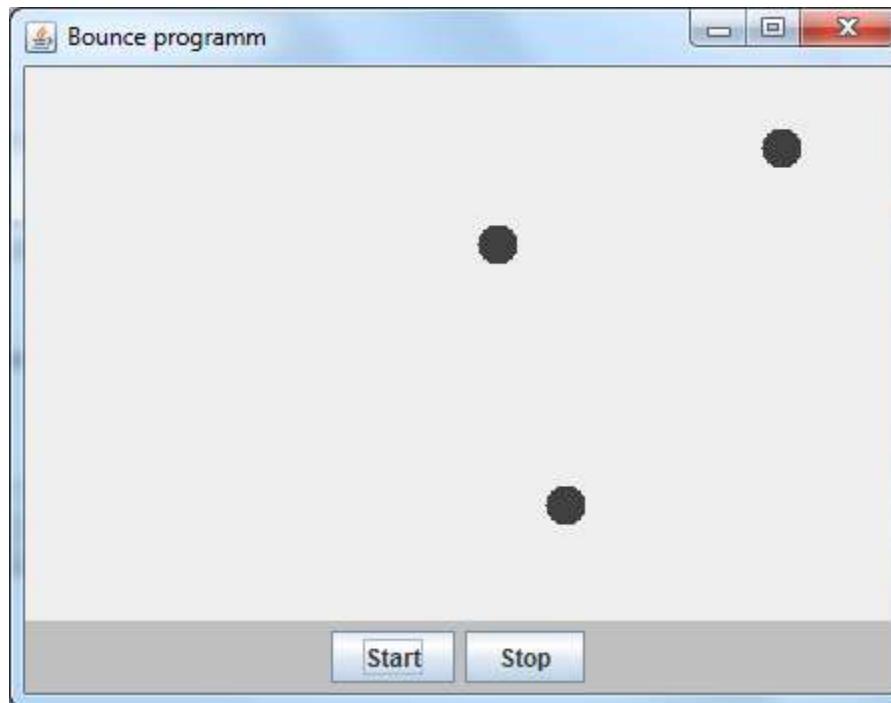
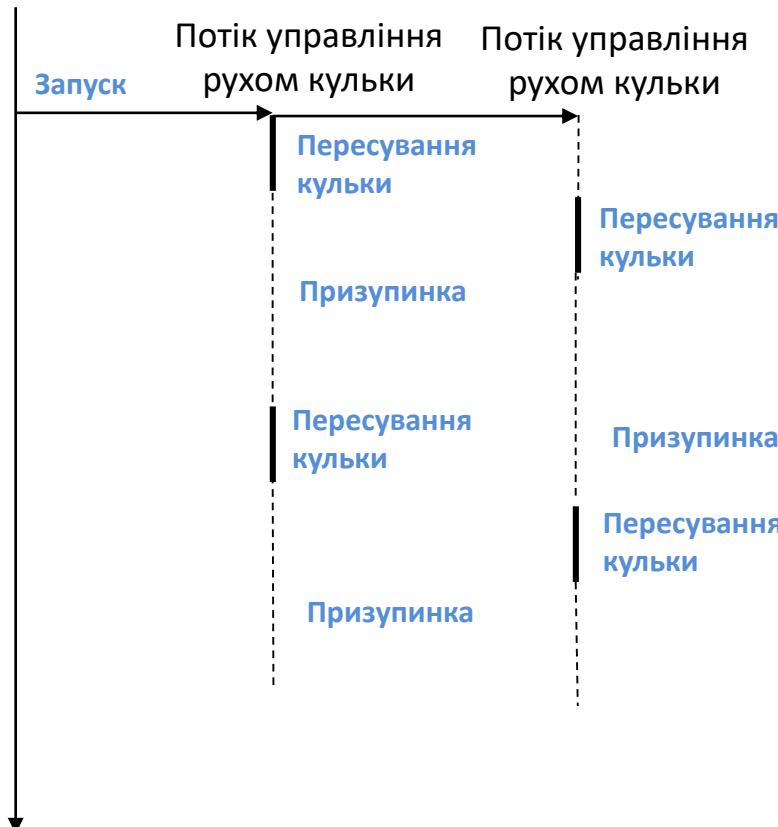


Схема управління потоком

Потік управління
подіями користувача



Bounce - NetBeans IDE 8.0.2

Файл Правка Вид Переход Источник Реорганизация кода Выполнить Отладка Профилировать Группа Сервис Окно Справка Помощь (Ctrl+I)

настройка по умолчанию

Файлы Службы

аппlet_HelloWorld Bounce

Пакеты исходных кодов

bounce

Ball.java BallCanvas.java BallThread.java Bounce.java BounceFrame.java

Библиотеки

BounceNonThread graphicsSquare HiWorld ParallelAlgorithms PetriObjPaint PetriObjPaint_14 RefusalAndRepair ResourceAllocation Testing Threads TrafficControl TutorialTest

Источник История

7
8 /**
9 *
10 * @author Inna
11 */
12 public class BallThread extends Thread {
13 private Ball b;
14
15 public BallThread(Ball ball){
16 b = ball;
17 }
18
19 @Override
20 public void run(){
21 try{
22 for(int i=1; i<10000; i++){
23 b.move();
24 System.out.println("Thread name = " + Thread.currentThread().getName());
25 Thread.sleep(5);
26 }
27 } catch(InterruptedException ex){
28 }
29 }
30 }
31

run - Навигатор

Компоненты <пуск...>

BallThread :: Thread

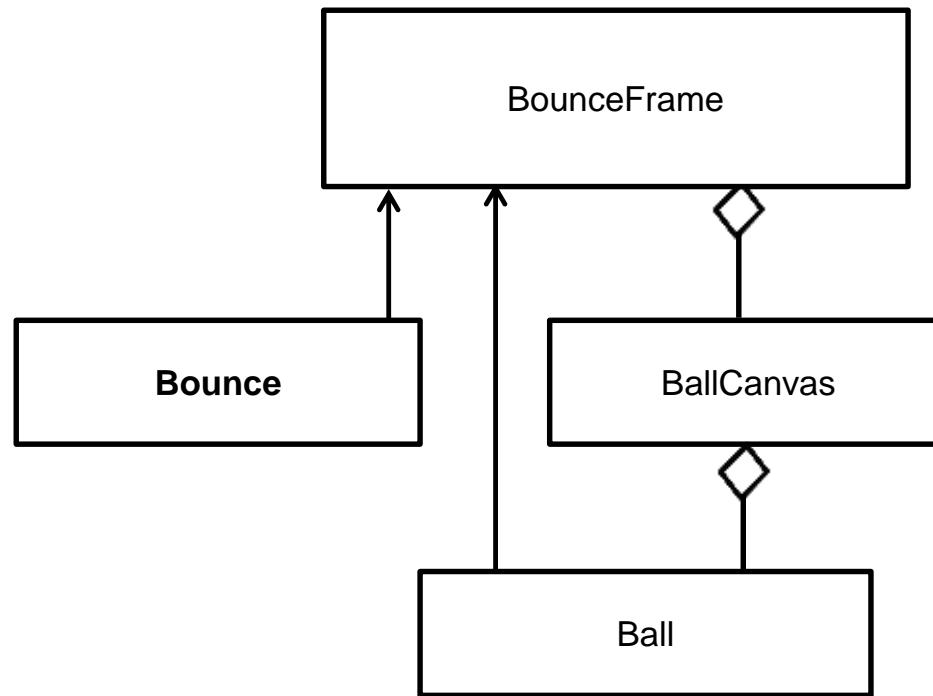
BallThread(Ball ball)
run()
b : Ball

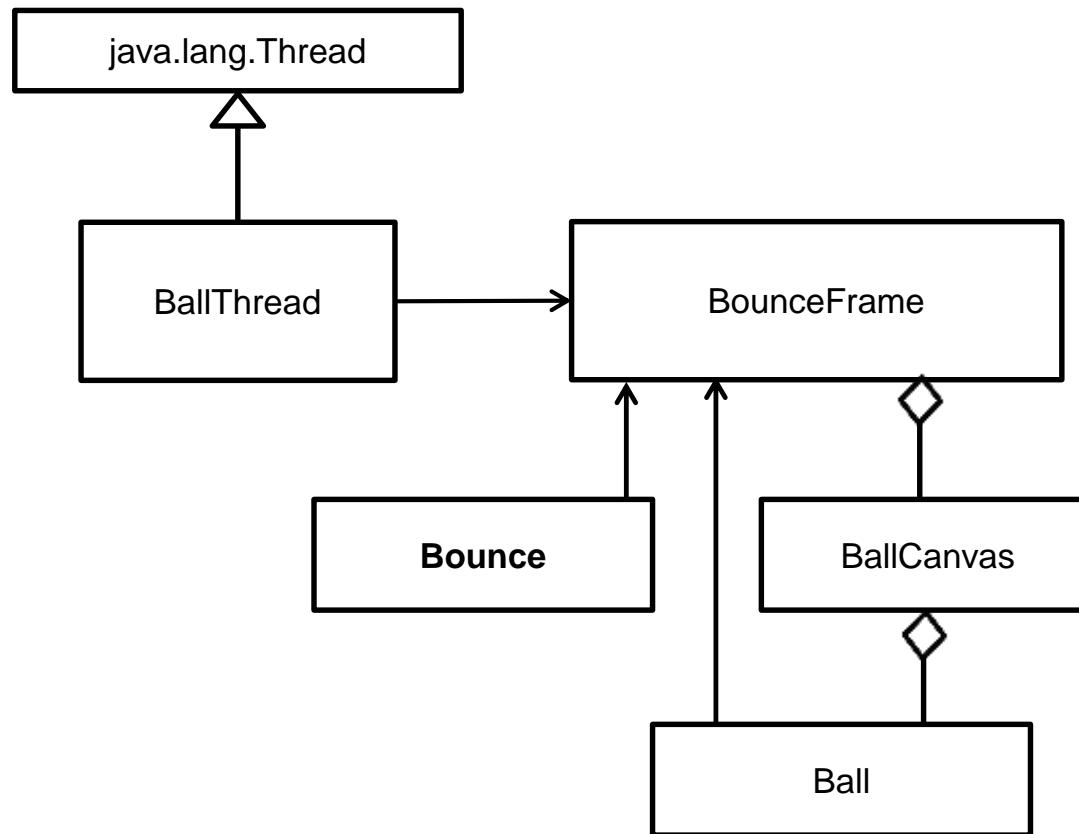
бинарные файлы

Выход - Bounce (run)

Thread name = Thread-4
Thread name = Thread-3
Thread name = Thread-4
Thread name = Thread-2
Thread name = Thread-3
Thread name = Thread-4
Thread name = Thread-2
Thread name = Thread-3
Thread name = Thread-2
Thread name = Thread-3
Thread name = Thread-4
Thread name = Thread-3
Thread name = Thread-4
Thread name = Thread-2
Thread name = Thread-3
Thread name = Thread-2
Thread name = Thread-3
Thread name = Thread-4
Thread name = Thread-2
Thread name = Thread-4
Thread name = Thread-3
Thread name = Thread-2

Bounce (run) running... 21:13 INS





Клас Ball

```
public class Ball {
    private Component canvas;
    private static final int XSIZE = 20;
    private static final int YSIZE = 20;
    private int x = 0;
    private int y = 0;
    private int dx = 2;
    private int dy = 2;

    public Ball(Component c) {
        this.canvas = c;
        if (Math.random() < 0.5) {
            x = new Random().nextInt(this.canvas.getWidth());
            y = 0;
        } else {
            x = 0;
            y = new Random().nextInt(this.canvas.getHeight());
        }
    }

    public void draw(Graphics2D g2) {
        g2.setColor(Color.darkGray);
        g2.fill(new Ellipse2D.Double(x, y, XSIZEx, YSIZE));
    }

    public void move() {
        x += dx;
        y += dy;
        if (x < 0) {
            x = 0; dx = -dx;
        }
        if (x + XSIZEx >= this.canvas.getWidth()) {
            x = this.canvas.getWidth() - XSIZEx; dx = -dx;
        }
        if (y < 0) {
            y = 0; dy = -dy;
        }
        if (y + YSIZE >= this.canvas.getHeight()) {
            y = this.canvas.getHeight() - YSIZE; dy = -dy;
        }
        this.canvas.repaint();
    }
}
```

Клас BallCanvas

```
package bounce;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.ArrayList;
import javax.swing.JPanel;

/**
 *
 * @author Inna
 */
public class BallCanvas extends JPanel{
    private ArrayList<Ball> balls = new ArrayList<>();

    public void add(Ball b){
        this.balls.add(b);
    }
    @Override
    public void paintComponent(Graphics g){
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        for(int i=0; i<balls.size();i++){
            Ball b = balls.get(i);
            b.draw(g2);
        }
        //this.setBackground(Color.green);
    }
}
```

Клас BallThread

```
package bounce;

/**
 *
 * @author Inna
 */
public class BallThread extends Thread {
    private Ball b;

    public BallThread(Ball ball) {
        b = ball;
    }

    @Override
    public void run() {
        try{
            for(int i=1; i<10000; i++){
                b.move();
                System.out.println("Thread name = " + Thread.currentThread().getName());
                Thread.sleep(5);

            }
        } catch(InterruptedException ex){
            ex.printStackTrace();
        }
    }
}
```

Клас BounceFrame

```
package bounce;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
/* @author Inna */
public class BounceFrame extends JFrame {
    private BallCanvas canvas;
    public static final int WIDTH = 450;
    public static final int HEIGHT = 350;

    public BounceFrame() {
        this.setSize(WIDTH, HEIGHT);
        this.setTitle("Bounce programm");

        this.canvas = new BallCanvas();
        System.out.println("In Frame Thread name = " + Thread.currentThread().getName());
        Container content = this.getContentPane();
        content.add(this.canvas, BorderLayout.CENTER);

        JPanel buttonPanel = new JPanel();
        buttonPanel.setBackground(Color.lightGray);

        JButton buttonStart = new JButton("Start");
        JButton buttonStop = new JButton("Stop");
```

Клас BounceFrame

```
buttonStart.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {

        Ball b = new Ball(canvas);
        canvas.add(b);
        BallThread thread = new BallThread(b);
        thread.start();
        System.out.println("Thread name = " + thread.getName());
    }
});

buttonStop.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});

buttonPanel.add(buttonStart);
buttonPanel.add(buttonStop);

content.add(buttonPanel, BorderLayout.SOUTH);
}
```

Клас Bounce

```
package bounce;

import javax.swing.JFrame;

/**
 * @author Inna
 */
public class Bounce {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        BounceFrame frame = new BounceFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

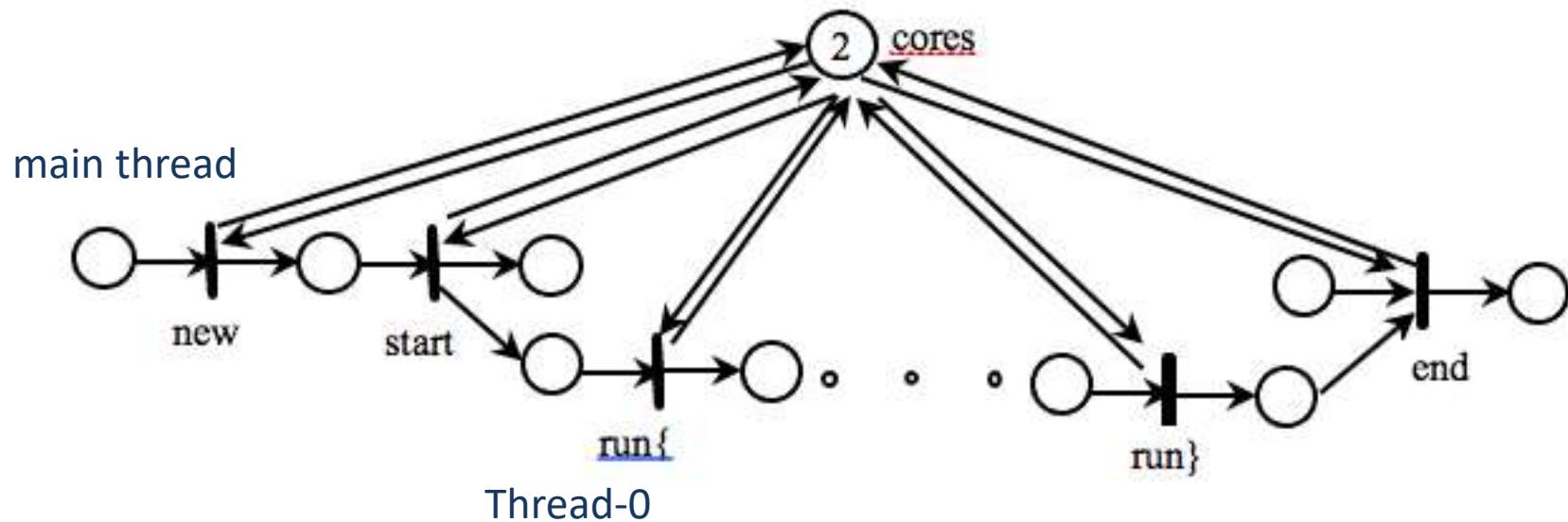
        frame.setVisible(true);
        System.out.println("Thread name = " + Thread.currentThread().getName());
    }
}
```

Багатопоточність!

Выход - Bounce (run) ■

	Thread name = Thread-32
	Thread name = Thread-37
	Thread name = Thread-33
	Thread name = Thread-9
	Thread name = Thread-23
	Thread name = Thread-31
	Thread name = Thread-17
	Thread name = Thread-11
	Thread name = Thread-21
	Thread name = Thread-19
	Thread name = Thread-15
	Thread name = Thread-3
	Thread name = Thread-7
	Thread name = Thread-27
	Thread name = Thread-35
	Thread name = Thread-13
	Thread name = Thread-5
	Thread name = Thread-29
	Thread name = Thread-41

Моделювання роботи потоків мережами Петрі



Завдання 1 до комп'ютерного практикуму

«Розробка потоків та дослідження пріоритету запуску потоків»

1. Реалізуйте програму імітації руху більярдних кульок, в якій рух кожної кульки відтворюється в окремому потоці (див. презентацію «Створення та запуск потоків в java» та приклад). Спостерігайте роботу програми при збільшенні кількості кульок. Поясніть результати спостереження. Опишіть переваги потокової архітектури програм. **10 балів.**
2. Модифікуйте програму так, щоб при потраплянні в «лузу» кульки зникали, а відповідний потік завершував свою роботу. Кількість кульок, яка потрапила в «лузу», має динамічно відображатись у текстовому полі інтерфейсу програми. **10 балів.**
3. Виконайте дослідження параметру priority потоку. Для цього модифікуйте програму «Більярдна кулька» так, щоб кульки червоного кольору створювались з вищим пріоритетом потоку, в якому вони виконують рух, ніж кульки синього кольору. Спостерігайте рух червоних та синіх кульок при збільшенні загальної кількості кульок. Проведіть такий експеримент. Створіть багато кульок синього кольору (з низьким пріоритетом) і одну червоного кольору, які починають рух в одному й тому ж самому місці більярдного стола, в одному й тому ж самому напрямку та з однаковою швидкістю. Спостерігайте рух кульки з більшим пріоритетом. Повторіть експеримент кілька разів, значно збільшуючи кожного разу кількість кульок синього кольору. Зробіть висновки про вплив пріоритету потоку на його роботу в залежності від загальної кількості потоків. **20 балів.**
4. Побудуйте ілюстрацію для методу join() класу Thread з використанням руху більярдних кульок різного кольору. Поясніть результат, який спостерігається. **10 балів.**
5. Створіть два потоки, один з яких виводить на консоль символ ‘-‘, а інший – символ ‘|’. Запустіть потоки в основній програмі так, щоб вони виводили свої символи в рядок. Виведіть на консоль 100 таких рядків. Поясніть виведений результат. **10 балів.** Використовуючи найпростіші методи управління потоками, добийтесь почергового виведення на консоль символів. **15 балів.**
6. Створіть клас Counter з методами increment() та decrement(), які збільшують та зменшують значення лічильника відповідно. Створіть два потоки, один з яких збільшує 100000 разів значення лічильника, а інший – зменшує 100000 разів значення лічильника. Запустіть потоки на одночасне виконання. Спостерігайте останнє значення лічильника. Поясніть результат. **10 балів.** Використовуючи синхронізований доступ, добийтесь правильної роботи лічильника при одночасній роботі з ним двох і більше потоків. Опрацюйте використання таких способів синхронізації: синхронізований метод, синхронізований блок, блокування об’єкта. Порівняйте способи синхронізації. **15 балів.**

Алгоритми паралельного множення матриць

Інна Вячеславівна Стеценко
професор кафедри ІПІ НТУУ «КПІ ім. Ігоря Сікорського»,
д.т.н., професор

Послідовне множення матриць

$$\begin{pmatrix} a_{1,1} & \cdots & & \\ \vdots & \ddots & \vdots & \\ & \cdots & a_{n-1,s-1} \end{pmatrix} \times \begin{pmatrix} b_{1,1} & \cdots & & \\ \vdots & \ddots & \vdots & \\ & \cdots & b_{s-1,m-1} \end{pmatrix} = \begin{pmatrix} c_{1,1} & \cdots & & \\ \vdots & \ddots & \vdots & \\ & \cdots & c_{n-1,m-1} \end{pmatrix}$$

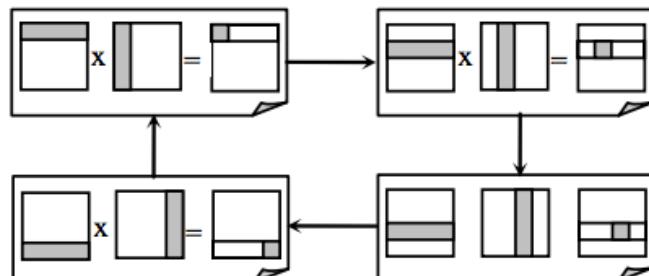
$$c_{i,j} = \sum_{k=0}^{s-1} a_{i,k} \cdot b_{k,j}, \quad 0 \leq i < n, \quad 0 \leq j < m$$

Стрічкові алгоритми паралельного множення матриць

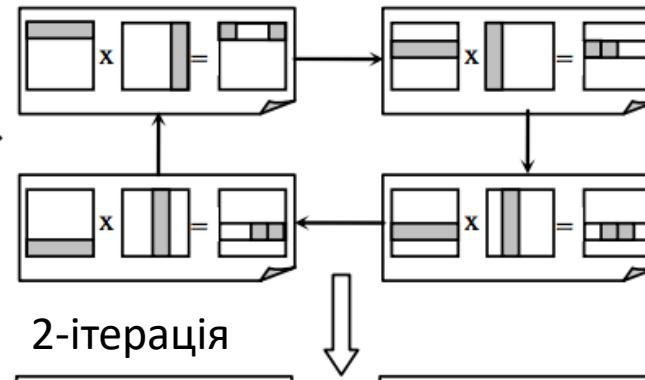
Перший алгоритм:

кожний процес розраховує рядок результуючої матриці,
кожна ітерація – один елемент рядка результуючої матриці

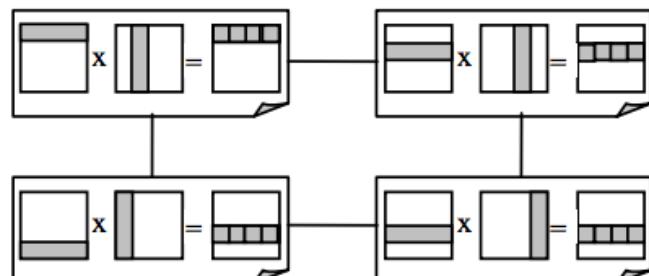
0-ітерація



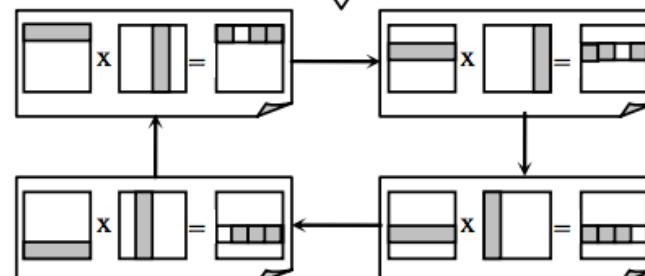
1-ітерація



3-ітерація



2-ітерація



$\forall i$ – процесу, $0 \leq i < n$, j – ітерації, $0 \leq j < m$:

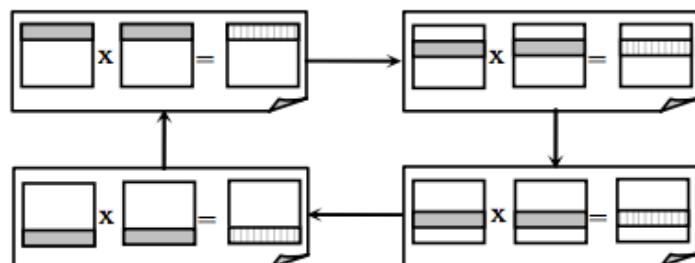
$$c_{i,j} = \sum_{k=0}^{s-1} a_{i,k} \cdot b_{k,j}$$

Ітерація	Процес 0: 0-рядок А $c_{00} = \sum a_{0k} b_{k0}$	Процес 1: 1-рядок А $c_{11} = \sum a_{1k} b_{k1}$	Процес 2: 2-рядок А $c_{22} = \sum a_{2k} b_{k2}$	Процес 3: 3-рядок А $c_{33} = \sum a_{3k} b_{k3}$
0	0-рядок А, 0 -стовпець В $c_{00} = \sum a_{0k} b_{k0}$	1-рядок А, 1 -стовпець В $c_{11} = \sum a_{1k} b_{k1}$	2-рядок А, 2 -стовпець В $c_{22} = \sum a_{2k} b_{k2}$	3-рядок А, 3 -стовпець В $c_{33} = \sum a_{3k} b_{k3}$
1	0-рядок А, 3 -стовпець В $c_{03} = \sum a_{0k} b_{k3}$	1-рядок А, 0 -стовпець В $c_{10} = \sum a_{1k} b_{k0}$	2-рядок А, 1 -стовпець В $c_{21} = \sum a_{2k} b_{k1}$	3-рядок А, 2 -стовпець В $c_{32} = \sum a_{3k} b_{k2}$
2	0-рядок А, 2 -стовпець В $c_{02} = \sum a_{0k} b_{k2}$	1-рядок А, 3 -стовпець В $c_{13} = \sum a_{1k} b_{k3}$	2-рядок А, 0 -стовпець В $c_{20} = \sum a_{2k} b_{k0}$	3-рядок А, 1 -стовпець В $c_{31} = \sum a_{3k} b_{k1}$
3	0-рядок А, 1 -стовпець В $c_{01} = \sum a_{0k} b_{k1}$	1-рядок А, 2 -стовпець В $c_{12} = \sum a_{1k} b_{k2}$	2-рядок А, 3 -стовпець В $c_{23} = \sum a_{2k} b_{k3}$	3-рядок А, 0 -стовпець В $c_{30} = \sum a_{3k} b_{k0}$

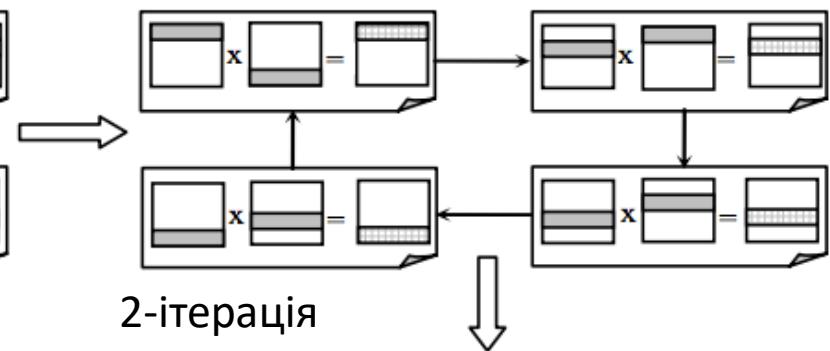
Другий алгоритм:

кожний процес розраховує рядок результуючої матриці,
кожна ітерація додає до кожного елемента рядка результуючої
матриці доданок

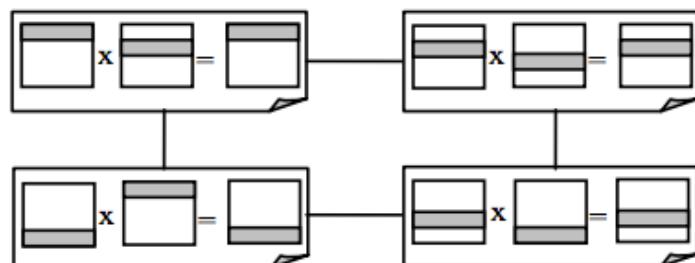
0-ітерація



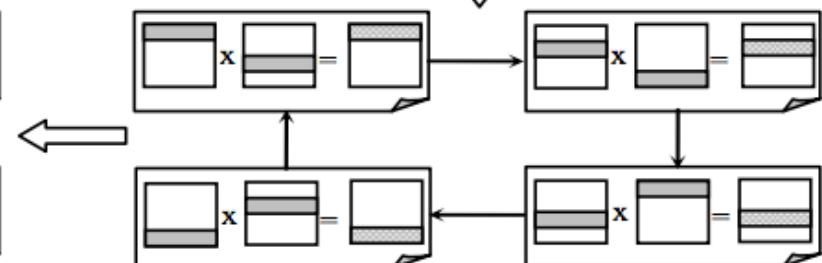
1-ітерація



3-ітерація



2-ітерація



$$\forall i - \text{процесу}, 0 \leq i < n, \quad k - \text{iтерації}, 0 \leq k < s$$

$$c_{i,j} = c_{i,j} + a_{i,k} \cdot b_{k,j}, \quad j = \overline{0, m-1}$$

Ітерація	Процес 0: 0-рядок А	Процес 1: 1-рядок А	Процес 2: 2-рядок А	Процес 3: 3-рядок А
0	0 -рядок В $c_{0j} = a_{00}b_{0j},$ $j = 0,..m - 1$	1 -рядок В $c_{1j} = a_{11}b_{1j},$ $j = 0,..m - 1$	2 - рядок В $c_{2j} = a_{22}b_{2j},$ $j = 0,..m - 1$	3 -рядок В $c_{3j} = a_{33}b_{3j},$ $j = 0,..m - 1$
1	3 - рядок В $c_{0j} = c_{0j} + a_{03}b_{3j},$ $j = 0,..m - 1$	0 -рядок В $c_{1j} = c_{1j} + a_{10}b_{0j},$ $j = 0,..m - 1$	1 - рядок В $c_{2j} = c_{2j} + a_{21}b_{1j},$ $j = 0,..m - 1$	2 -рядок В $c_{3j} = c_{3j} + a_{32}b_{2j},$ $j = 0,..m - 1$
2	2 -рядок В $c_{0j} = c_{0j} + a_{02}b_{2j},$ $j = 0,..m - 1$	3 - рядок В $c_{1j} = c_{1j} + a_{13}b_{3j},$ $j = 0,..m - 1$	0 -рядок В $c_{2j} = c_{2j} + a_{20}b_{0j},$ $j = 0,..m - 1$	1 -рядок В $c_{3j} = c_{3j} + a_{31}b_{1j},$ $j = 0,..n - 1$
3	1 -рядок В $c_{0j} = c_{0j} + a_{01}b_{1j},$ $j = 0,..m - 1$	2 - рядок В $c_{1j} = c_{1j} + a_{12}b_{2j},$ $j = 0,..m - 1$	3 - рядок В $c_{2j} = c_{2j} + a_{23}b_{3j},$ $j = 0,..m - 1$	0 - рядок В $c_{3j} = c_{3j} + a_{30}b_{0j},$ $j = 0,..m - 1$

Процес 0 у випадку матриці 4×4

0-ітерація: $c_{0,0} := a_{0,0} \cdot b_{0,0}$

$$c_{0,1} := a_{0,0} \cdot b_{0,1}$$

$$c_{0,2} := a_{0,0} \cdot b_{0,2}$$

$$c_{0,3} := a_{0,0} \cdot b_{0,3}$$

1-ітерація: $c_{0,0} := c_{0,0} + a_{0,3} \cdot b_{3,0}$

$$c_{0,1} := c_{0,1} + a_{0,3} \cdot b_{3,1}$$

$$c_{0,2} := c_{0,2} + a_{0,3} \cdot b_{3,2}$$

$$c_{0,3} := c_{0,3} + a_{0,3} \cdot b_{3,3}$$

2-ітерація: $c_{0,0} := c_{0,0} + a_{0,2} \cdot b_{2,0}$

$$c_{0,1} := c_{0,1} + a_{0,2} \cdot b_{2,1}$$

$$c_{0,2} := c_{0,2} + a_{0,2} \cdot b_{2,2}$$

$$c_{0,3} := c_{0,3} + a_{0,2} \cdot b_{2,3}$$

3-ітерація: $c_{0,0} := c_{0,0} + a_{0,1} \cdot b_{1,0}$

$$c_{0,1} := c_{0,1} + a_{0,1} \cdot b_{1,1}$$

$$c_{0,2} := c_{0,2} + a_{0,1} \cdot b_{1,2}$$

$$c_{0,3} := c_{0,3} + a_{0,1} \cdot b_{1,3}$$

Результати експериментів

Matrix Size	Serial Algorithm	2 processors		4 processors		8 processors	
		Time	Speed Up	Time	Speed Up	Time	Speed Up
500	0,8752	0,3758	2,3287	0,1535	5,6982	0,0968	9,0371
1000	12,8787	5,4427	2,3662	2,2628	5,6912	0,6998	18,4014
1500	43,4731	20,9503	2,0750	11,0804	3,9234	5,1766	8,3978
2000	103,0561	45,7436	2,2529	21,6001	4,7710	9,4127	10,9485
2500	201,2915	99,5097	2,0228	56,9203	3,5363	18,3303	10,9813
3000	347,8434	171,9232	2,0232	111,9642	3,1067	45,5482	7,6368

Ефективність паралельного алгоритму оцінюється прискоренням (speedup), що визначає у скільки разів зменшився час виконання алгоритму:

$$speedup = \frac{T_{sequential}}{T_{parallel}}$$

Алгоритм Фокса

$$\begin{pmatrix} A_{1,1} & \cdots & & \\ \vdots & \ddots & \vdots & \\ & \cdots & A_{q-1,q-1} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & \cdots & & \\ \vdots & \ddots & \vdots & \\ & \cdots & B_{q-1,q-1} \end{pmatrix} = \begin{pmatrix} C_{1,1} & \cdots & & \\ \vdots & \ddots & \vdots & \\ & \cdots & C_{q-1,q-1} \end{pmatrix}$$

$C_{ij} = \sum_{s=0}^{q-1} A_{is}B_{sj}$, де A_{is}, B_{sj} - блоки матриць A і B розміром $m \times m$

Ітерація
0
1
2
3

$$\begin{aligned} C_{ij} = & A_{i,i}B_{i,j} + \\ & + A_{i,(i+1)}B_{(i+1),j} + \dots \\ & + A_{i,(i+k) \bmod q}B_{(i+k) \bmod q, j} + \dots \\ & + A_{i,(i-1)}B_{(i-1),j} \end{aligned}$$

Алгоритм Фокса

- Спочатку виконується розсылка блоків A_{ii} та B_{ij} в усі процеси P_{ij} :

$$A_{ii}, B_{ij} \rightarrow P_{ij}$$

Знаходиться їх добуток і присвоюється результиуючій матриці процесу

$$C_{ij} = A_{ii}B_{ij}$$

- На кожній ітерації:

- виконується циклічна пересилка блоків А (по другому індексу) і В (по першому індексу):

якщо $i+1 == q$, то $i+1:=0$,

$$A_{i(i+1)}, B_{(i+1)j} \rightarrow P_{ij} .$$

- знаходиться добуток блоків, які зберігаються в процесі, і додається до результиуючої матриці процесу

Таким чином,

$$C_{ij} += A_{i,(i+k)\bmod q} B_{(i+k)\bmod q,j} , \text{де } k - \text{номер ітерації}$$

- Результиуючі матриці процесів передаються в головний процес, де з них складається результиуюча матриця.

Якщо розмір матриць A і B $n \times n$, а кількість блоків по горизонталі і діагоналі q , то алгоритм реалізується на q^2 процесах, а розмір блоків матриць $m=n/q$.

Алгоритм Фокса.

Приклад для 4 процесів ($q=2$)

Крок 0

Розсилка в процеси блоків вихідних матриць

$A_{ij} \rightarrow P_{ij}$,

$B_{ij} \rightarrow P_{ij}$.

Розрахунок результиуючої матриці C_{ij}

$A_{ij} \cdot B_{ij} \rightarrow C_{ij}$

A_{00} B_{00} $C_{00} = A_{00} B_{00}$	A_{00} B_{01} $C_{01} = A_{00} B_{01}$
A_{11} B_{10} $C_{10} = A_{11} B_{10}$	A_{11} B_{11} $C_{11} = A_{11} B_{11}$

Крок 1

Розсилка в процеси наступних блоків вихідних матриць: $k = (i+1) \bmod 2$

$A_{ik} \rightarrow P_{ij}$,

$B_{kj} \rightarrow P_{ij}$.

Розрахунок результиуючої матриці

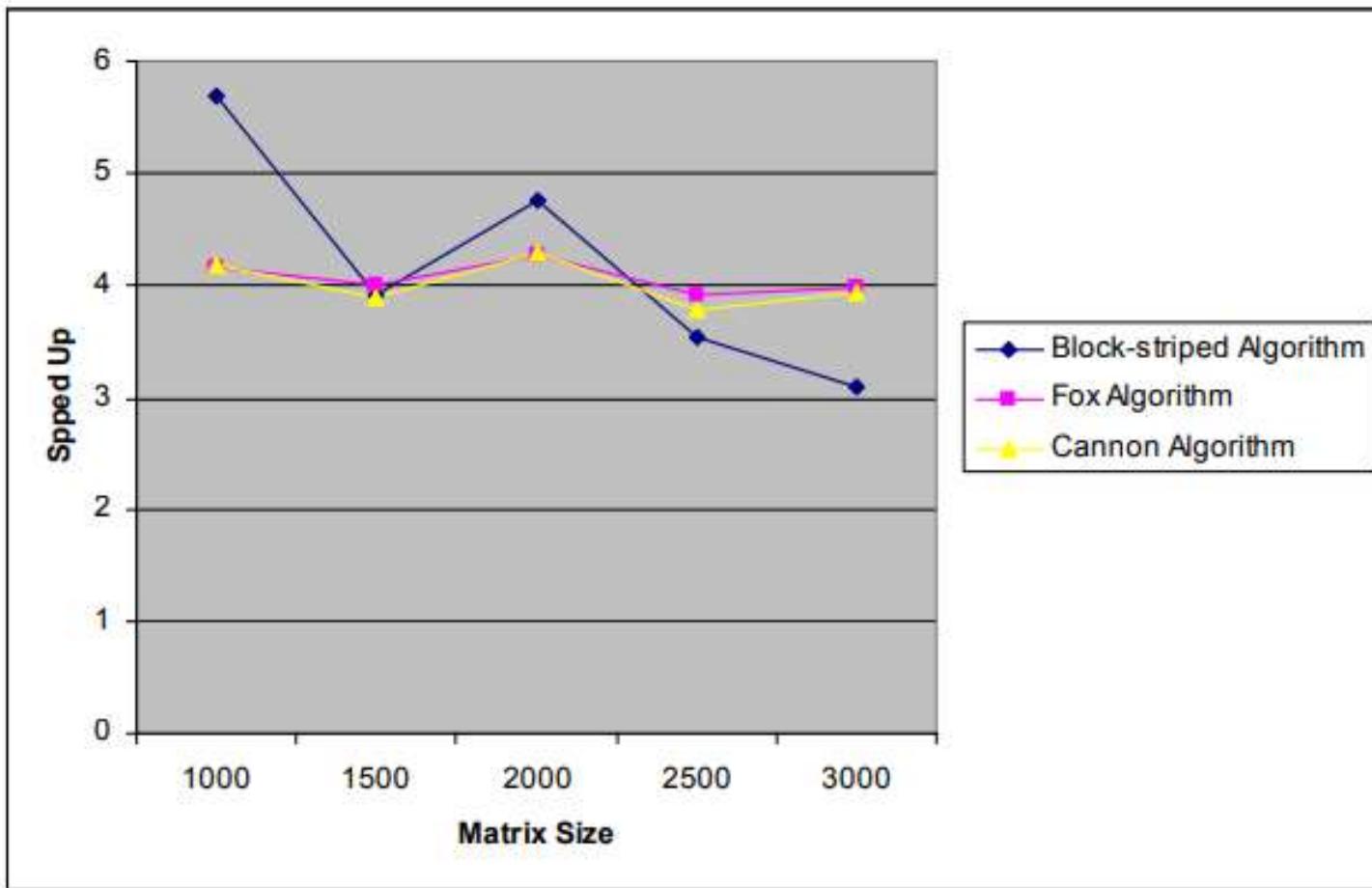
$C_{ij} + A_{ik} \cdot B_{kj} \rightarrow C_{ij}$

A_{01} B_{10} $C_{00} += A_{01} B_{10}$	A_{01} B_{11} $C_{01} += A_{01} B_{11}$
A_{10} B_{00} $C_{10} += A_{10} B_{00}$	A_{10} B_{01} $C_{11} += A_{10} B_{01}$

Результати експериментів

Matrix Size	Serial Algorithm	Parallel Algorithm			
		4 processors		9 processors	
		Time	Speed Up	Time	Speed Up
500	0,8527	0,2190	3,8925	0,1468	5,8079
1000	12,8787	3,0910	4,1664	2,1565	5,9719
1500	43,4731	10,8678	4,0001	7,2502	5,9960
2000	103,0561	24,1421	4,2687	21,4157	4,8121
2500	201,2915	51,4735	3,9105	41,2159	4,8838
3000	347,8434	87,0538	3,9957	58,2022	5,9764

Порівняння стрічкового, Фокса та Кеннона алгоритмів (4 процесори)



Використані ресурси

- Gergel V.P. Parallel Methods for Matrix Multiplication
http://www.lac.inpe.br/~stephan/CAP-372/matrixmult_microsoft.pdf

Управління потоками

Інна Вячеславівна Стеценко
професор кафедри ІПІ НТУУ «КПІ ім. Ігоря Сікорського»,
д.т.н., професор

Методи sleep(), wait() та InterruptedException

Потік, що знаходитьться в стані TIMED_WAITING, може отримати сигнал про переривання від іншого потоку, тому для цих бібліотечних методів передбачена обробка InterruptedException.

```
public void print() {
    for (int j = 0; j < 500; j++) {
        for (int count = 0; count < 50; count++) {

            System.out.print(count);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                Logger.getLogger(SymbolsynchSleep.class.getName())
                    .log(Level.SEVERE, null, ex);
            }
        }
        System.out.println();
    }
}
```

Переривання потоку

- Переривання означає термінове зупинення виконання дій потоку
- Немає можливості примусово зупинити роботу потоку (метод `stop()` заборонений для використання!). Замість цього використовується метод `interrupt()`, який запитує про можливість припинення роботи даного потоку. Виклик методу `interrupt()` змінює `interrupted status` потоку на `true`. Якщо змінювання статусу переривання відбувається для методу, що викидає `InterruptedException` (наприклад, `sleep()`, `wait()`), то управління передається обробнику винятку `InterruptedException`.
- Програмісту надається можливість вирішити, як потік має реагувати на зміну значення `interrupted status`. Виклик статичного методу `interrupted()` перевіряє чи змінений `interrupted status` потоку і водночас встановлює це значення в початкове, тобто `false`. Виклик методу `isInterrupted()` перевіряє, але не змінює значення `interrupted status`.

Методи класу java.lang.Thread

- Метод `void interrupt()` надсилає запит на переривання потоку. Статус переривання змінюється при виклику цього методу на `true`. Якщо потік в момент запиту на переривання призупинений методу `wait` чи `sleep`, то генерується виняток `InterruptedException`. Виняток очищає значення статусу переривання, тобто встановлює його у значення `false`.
- Метод `static boolean interrupted()` перевіряє, чи перерваний поточний потік (тобто потік, який виконує цю команду). Зверніть увагу, що виклик цього методу змінює значення статусу переривання поточного потоку на значення `false`.
- Метод `boolean isInterrupted()` повертає значення статусу переривання. Цей виклик не змінює статус переривання потоку.
- Метод `boolean isAlive()` повертає значення `true`, якщо потік активний. Цей метод стане `false` тільки при завершенні роботи потоку.

Приклади використання методів interrupt(), interrupted(), isInterrupted()

```
public static void testInterrupted() throws InterruptedException {  
    Thread.currentThread().interrupt();  
  
    if (Thread.interrupted()) {  
        throw new InterruptedException();  
    }  
}
```

```
public static void testIsInterrupt() {  
    Thread.currentThread().interrupt();  
    System.out.println("interrupt = "  
                      + Thread.currentThread().isInterrupted());  
}
```

Приклад

```
public class First extends Thread {  
    @Override  
    public void run() {  
        System.out.println("tic....");  
        try {  
            Thread.sleep(4000);  
        } catch (InterruptedException ex) {  
            System.out.println("InterruptedException occurred.");  
            Logger.getLogger(SymbolsSynchSleep.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        First first = new First();  
        Second second = new Second(first);  
        first.start();  
        second.start();  
    }  
}
```

```
public class Second extends Thread{  
    private Thread thread;  
    public Second(Thread t){  
        this.thread = t;  
    }  
  
    @Override  
    public void run() {  
        try {  
            Thread.sleep(1000);  
            System.out.println("isInterrupted = " +thread.isInterrupted());  
            thread.interrupt();  
            thread.join();  
            System.out.println("isInterrupted = " +thread.isInterrupted());  
        } catch (InterruptedException ex) {  
            Logger.getLogger(Second.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
}
```

Стани потоку

- Створений (NEW)
- Запущений (RUNNABLE) , може бути виконуваним та невиконуваним в даний момент часу
- Блокований (BLOCKED), очікує звільнення локера
- В очікуванні (WAITING/TIMED_WAITING), очікує або сигналу від іншого потоку або завершення часової затримки
- Завершений (TERMINATED)

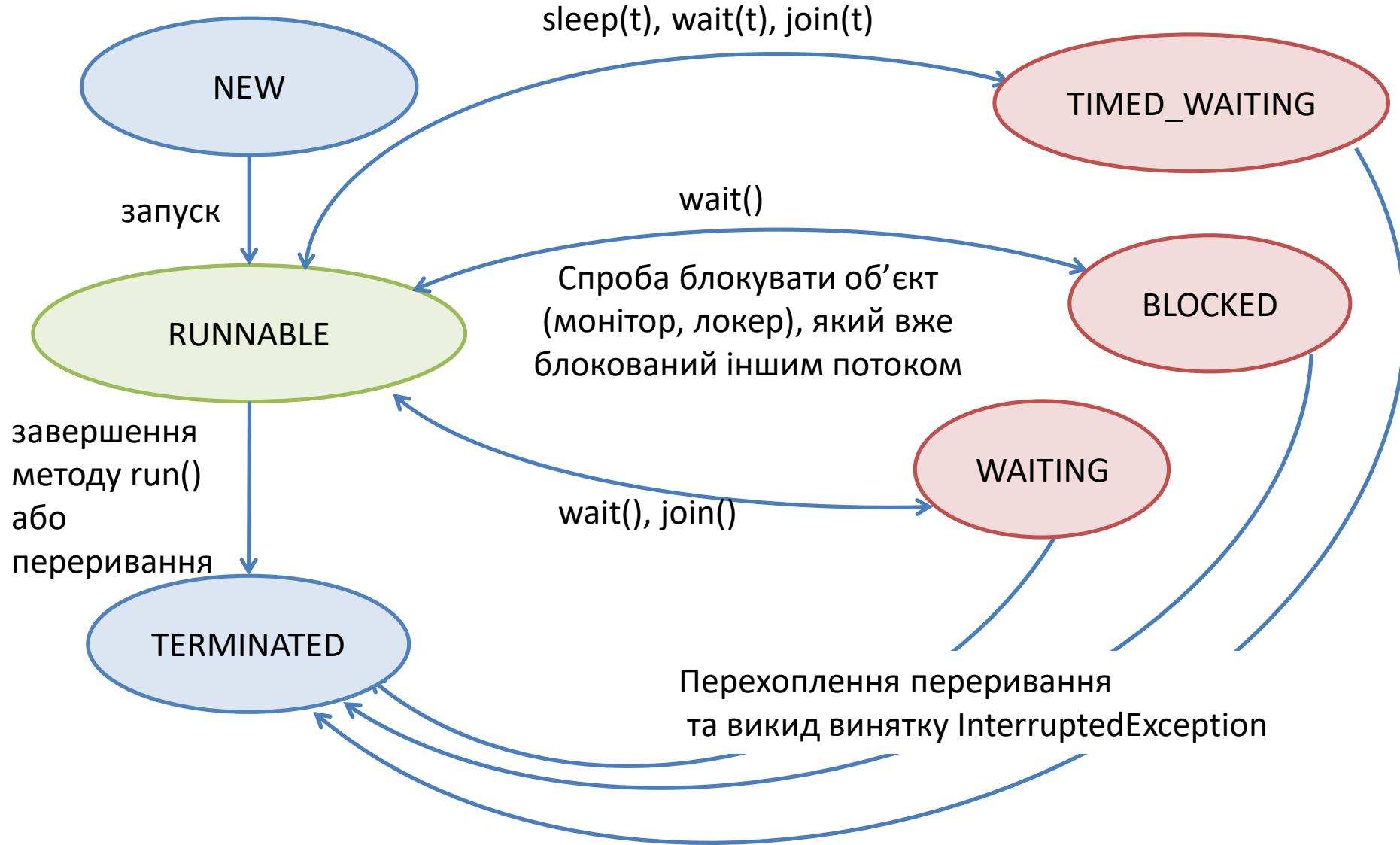
Стан потоку можна отримати викликом методу `getState()` , що повертає значення типу `Thread.State`

Тип `Thread.State` може приймати наступні значення (див документацію Java):

- NEW
- RUNNABLE
- BLOCKED
- WAITING
- TIMED_WAITING
- TERMINATED

Чи є потік запущеним (і ще не завершеним), можна дізнатись викликом методу `isAlive()`

Схема можливих змін станів потоку



Блокований потік

Потік блокований, якщо він намагається блокувати об'єкт, який вже блокований іншим потоком.

Зауваження: виконання усіх операцій введення-виведення призводить до блокування потоку. Потік продовжить виконання після завершення операції введення/виведення.

Вихід потоку з блокованого стану відбувається якщо:

- Якщо потік очікує розблокування об'єкта, заблокованого іншим потоком, то він отримає можливість продовжити роботу, коли буде зняте блокування об'єкта
- Якщо в потоці був викликаний метод `wait`, то в іншому потоці повинен бути викликаний метод `notifyAll` або `notify`.
- Якщо потік заблокований через виконання операції введення-виведення, то по завершенню цієї операції потік продовжить роботу

Потік у стані очікування

Потік у стані очікування, якщо:

- для нього викликаний метод `sleep()`
- потік викликає метод `wait()`
- потік викликає метод `join()`

Вихід зі стану очікування, якщо:

- запущений метод `sleep`, то потік продовжить виконання після того, як сплине заданий час паузи.
- в потоці був викликаний метод `wait`, то в іншому потоці повинен бути викликаний метод `notifyAll` або `notify`.
- потік очікує розблокування об'єкта, заблокованого іншим потоком, то він отримає можливість продовжити роботу, коли буде зняте блокування об'єкта.

Завершений потік

Потік завершений, якщо:

- метод `run()` нормально завершив свою роботу
- стався викид винятку `InterruptedException` при виконанні методу `run()`

Метод yield()

Метод static void yield()
призначений для того, щоб дати вказівку
планувальнику роботи потоків про те, що в
цьому місці роботи потоку виконання його дій
може бути призупинено, а ресурс може бути
переданий для виконання іншим потокам.

Зауваження:

Планувальник роботи потоків може вирішити продовжити
виконання дій в тому ж потоці, що викликав yield(). Проте
виклик методу потенційно надає можливість іншим потокам
перехопити обчислювальний ресурс.

Метод `join()`

Якщо при виконанні потоку *a* потрібно в певному місці дочекатись завершення роботи потоку *b*, то слід викликати метод `join()` для потоку *b* в цьому місці виконання потоку *a*. Іншими словами, метод `join()` примушує потік очікувати завершення роботи іншого потоку, для якого він викликаний. Потік, в методі `run()` якого був викликаний метод `join()`, зможе продовжити свою роботу тоді і тільки тоді, коли завершиться виконання потоку, що викликає метод `join()`. При цьому не розрізняється чи робота потоку, завершення роботи якого чекають, завершив роботу нормальню чи в результаті переривання. Метод підтримує викидання `InterruptedException`.

Планувальник виконання потоків

Планувальник вибирає потік серед запущених в даний момент (для потоків з вищим пріоритетом вибирає з більшою ймовірністю). Віртуальна машина Java встановлює відповідність між пріоритетами потоків та пріоритетами процесів в операційній системі.

Запущений потік виконується до тих пір, доки не виконана одна з таких дій:

- Потік виконав усі дії, що задані в його методі `run()` або стався вихід через виняток
- Потік передає управління планувальнику методом `yield()` і далі планувальник знову обирає потік, який буде запущено.
- Змінюється стан потоку, тобто запущений потік переходить в стан очікуванні, або блокований, або зупинений.
- Запущений потік з більш високим пріоритетом (або спливнув час заданої паузи, або після завершення операцій введення/виведення, або після розблокування об'єкта, який потрібен даному потоку, за допомогою методу `notifyAll` /`notify`).

Зауваження:

Якщо потоків з найвищим пріоритетом кілька, то НЕ гарантується рівноправний їх запуск.

Робота планувальника виконання потоків сильно залежить від платформи, тому результат запуску програми на різних платформах може сильно відрізнятись.

Спільне використання об'єктів

Якщо потоки виконують не унарні операції зі спільними даними, то може статись помилка. Потрібно пам'ятати, що навіть такі операції, як збільшення на 1 чи перевірка умови, складаються з трьох елементарних дій. Тому наступні фрагменти програм можуть бути такими, що не гарантують цілісність даних.

- **Read-modify-write**

```
public class UnsafeModify {  
    private long count = 0;  
    public long getCount() { return count; }  
    public void service(){ ++count; }  
}
```

- **Check-then-act**

```
public class UnsafeCheck{  
    private ExpensiveObject instance = null;  
    public ExpensiveObject getInstance() {  
        if (instance == null)  
            instance = new ExpensiveObject();  
        return instance;  
    }  
}
```

Перегони

(race condition, memory consistency error)

Якщо два потоки здійснюють доступ до одного і того ж об'єкта та викликають методи, що змінюють його стан, то вони «наступають один одному на п'яти» і, в залежності від порядку, в якому здійснюється доступ різних потоків до даних, інформація, яка зберігається в об'єкті, може бути неочікуваною.

Приклад «Банк»

```
class Bank {  
    public static final int NTEST = 10000;  
    private final int[] accounts;  
    private long ntransacts = 0;  
  
    public Bank(int n, int initialBalance) {  
        accounts = new int[n];  
        int i;  
        for (i = 0; i < accounts.length; i++) {  
            accounts[i] = initialBalance;  
        }  
        ntransacts = 0;  
    }  
  
    public void transfer(int from, int to, int amount) {  
        accounts[from] -= amount;  
        accounts[to] += amount;  
        ntransacts++;  
        if (ntransacts % NTEST == 0) {  
            test();  
        }  
    }  
  
    public void test() {  
        int sum = 0;  
        for (int i = 0; i < accounts.length; i++) {  
            sum += accounts[i];  
        }  
        System.out.println("Transactions:" + ntransacts+ " Sum: " + sum);  
    }  
}
```

Приклад «Банк»

```
class TransferThread extends Thread {  
    private Bank bank;  
    private int fromAccount;  
    private int maxAmount;  
    private static final int REPS = 1000;  
  
    public TransferThread(Bank b, int from, int max) {  
        bank = b;  
        fromAccount = from;  
        maxAmount = max;  
    }  
    @Override  
    public void run() {  
        while (true) {  
            for (int i = 0; i < REPS; i++) {  
                int toAccount = (int) (bank.size() * Math.random());  
                int amount = (int) (maxAmount * Math.random() / REPS);  
                bank.transfer(fromAccount, toAccount, amount);  
            }  
        }  
    }  
}  
}  
  
public class BankTest {  
    public static final int NACCOUNTS = 10;  
    public static final int INITIAL_BALANCE = 10000;  
  
    public static void main(String[] args) {  
        Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);  
        for (int i = 0; i < NACCOUNTS; i++) {  
            TransferThread t = new TransferThread(b, i, INITIAL_BALANCE);  
            t.setPriority(Thread.NORM_PRIORITY + i % 2);  
            t.start();  
        }  
    }  
}
```

Безпечний (immutable) об'єкт

- Об'єкт, стан якого не може бути змінений після того як він створений конструктором класу, є **безпечним** для спільногого використання потоками.
- Такий об'єкт або не має полів, або має тільки поля, значення яких не змінюється. Якщо поля примітивних типів, то їх незмінність гарантується модифікатором `final`. Якщо поля містять посилання на об'єкти, то безпечний об'єкт повинен не використовувати методи, які можуть змінити значення поля, та не передавати такі поля в інші об'єкти.

Синхронізація

Якщо потрібно вказати, що виконання методу для об'єкту не може здійснюватися кількома потоками одночасно, то використовують ключове слово **synchronized** в описі методу.

Наприклад,

```
public synchronized void transfer(  
        int from, int to, int amount) {  
    ...  
}
```

Виклик синхронізованого методу для одного й того ж об'єкту різними потоками гарантує, що усі дії методу будуть виконуватись з об'єктом від початку до завершення тільки одним потоком.

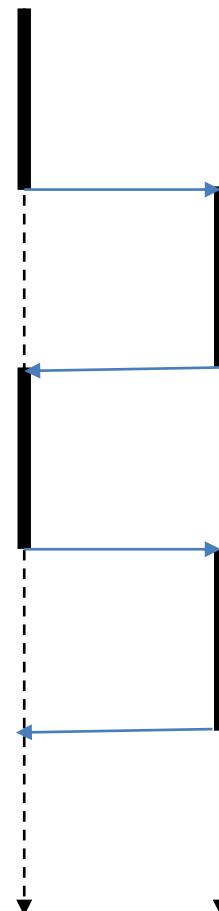
Такий підхід запропонував Тоні Хоар (Tony Hoare).

Схематичне представлення роботи синхронізованого методу

Асинхронне виконання



Синхронізоване виконання



Поняття неявного локера

- ‘intrinsic lock’ = ‘monitor’.
- Кожен об’єкт має неявний локер (монітор). Кожного разу, коли потрібно забезпечити послідовний (синхронізований) доступ до полів об’єкта, потік має зробити запит на захоплення монітора об’єкта (**блокування об’єкта**) і отримати позитивну відповідь. Після завершення синхронізованої дії з об’єктом неявний локер розблоковується.
- Якщо потік захопив монітор об’єкта, то інші потоки не можуть захопити його. При спробі захопити неявний локер об’єкта в момент, коли він заблокований іншим потоком, потік переходить у стан блокований (**блокований потік**) і очікує звільнення монітора
- Потік, який захопив локер, може захопити його повторно (властивість Reentrant). Підраховується кількість захоплень локера. Об’єкт буде розблокований, якщо кількість звільнень локера співпадає з кількістю захоплень.

Механізм Intrinsic lock (monitor)

```
Thread first = new Thread( new Runnable(){  
    public void run() {  
        counter.setName("first thread");  
        for(int j=0;j<100000000;j++){  
            counter.incMethod(1);
```

Запит на захоплення
монітора об'єкта

Неблокований об'єкт



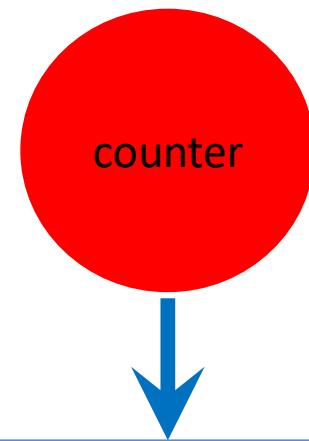
```
public synchronized void incMethod(int k){  
    c+=k;  
}
```

```
    }  
});
```

Механізм Intrinsic lock

```
Thread first = new Thread( new Runnable(){  
    public void run() {  
        counter.setName("first thread");  
        for(int j=0;j<100000000;j++){  
            counter.incMethod(1);  
        }  
    }  
});
```

Блокований об'єкт



Виконання синхронізованого методу

```
public synchronized void incMethod(int k){  
    c+=k;  
}
```

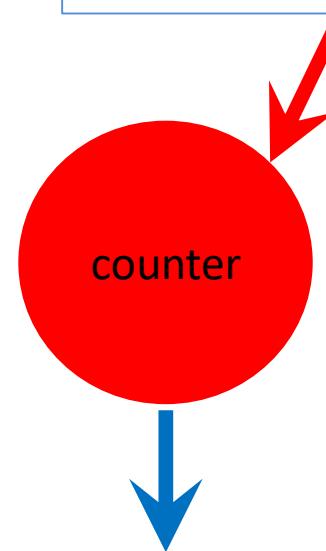
```
});
```

Механізм Intrinsic lock

```
Thread first = new Thread( new Runnable(){  
    public void run() {  
        counter.setName("first thread");  
        for(int j=0;j<100000000;j++){  
            counter.incMethod(1);  
        }  
    }  
});
```

Виконання синхронізованого методу

Запит на блокування об'єкта від іншого потоку (other)



```
public synchronized void incMethod(int k){  
    c+=k;  
}
```

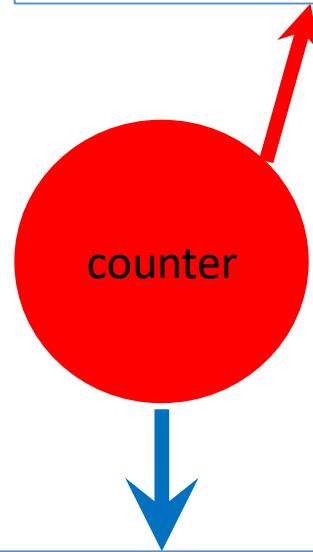
```
}
```

Механізм Intrinsic lock

```
Thread first = new Thread( new Runnable(){  
    public void run() {  
        counter.setName("first thread");  
        for(int j=0;j<100000000;j++){  
            counter.incMethod(1);  
        }  
    }  
});
```

Виконання синхронізованого методу

Потік 'other' блокований

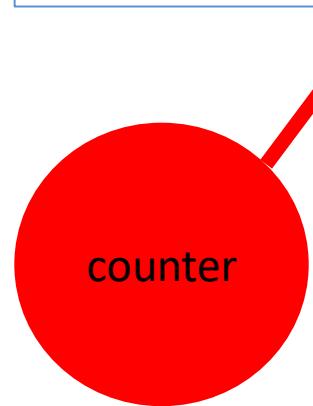


```
public synchronized void incMethod(int k){  
    c+=k;  
}
```

Механізм Intrinsic lock

```
Thread first = new Thread( new Runnable(){  
    public void run() {  
        counter.setName("first thread");  
        for(int j=0;j<100000000;j++){  
            counter.incMethod(1);  
  
        }  
    }  
});
```

Потік 'other' блокований



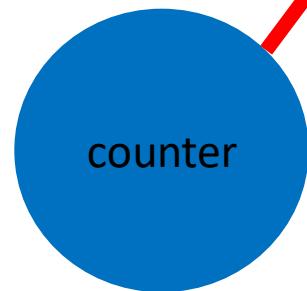
Звільнення монітора

```
public synchronized void incMethod(int k){  
    c+=k;  
}
```

Механізм Intrinsic lock

```
Thread first = new Thread( new Runnable(){  
    public void run() {  
        counter.setName("first thread");  
        for(int j=0;j<100000000;j++){  
            counter.incMethod(1);  
        }  
    }  
});
```

Потік 'other' блокований



```
public synchronized void incMethod(int k){  
    c+=k;  
}
```

Перехід на наступні за викликом
методу інструкції

Механізм Intrinsic lock

```
Thread first = new Thread( new Runnable(){
```

```
    public void run() {
```

```
        counter.setName("first thread");
```

```
        for(int j=0;j<100000000;j++){
```

```
            counter.incSyncMethod(1);
```

```
}
```

```
});
```

Потік 'other' розблокований



```
public synchronized void incMethod(int k){
```

```
    c+=k;
```

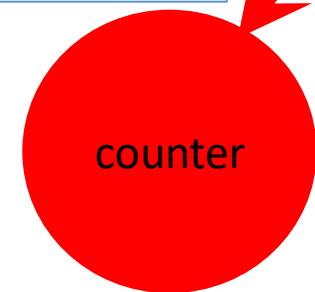
```
}
```

Виконання наступних інструкцій

Механізм Intrinsic lock

```
Thread first = new Thread( new Runnable(){  
    public void run() {  
        counter.setName("first thread");  
        for(int j=0;j<100000000;j++){  
            counter.incSyncMethod(1);  
        }  
    }  
});
```

Захоплення монітора
об'єкта іншим потоком



```
public synchronized void incMethod(int k){  
    c+=k;  
}
```

Виконання наступних інструкцій

Синхронізований блок (Synchronized Statement)

Явне використання об'єкта, який блокується на час виконання дій блоку

```
public class Counter {  
    private Object sync = new Object();  
    private int c=0;  
    private int q=0;  
    public void incBlockSync(int k) {  
        synchronized(sync) {  
            c+=k;  
        }  
        q+=k;  
    }  
}
```

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

Представлення асинхронного та синхронізованого виконання transfer() методу мережею Петрі

Блоковані об'єкти

- Об'єкт, з яким виконуються дії в синхронізованому методі, на весь час роботи методу блокується.
- Інші потоки не мають доступу до блокованого об'єкта.
- Потік, який намагається викликати синхронізований метод, перевіряє наявність блокування у об'єкта. Якщо об'єкт блокований, то потік очікує розблокування об'єкта і на весь час очікування переходить у стан Blocked.
- Планувальник роботи потоків, як тільки надходить сигнал про зміну стану об'єкта, активізує потоки, що очікують його розблокування. Тільки один з потоків зможе захопити об'єкт, який звільнився від блокування.

Виклик послідовності синхронізованих методів для спільного об'єкта не є безпечним

- Навіть якщо усі методи класу синхронізовані, це не гарантує безпечну роботу з ним. Між викликами синхронізованих методів може відбуватись робота з даними об'єкта інших потоків і результат може виявитись не таким, яким передбачав його програміст.
- Наприклад, якщо потоки A і В виконують послідовно виклик синхронізованих методів `group.deleteElement()` та `group.getList()` для спільного об'єкта `group`, то фактична послідовність дій в програмі може бути такою:

потік А виконує `group.deleteElement()`,

потік В виконує `group.deleteElement()`,

потік А виконує `group.getList()`,

потік В виконує `group.getList()`.

Очевидно, що в такому випадку результат виконання буде містити помилку. Зверніть увагу, що така послідовність може відбутись або не відбутись, оскільки має місце стохастичність захоплення обчислювального ресурсу потоками. Програміст може помітити неправильний результат роботи програми тільки в результаті ретельного тестування. Тому потрібно бути надзвичайно уважним при написанні паралельних програм.

Синхронізація сповільнює виконання програми

Не зловживайте синхронізацією, оскільки вона сповільнює виконання програми. Синхронізуйте по можливості виконання методів, які виконують зчитування спільних даних та їх оновлення. Намагайтесь зробити ці методи якомога більш «короткими» і такими, що включають тільки дії, що потребують синхронізації.

Методи `wait()` та `notify()` класу `Object` для очікування за умовою

Метод `wait()` використовують, коли потрібно призупинити виконання дій потоку до виконання певної умови. Про необхідність перевірки умови повідомляє метод `notify()`.

Виклик методу `wait()` означає, що потік переходить у стан `WAITING`, а об'єкт, для якого цей метод був викликаний, звільняється.

Потік потрапляє у список очікування (`wait list`) даного об'єкта і планувальник потоків ігнорує його, доки він не буде виведеній зі стану очікування.

Виведення зі стану очікування відбувається методом `notifyAll()` або `notify()`. Перший видаляє усі потоки зі списку очікувань, другий – тільки даний потік.

Виклик методу `notifyAll()` або `notify()` обов'язково відбувається за умови блокування об'єкта

Виконання методів `wait()` та `notify()` обов'язково має бути синхронізованим, тому виконуються тільки для блокованого об'єкта.

Приклад

```
public synchronized void transfer(int from, int to, int amount)
throws InterruptedException{
    while (accounts[from] < amount)
        wait();
    accounts[from] -= amount;
    accounts[to] += amount;
    ntransacts++;
    notifyAll() ;
    if (ntransacts % NTEST == 0)
        test();
}
```

Механізм синхронізації

- Для виклику синхронізованого методу об'єкт, для якого він викликаний, має бути незаблокованим. Об'єкт блокується після виклику синхронізованого методу. При завершенні роботи методу, об'єкт розблоковується.
- Коли потік виконує метод `wait()`, він знімає блокування з об'єкта та реєструє його в списку очікування об'єкта.
- Для видалення потоку зі списку очікування інший потік повинен викликати `notifyAll()` для цього ж об'єкта. Якщо об'єкт незаблокований, то сигнал надійде потоку, що очікував звільнення цього об'єкта. Управління буде передано в наступну після `wait()` інструкцію.

Будьте уважні!

Неправильне управління `wait()` та
`notify()` в програмі може призводити до
нескінченного очікування.

Взаємні блокування

Використання синхронізації може призводити до негативних наслідків у разі неправильного керування взаємодією потоків, а саме до взаємного блокування потоків (дедлоку).

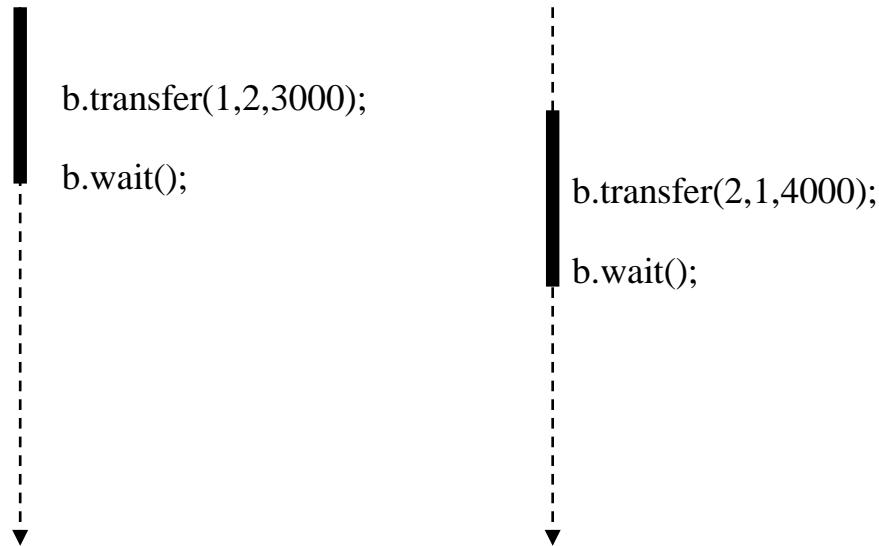
Приклад:

Потік 1

```
new TransferThread(b, 1, 2000);
```

Потік 2

```
new TransferThread(b, 2, 3000);
```



Правила, яких слід дотримуватись

- Якщо два чи більше потоків використовують методи, які модифікують (спільний) об'єкт, то ці методи повинні бути синхронізовані.
- Якщо потоку потрібно очікувати виконання певної умови, то таке очікування повинно бути організовано за допомогою методів `wait()` та `notifyAll()`. Метод `wait()` переводить потік у стан очікування, а метод `notifyAll()` подає сигнал потоку, щоб примусити його виконати наступну за `wait()` інструкцію.
- Післяожної модифікації об'єкта має бути виклик методу `notifyAll()`.
- Методи `wait()` та `notifyAll()` прописані в класі `Object`, виклик цих методів має відповідати одному ж об'єкту.

Документація методів в класі java.lang.Object

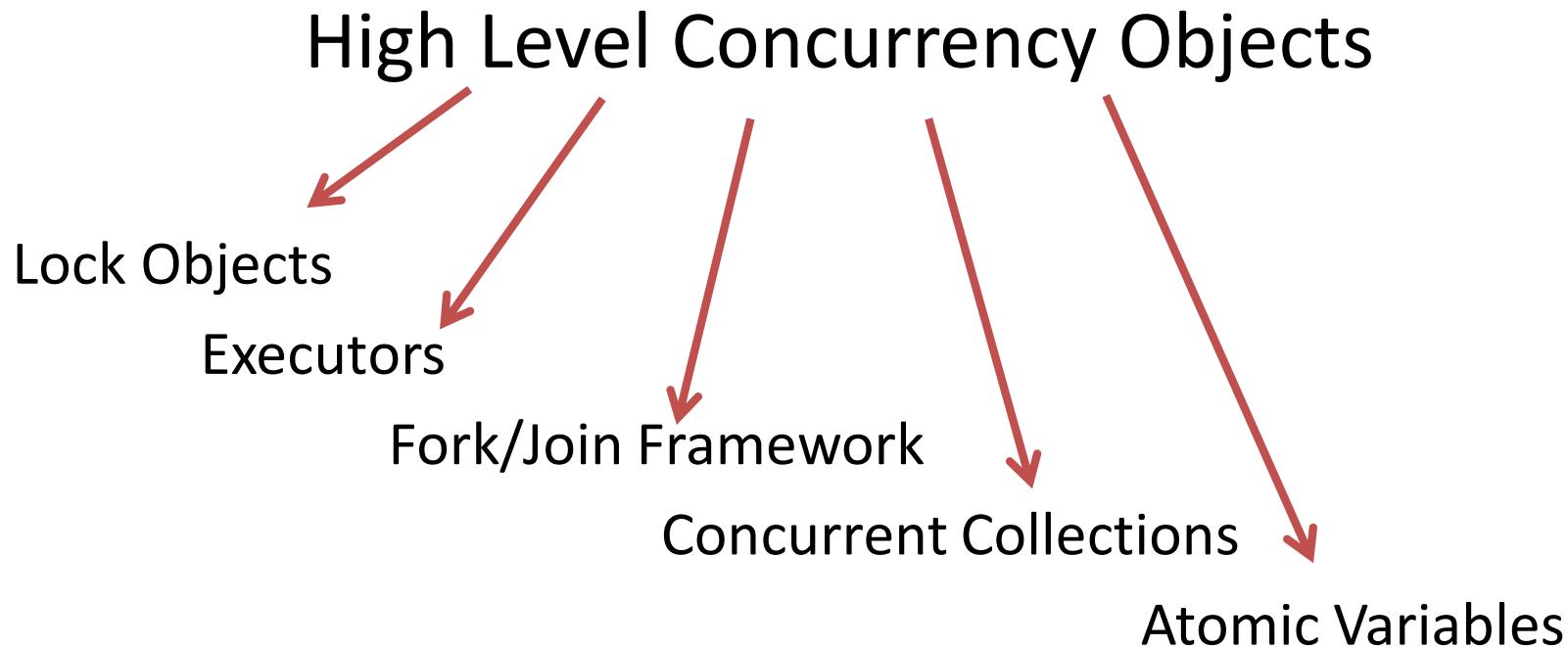
- Метод `void notifyAll()` активізує потоки, які знаходяться в очікуванні об'єкта, для якого викликається метод.
- Метод `void notify()` активізує довільно обраний потік серед тих, які викликали метод `wait` для даного об'єкта.
- Метод `void wait()` переводить потік у стан очікування, доки йому не буде надіслане повідомлення. Для методу передбачений викид `InterruptedException` у разі, якщо буде переривання очікування.
- Усі три методи можна викликати тільки в межах синхронізованого методу чи блоку. Викид винятку `IllegalMonitorStateException` станеться, якщо виклик методу здійснюється не в синхронізованому блоці. Виняток сигналізує, що потік намагається захопити монітор, власником якого не є.

Використані джерела

- 1. Документація Java™ Platform, Standard Edition 8
<https://docs.oracle.com/javase/8/docs/api/overview-summary.html>
- 2. Cay Horstmann Core Java Volume I--Fundamentals (Core Series) 11th Edition

Високорівневі засоби паралельного програмування в технології java

Високорівневі засоби управління потоками



Інтерфейс Lock

Методи інтерфейсу Lock:

`lock()` – блокує об'єкт, якщо той незаблокований іншим потоком, або встановлює потік у стан заблокований і примушує його очікувати звільнення локера (може повторно блокувати об'єкт, якщо локер був захоплений тим самим потоком)

`tryLock()` – блокує об'єкт тільки, якщо він в стані незаблокований, і надає інформацію про успішність блокування (таку інформацію можна потім обробити в програмі)

`unlock()` – розблоковує об'єкт, якщо він був блокований

`newCondition()` – створює екземпляр класу `Condition`, який прикріплюється до даного `Lock`-екземпляру і використовується методами `await()`, `signal()` для організації очікування за умовою.

Example:

```
Lock lock = ...;
lock.lock(); //блокування
try {
    // доступ до ресурсу, захищеного цим блоком
}
finally {
    lock.unlock(); // розблокування
}
```

Клас ReentrantLock

- Підтримує інтерфейс Lock
- Об'єкт класу ReentrantLock привласнюється потоком у разі успішного блокування і належить йому до розблокування. Блокування об'єкту відбувається успішно, якщо на момент запиту на блокування він не привласнений іншим потоком.
- Потік, який є власником об'єкта, може повторно заблокувати його необмежену кількість разів. Лічильник блокувань відслідковує, щоб кількість блокувань дорівнювала кількості розблокувань.
- З використанням методів isHeldByCurrentThread(), getHoldCount() можна перевірити чи є даний екземпляр класу власником потоку
- Додатковий конструктор класу ReentrantLock(boolean fair) містить аргумент fair. Значення fair, що дорівнює true, встановлює, що можливість блокування об'єкта буде надаватись у першу чергу потокам, які мають найбільший час його очікування. Така «справедливість» коштує додаткових витрат часу на планування, проте запобігає «голодуванню» потоків тобто надто довгому їх очікуванню.

Example:

```
private final ReentrantLock lock = new ReentrantLock();
public void method() {
    lock.lock();
    try { // завжди використовуйте блок try/finally, щоб гарантувати виклик unlock()
        // ... method body
    }
    finally {
        lock.unlock()
    }
}
```

Reentrance

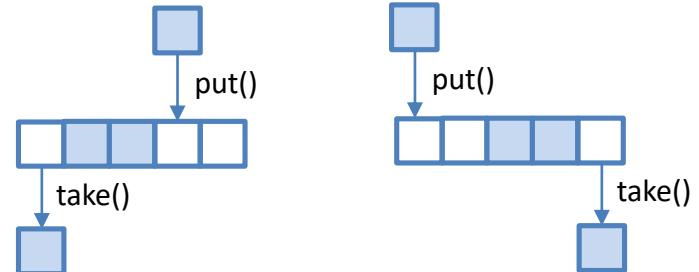
```
public class ReentranceExample{  
    public synchronized f() {  
        g(); // виклик можливий, оскільки монітор належить потоку  
    }  
    public synchronized g() {  
        ... // код методу  
    }  
}
```

Інтерфейс Condition

Призначений для створення множинних наборів очікування для одного об'єкта.

Основні методи `await()`, `signal()`, `signalAll()`

```
class BoundedBuffer {  
    final Lock lock = new ReentrantLock();  
    final Condition notFull = lock.newCondition();  
    final Condition notEmpty = lock.newCondition();  
  
    final Object[] items = new Object[100];  
    int putptr, takeptr, count;  
  
    public void put(Object x)  
        throws InterruptedException{  
        lock.lock();  
        try {  
            while (count == items.length)  
                notFull.await();  
            items[putptr] = x;  
            if (++putptr == items.length)  
                putptr = 0;  
            ++count;  
            notEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }
```



```
public Object take()  
    throws InterruptedException {  
    lock.lock();  
    try {  
        while (count == 0)  
            notEmpty.await();  
        Object x = items[takeptr];  
        if (++takeptr == items.length)  
            takeptr = 0;  
        --count;  
        notFull.signal();  
        return x;  
    } finally {  
        lock.unlock();  
    }  
}
```

Concurrent Collections

- Надають можливість уникнути проблеми Memory Consistency Errors
- BlockingQueue, ConcurrentMap та інші
- Наприклад, використання BlockingQueue в наведеному вище прикладі дозволяє відмовитись від синхронізації методів take(), put().

```
public class Producer implements Runnable {  
    private BlockingQueue<String> drop;  
  
    public Producer(BlockingQueue<String> drop) {  
        this.drop = drop;  
    }  
  
    public void run() {  
        String importantInfo[] = {  
            "Mares eat oats",  
            "Does eat oats",  
            "Little lambs eat ivy",  
            "A kid will eat ivy too"  
        };  
        Random random = new Random();  
  
        try {  
            for (int i = 0;  
                i < importantInfo.length;  
                i++) {  
                drop.put(importantInfo[i]);  
                Thread.sleep(random.nextInt(5000));  
            }  
            drop.put("DONE");  
        } catch (InterruptedException e) {}  
    }  
}
```

```
public class Consumer implements Runnable {  
    private BlockingQueue<String> drop;  
  
    public Consumer(BlockingQueue<String> drop) {  
        this.drop = drop;  
    }  
  
    public void run() {  
        Random random = new Random();  
        try {  
            for (String message = drop.take();  
                ! message.equals("DONE");  
                message = drop.take()) {  
                System.out.format("MESSAGE RECEIVED: %s%n",  
                    message);  
                Thread.sleep(random.nextInt(5000));  
            }  
        } catch (InterruptedException e) {}  
    }  
  
    public class ProducerConsumerExample {  
        public static void main(String[] args) {  
            BlockingQueue<String> drop =  
                new SynchronousQueue<String>();  
            (new Thread(new Producer(drop))).start();  
            (new Thread(new Consumer(drop))).start();  
        }  
    }  
}
```

Interface BlockingQueue

- Інтерфейс успадковує інтерфейс Queue і методи цього інтерфейсу element(), peek(), poll(), remove(), а також успадковує інтерфейс Collection
- Інтерфейс визначає типи для черг, що забезпечують безпечну (синхронізовану) їх роботу
- Інтерфейс містить
 - методи put(), take(), що передбачають очікування у разі, якщо виконати операцію неможливо
 - методи offer(), poll(), що передбачають спробу виконати операцію та повертають булеве значення true у разі успішності виконання та false у противному випадку
 - методи add(), remove(), що передбачають виконання операції, якщо можливо виконати її. Метод add() викидає IllegalStateException, якщо черга переповнена
- Інтерфейс реалізують такі класи
 - LinkedBlockingQueue,
 - LinkedBlockingDeque,
 - ArrayBlockingQueue,
 - SynchronousQueue
 - DelayQueue,
 - LinkedTransferQueue,
 - PriorityBlockingQueue,
- Наведені класи відрізняються правилами вставки та вилучення з черги, за допомогою конструкторів можна створювати обмежену чи необмежену чергу. Передивітесь документацію цих класів!

Атомарні (Atomic) операції

- Операції read/write є атомарними для змінних посилальних типів та більшості примітивних типів (усіх окрім long та double).
- Операції read/write є атомарними для усіх змінних, декларованих з ключовим словом **volatile** (включно з long та double змінними).
- Атомарні операції не можуть перериватись іншими потоками, тому такі операції є безпечними.
- **volatile** гарантує пріоритет операції write (по відношенню до операції read)

Атомарні (Atomic) змінні

- Належать класам, які забезпечують атомарні операції зі змінними
- Наприклад, класи `AtomicBoolean`, `AtomicInteger`, `AtomicLong`, `AtomicReference<V>`, `AtomicIntegerArray`,....

```
class AtomicCounter {  
    private AtomicInteger c = new AtomicInteger(0);  
  
    public void increment() {  
        c.incrementAndGet(); // спеціальні методи класу для арифметичних операцій  
    }  
  
    public void decrement() {  
        c.decrementAndGet();  
    }  
  
    public int value() {  
        return c.get();  
    }  
}
```

Незмінні (immutable) об'єкти

- Об'єкт, стан яких не може бути змінений в ході виконання програми, називають незмінними (immutable)
- Якщо об'єкт не має полів або має поля виключно примітивних типів, позначених модифікаторами `private final`, то він є незмінним (immutable).
- Такі змінні є безпечними при використанні в потоках, оскільки не потребують синхронізації
- !!! Якщо об'єкт має поля, які є посиланнями на об'єкти, то `private final` гарантує незмінюваність посилання на об'єкт, але не незмінюваність значень, які містяться за посиланням. Для забезпечення незмінюваності об'єкта для таких полів не мають використовуватись методи, які можуть змінити їх стан, а також такі поля не можна передавати в інші потоки.

Використані ресурси

- **The Java™ Tutorials: Concurrency**

Негативні ефекти застосування управління потоками

Види негативних наслідків управління потоками

Синхронізація виконання потоків є засобом для запобігання помилок Memory Consistency Errors (або Race Condition).

Взаємне блокування потоків (Deadlock) є наслідком застосування синхронізації виконання потоків. Потоки очікують сигналу для припинення очікування один і від одного і через це не можуть продовжити свою роботу.

Голодування потоків (starvation) виникає, коли через надмірну активність одних потоків інші не можуть отримати обчислювальні ресурси для виконання своїх дій.

Блокування активністю (livelock) виникає, коли потоки виконують свої дії, проте це не призводить до завершення роботи програми. Наприклад, дії одного потоку спричиняють дії іншого, які повертають перший потік на попередні дії, і так до нескінченності.

Взаємне блокування потоків (Deadlock)

```
public class Deadlock {  
    static class Friend {  
        private final String name;  
  
        public Friend(String name) { this.name = name; }  
        public String getName() { return this.name; }  
  
        public synchronized void bow(Friend other) {  
            System.out.format("%s: %s" + " has bowed to me!%n", name, other.getName());  
            other.bowBack(this);  
        }  
        public synchronized void bowBack(Friend bower) {  
            System.out.format("%s: %s" + " has bowed back to me!%n", name, other.getName());  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    final Friend a = new Friend("A");  
    final Friend b = new Friend("B");  
    new Thread(new Runnable() {  
        public void run() { a.bow(b); }  
    }).start();  
    new Thread(new Runnable() {  
        public void run() { b.bow(a); }  
    }).start();  
}
```

Взаємне блокування потоків (Deadlock)

Що відбувається:

При запуску синхронізованого методу `a.bow(b)` потік стає власником монітора об'єкта `a`. При виконанні методу `b.bowBack(a)` робить запит на захоплення монітора об'єкта `b`.

Одночасно з першим потоком, другий потік здійснює запуск синхронізованого методу `b.bow(a)` і стає власником монітора об'єкта `b`. Саме тому перший потік наштовхується на вже зайнятий іншим (другим) потоком монітор об'єкта `b` і переходить у стан `BLOCKED`. Далі, другий потік при виконанні методу `a.bowBack(b)` робить запит на захоплення монітора об'єкта `a` і наштовхується на вже зайнятий іншим (першим) потоком монітор об'єкта `a`. Тому він теж переходить у стан `BLOCKED`.

Оскільки обидва потоки у стані `BLOCKED`, тобто очікують розблокування об'єкта (перший – об'єкта `b`, а другий - об'єкта `a`), то розблокування жодного з об'єктів відбутись не може і обидва будуть очікувати нескінченно довго. Програма працює успішно, але нічого окрім очікування не відбувається!

Як запобігти дедлоку?

Реалізація прикладу «Взаємне вітання друзів» БЕЗ deadlock

```
public class LockFriends {  
    static class Friend {  
        private String name;  
        private Lock lock = new ReentrantLock();  
  
        public Friend(String name) { this.name = name; }  
        public String getName() { return name; }  
  
        public boolean impendingBow(Friend bower) {  
            Boolean myLock = false;  
            Boolean yourLock = false;  
            try {  
                myLock = lock.tryLock(); //true if lock was acquired  
                yourLock = bower.lock.tryLock(); //true if lock was acquired  
            } finally {  
                if (!(myLock && yourLock)) {  
                    if (myLock) {  
                        lock.unlock(); // release this lock  
                    }  
                    if (yourLock) {  
                        bower.lock.unlock();  
                    }  
                }  
            }  
            return myLock && yourLock;  
        }  
    }  
}
```

Реалізація прикладу «взаємне вітання друзів» БЕЗ deadlock

```
public void bow(Friend bower) {  
    if (impendingBow(bower)) {  
        try { System.out.format("%s: %s" + " has bowed to me!%n",  
                               this.name, bower.getName());  
              bower.bowBack(this);  
        } finally {  
            lock.unlock();  
            bower.lock.unlock();  
        }  
    } else {  
        System.out.format("%s: %s started " + " to bow to me,  
                          but saw that I was already bowing to him.%n",  
                          this.name, bower.getName());  
    }  
}  
  
public void bowBack(Friend bower) {  
    System.out.format("%s: %s" + " has bowed back to me!%n",  
                      this.name, bower.getName());  
}  
} // кінець класу LockFriends
```

Реалізація прикладу

«Взаємне вітання друзів» БЕЗ deadlock

```
static class BowLoop implements Runnable{
    private Friend a;
    private Friend b;
    public BowLoop(Friend bower, Friend bowee) {
        a = bower;
        b = bowee;
    }
    @Override
    public void run() {
        Random random = new Random();
        for(;;) {
            try {
                Thread.sleep(random.nextInt(1));
            } catch (InterruptedException ex) {
                //... обробка InterruptedException
            }
            b.bow(a);
        }
    }
}
public static void main(String[] args) {
    final Friend a = new Friend("A");
    final Friend b = new Friend("B");
    new Thread(new BowLoop(a,b)).start();
    new Thread(new BowLoop(b,a)).start();
}
```

Виокристані ресурси

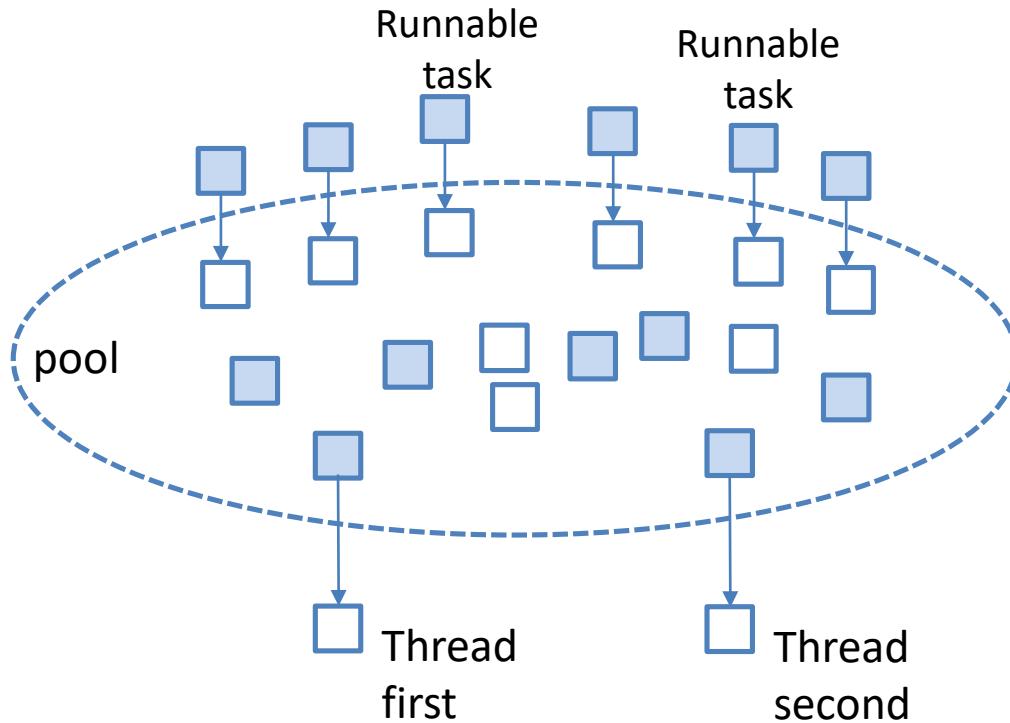
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html>

Високорівневі засоби
багатопочного програмування
java. Пули потоків. Інтерфейс
Executor.

Пул потоків

- Щоб зменшити накладні витрати при створенні великої кількості потоків у застосунках, використовують пули потоків
- Пул потоків (Thread Pool) – це сукупність потоків, призначена для обслуговування “задач” (tasks)
- Задача (Task) – об’єкт типу Runnable, призначений для виконання дій окремої невеличкої за обсягом підзадачі.

Пул потоків



Розробка пулу потоків: основні кроки

- Створити задачі
 - тип Runnable / Callable
- Створити пул потоків
 - тип, що підтримує інтерфейси Executor, ExecutorService:
ForkJoinPool, ScheduledThreadPoolExecutor,
ThreadPoolExecutor
 - фабрики пулів потоків є в класі Executors
- Завантажити задачі в пул
 - методи інтерфейсів Executor, ExecutorService:
execute(), submit(), invokeAll(), invokeAny()
- Завершити роботу пулу потоків
 - методи інтерфейсу ExecutorService:
shutdown(), shutdownNow(), awaitTermination()
- Отримати результат виконання потоків
 - методи інтерфейсу Future:
get(), get(long timeout, TimeUnit unit), isDone(),
cancel(), isCancelled()

Клас Executors

Клас Executors містить декілька методів-фабрик для створення пулів потоків.

- Методи створення пулу потоків:
`Executors.newFixedThreadPool()`,
`Executors.newCachedThreadPool()`,
`Executors.newSingleThreadExecutor()`
- «Фабрики» пулів потоків містяться також в класах
`ThreadPoolExecutor` та
`ScheduledThreadPoolExecutor`

Інтерфейс Executor

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html>

Містить єдиний метод void execute(Runnable task) для завантаження задач, які мають бути виконані:

```
executor.execute(new RunnableTask1());  
executor.execute(new RunnableTask2());
```

Приклади реалізації методу execute(Runnable task):

```
class DirectExecutor implements Executor {  
    public void execute(Runnable task) {  
        task.run();  
    }  
}  
  
class ThreadPerTaskExecutor implements Executor {  
    public void execute(Runnable task) {  
        new Thread(task).start();  
        // формально можливо, але не рекомендується документацією Java  
        // новий потік не створюється, вирішує виконавець,  
        // якому потоку доручити виконання задачі  
    }  
}
```

Приклад визначення порядку виконання задач

```
class SerialExecutor implements Executor {  
    final Queue<Runnable> tasks = new ArrayDeque<Runnable>();  
    final Executor executor;  
    Runnable active;  
    SerialExecutor(Executor executor) {  
        this.executor = executor;  
    }  
    public synchronized void execute(final Runnable r) {  
        tasks.offer(new Runnable() {  
            public void run() {  
                try {  
                    r.run();  
                } finally {  
                    scheduleNext();  
                }  
            }  
        });  
        if (active == null) {  
            scheduleNext();  
        }  
    }  
}
```

```
protected synchronized void scheduleNext() {  
    if ((active = tasks.poll()) != null) {  
        executor.execute(active);  
    }  
}
```

Інтерфейс ExecutorService extends Executor

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html>

Окрім методу execute() цей інтерфейс містить такі методи для завантаження задач:

Future<?> **submit**(Runnable task) – завантажує задачу в пул, метод get об'єкту Future повертає null у разі успішного виконання задачі де Future – спеціальний тип для повернення результату виконання run() методу

<T> Future<T> **submit**(Runnable task, T result) – завантажує задачу в пул, метод get об'єкту Future повертає заданий в аргументі result у разі успішного виконання задачі

<T> Future<T> **submit**(Callable<T> task)

<T> List<Future<T>> **invokeAll**(Collection<? extends Callable<T>> tasks) – завантаження колекції задач, результати виконання задач повертаються, коли виконання задач завершено (успішно або через exception)

<T> T **invokeAny**(Collection<? extends Callable<T>> tasks) – завантаження колекції задач, результат повертається, коли завершено виконання однієї з задач, виконання інших при цьому відміняється.

Інтерфейс ExecutorService extends Executor

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html>

Методи для припинення завантаження задач:

void **shutdown()** – припинення завантаження нових задач: завантажені раніше задачі продовжують виконання, а при спробі завантажити нову задачу в пул буде викид RejectedExecutionException
List<Runnable> **shutdownNow()** – намагається зупинити усі задачі, які виконуються (проте без гарантії успішної зупинки), та усі, які очікують виконання, повертає список задач, які очікують виконання (при цьому завантажені раніше задачі, які не вдалось зупинити, продовжують виконання)

Інтерфейс ExecutorService extends Executor

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html>

Методи для очікування завершення виконання усіх задач:

boolean **awaitTermination**(long time, TimeUnit unit) – блокування пулу доки: або завершиться виконання усіх його задач після виклику shutdown(), або сплине часова затримка, або дія ‘current thread’ перервана викликом interrupt(). Метод повертає значення true у разі успішного завершення роботи усіх потоків з пулу, false – у противному випадку, і викидає InterruptedException у разі отримання інформації про зміну статусу переривання.

Документація Java рекомендує для гарантованого завершення роботи пулу потоків такий фрагмент:

```
pool.shutdown(); //заборонити надходження нових задач
try {
    // дати затримку на завершення роботи тим потокам, які раніше завантажені
    if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
        // припинити роботу потоків, що виконуються
        pool.shutdownNow();
        //дати затримку на очікування відповіді про успішність завершення роботи потоків
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("Pool did not terminate");
    }
} catch (InterruptedException e) {
    // цей фрагмент на випадок якщо робота пулу потоків була перервана іншим
    pool.shutdownNow();
    Thread.currentThread().interrupt();
}
```

Інтерфейс ExecutorService extends Executor

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html>

Містить методи submit (Runnable task), submit(Callable<T> task) для завантаження задач та методи shutdown(), shutdownNow() для припинення завантаження задач

```
class NetworkService implements Runnable {  
    private final ServerSocket serverSocket;  
    private final ExecutorService pool;  
  
    public NetworkService(int port, int poolSize) throws IOException {  
        serverSocket = new ServerSocket(port);  
        pool = Executors.newFixedThreadPool(poolSize);  
    }  
  
    public void run() { // run the service  
        try {  
            for (;;) {  
                pool.execute(  
                    new Handler(serverSocket.accept()));  
            }  
        } catch (IOException ex) {  
            pool.shutdown();  
        }  
    }  
}
```

```
class Handler implements Runnable {  
    private final Socket socket;  
  
    Handler(Socket socket) {  
        this.socket = socket;  
    }  
  
    public void run() {  
        // read and service request on socket  
    }  
}
```

Інтерфейс Future<V>

Містить методи для отримання значення результату виконання задач, що виконуються асинхронно

V get() – отримати значення результату виконання задачі

V get(long time, TimeUnit unit) – отримати значення результату виконання задачі або завершити передчасно виконання, якщо час time вичерпаний

boolean isDone() – повертає true, якщо виконання задачі завершене успішно

```
ExecutorService pool = Executors.newSingleThreadExecutor();
Future<Integer> future = pool.submit(task);
System.out.println("future done? " + future.isDone());

Integer result = future.get(); // очікування завершення обчислень задачі

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

Інтерфейс Future<V>

```
ExecutorService executor = Executors.newSingleThreadExecutor();

Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(2); //задача виконується 2 секунди
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
}) ;

future.get(1, TimeUnit.SECONDS);
                    //очікувати результат не більше 1 секунди
```

Інтерфейс Callable

Callable задача, на відміну від Runnable, може повертати результат обчислень

```
Callable task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    } catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
}
```

Метод submit() інтерфейсу ExecutorService повертає об'єкт Future і може бути використаний для отримання результату виконання Callable задачі

Інтерфейс ScheduledExecutorService extends ExecutorService

Містить методи schedule(...), scheduleAtFixedRate(...),
scheduleWithFixedDelay(...) для запуску задач із заданою затримкою.
Всі методи повертають змінні спеціального типу ScheduledFuture<V>

```
public class ScheduleTest {  
  
    public static void main(String[] args) {  
        ScheduledExecutorService scheduler  
            = Executors.newScheduledThreadPool(1);  
  
        final Runnable task1 = () -> {  
            System.out.println("task1");  
        };  
        final Runnable task2 = () -> {  
            System.out.println("task2");  
        };  
  
        scheduler.schedule(task1, 2, TimeUnit.SECONDS);  
        scheduler.schedule(task2, 4, TimeUnit.SECONDS);  
        scheduler.shutdown();  
    }  
}
```

Приклад: виведення символів

```
public static void testExecutor() {
    Runnable task = new Symbol('|'); // run-method do 1000 times print of a symbol
    Runnable taskOther = new Symbol('-');
    ExecutorService executor = Executors.newFixedThreadPool(2);

    executor.execute(task);
    executor.execute(taskOther);
    executor.shutdown();
}
```

```
public static void testExecutor() {
    Runnable task = new Symbol('|');
    Runnable taskOther = new Symbol('-');
    Future<String> res;
    Future<?> result;

    ExecutorService executor = Executors.newFixedThreadPool(2);

    res = executor.submit(task, "done"); // submits with 2 arguments
    result = executor.submit(taskOther); // submits with 1 argument
    executor.shutdown();
    try {
        System.out.println("\n"+res.get()+" "+result.get());
    } catch (InterruptedException | ExecutionException ex) {
        // print exception info
    }
}
```

Приклад: виведення символів

```
public static void testExecutorControl() {  
    Runnable task = new Symbol('|');  
    Runnable taskOther = new Symbol('-');  
  
    ExecutorService executor = Executors.newFixedThreadPool(2);  
    ExecutorControl executorControl = new ExecutorControl(executor);  
  
    executorControl.execute(task);  
    executorControl.execute(taskOther);  
  
    executor.shutdown();  
}
```

Методи invokeAll(), invokeAny

```
ExecutorService executor = Executors.newWorkStealingPool(); // (!) створює ForkJoinPool
List<Callable<String>> callables = Arrays.asList(() -> "task1",
                                                       () -> "task2",   () -> "task3");
executor.invokeAll(callables).stream().map(future -> {
    try {
        return future.get();
    }
    catch (Exception e) {
        throw new IllegalStateException(e);
    }
}).forEach(System.out::println);
```

```
Callable callable(String result, long seconds) {
    return () -> {
        TimeUnit.SECONDS.sleep(seconds);
        return result;
    };
}
ExecutorService executor = Executors.newWorkStealingPool();
List<Callable<String>> callables = Arrays.asList(callable("task1", 2),
                                                       callable("task2", 1), callable("task3", 3));
String result = executor.invokeAny(callables);
System.out.println(result);
```

Приклад: множення матриць

```
public class TaskMultiple implements Callable<Matrix> {
    private Matrix one;
    private Matrix another;

    public TaskMultiple(Matrix one, Matrix another) {
        this.one = one;
        this.another = another;
    }

    @Override
    public Matrix call() throws Exception {
        return MatrixMathematics.multiply(one, another);
    }
}
```

Приклад: множення матриць

```
public class ThreadPoolMatrixMultiplication {  
    public static void main(String[] args) throws InterruptedException,  
                                         ExecutionException {  
        Matrix a = new Matrix(1000,1000);  
        Matrix b = new Matrix(1000,1000);  
        a.setAllValues(1.0);  
        b.setAllValues(1.0);  
        concurrencyMultiple(a,b,100).print();  
    }  
    public static Matrix concurrencyMultiple(Matrix a, Matrix b, int numTasks)  
                                         throws InterruptedException, ExecutionException {  
        Matrix c = new Matrix(a.getNrows(),b.getNcols());  
        int cols = a.getNcols();  
        int rows = a.getNrows();  
        int r = rows/numTasks;  
        ExecutorService executor = Executors.newFixedThreadPool(2);  
        List<TaskMultiply> tasks = new ArrayList<>();  
        for(int j=0;j<numTasks; j++){  
            tasks.add(j,new TaskMultiply(a.getFragment(r*j, r*j+r-1, 0, cols-1),b));  
        }  
        List<Future<Matrix>> result = executor.invokeAll(tasks);  
        executor.shutdown();  
        for (int j=0; j<numTasks; j++)  
            c.setFragment(result.get(num).get(), r*num, r*num+r-1, 0, cols-1);  
        return c;  
    }  
}
```

Приклад: ScheduledFuture and ScheduledExecutorService

ScheduledExecutorService може запускати задачі з заданим інтервалом.
Повертає тип ScheduledFuture

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> {
    System.out.println("Scheduling: " + System.nanoTime());
};

ScheduledFuture<?> future =
    executor.schedule(task, 7, TimeUnit.SECONDS);
try {
    Thread.sleep(1000);
} catch (InterruptedException ex) {
    // обробка винятку
}
System.out.printf("Remaining Delay: %s s\n",
    future.getDelay(TimeUnit.SECONDS));

executor.shutdown();
```

Синхронізований доступ в сервісах-виконавцях

Синхронізовані методи використовуються для задач так само, як і в потоках

ReentrantLock

ReentrantReadWriteLock

StampedLock

Semaphore

ForkJoin Framework

ForkJoin Framework

- Розміщує задачі в пул потоків ForkJoinPool
`extends AbstractExecutorService`
- Забезпечує більшу швидкість за рахунок використання work-stealing алгоритму: потоки, які вільні, можуть здійснювати «крадіжку» роботи у потоків, які є зайнятими.
- Розроблений для алгоритмів, в яких підзадачі створюються рекурсивно
- Основні класи фреймворку
 - `ForkJoinPool` (`implements ExecutorService`) ,
 - `ForkJoinTask` (`implements Future`) , `RecursiveTask` (`implements Future`) ,
 - `RecursiveAction` (`implements Future`)

- ! Алгоритми ForkJoin фреймворку використовуються класом Arrays для паралельного сортування об'єктів:
`Arrays.parallelSort()`
- ! Алгоритми ForkJoin фреймворку використовуються `java.util.streams` для паралельної обробки масивів

Етапи розробки паралельного алгоритму

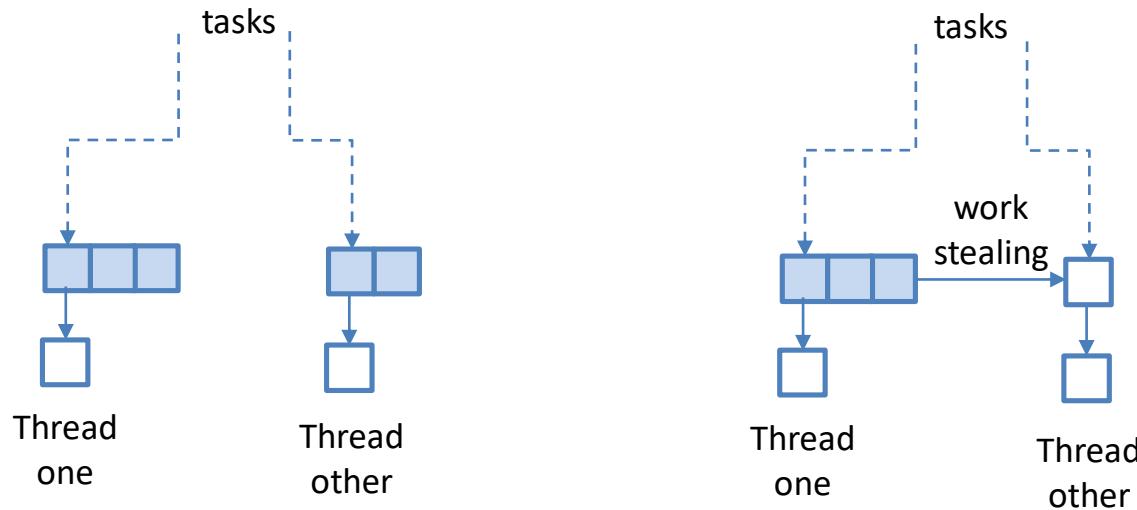
- Задача поділяється на підзадачі, які можуть виконуватись незалежно. Підзадачі можуть розроблятись як такі, що створюють нові підзадачі.
- Для підзадач розробляється (-ються) ForkJoinTask клас (-и)
- Підзадачі (-а) додаються в пул потоків і пул запускається на виконання
- З розв'язків підзадач складається розв'язок задачі

Технологія «крадіжки задачі»

Кожний потік («воркер») пулу потоків типу ForkJoinPool може виконувати кілька задач одночасно. Кількість воркерів, за замовчуванням, дорівнює кількості доступних процесорів в обчислювальній системі.

Кожний воркер має двосторонню чергу ForkJoinTask-задач. При надходженні задачі ставлять за правилом LIFO.

Пришвидшення відбувається за рахунок технології «крадіжки роботи»: воркер, який має порожню чергу задач, бере на виконання задачу з іншого потоку за правилом FIFO.



Застосування класів ForkJoinFramework

- Задачі створюються як такі, що успадковують клас ForkJoinTask
- Виконавці створюються як пул потоків з використанням ForkJoinPool, який підтримує інтерфейс ExecutorService.

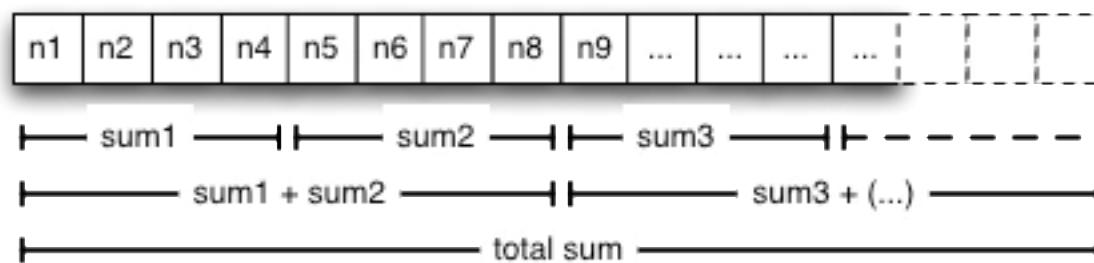
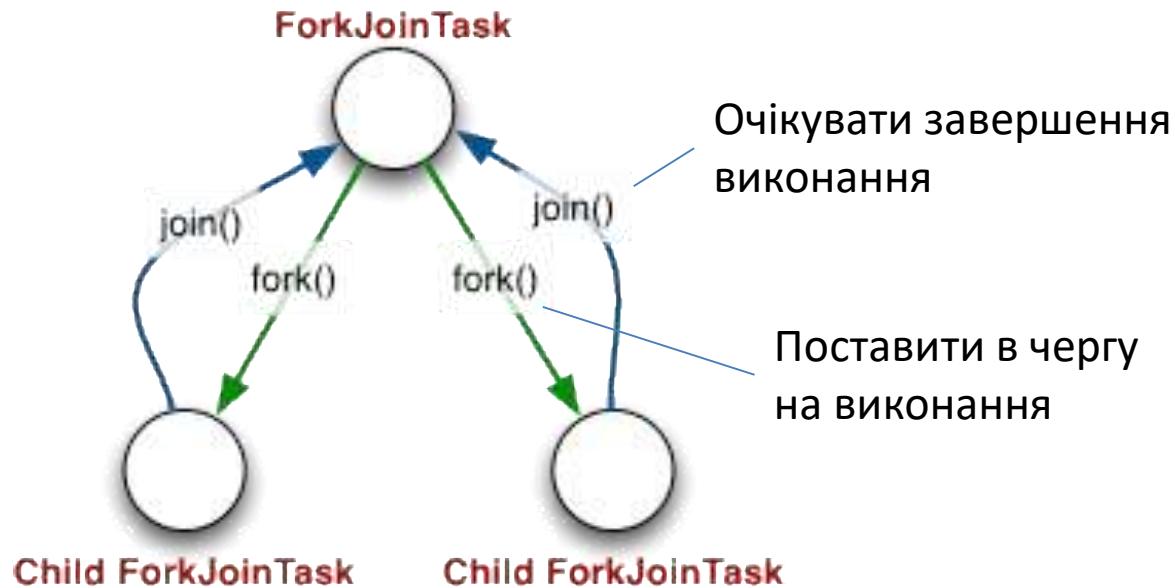
ForkJoinTask:

```
if (my portion of the work is small enough)
    do the work directly
else
    split my work into two pieces
    invoke the two pieces and wait for the results
```

ForkJoinPool:

```
invoke ()    all ForkJoinTask objects
```

Генерування підзадач у ForkJoin Framework



[<https://www.oracle.com/technical-resources/articles/java/fork-join.html>]

Документація класу ForkJoinPool

Class **ForkJoinPool**

java.lang.Object

java.util.concurrent.AbstractExecutorService

java.util.concurrent.ForkJoinPool

Клас імплементує такі інтерейси:

Executor, ExecutorService

Методи для запуску задач на виконання:

	Викликається ззовні fork/join обчислень (для пулу)	Викликається всередині fork/join обчислень (для задачі)
Запустити асинхронне виконання	execute (ForkJoinTask task)	ForkJoinTask.fork ()
Очікувати і отримати результат	invoke (ForkJoinTask task)	ForkJoinTask.invoke ()
Запустити на виконання і отримати Future	submit (ForkJoinTask task)	ForkJoinTask.fork () (ForkJoinTasks are Futures)

Основні методи класу ForkJoinPool

static ForkJoinPool **commonPool** () – returns the common pool instance.

void **execute** (ForkJoinTask<?> task) – arranges for (asynchronous) execution of the given task.

void **execute** (Runnable task) – executes the given command at some time in the future.

<T> T **invoke** (ForkJoinTask<T> task) – performs the given task, returning its result upon completion.

void **shutdown** () – possibly initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

List<Runnable> **shutdownNow** () – possibly attempts to cancel and/or stop all tasks, and reject all subsequently submitted tasks.

<T> ForkJoinTask<T> **submit** (Callable<T> task) – submits a value-returning task for execution and returns a Future representing the pending results of the task.

<T> ForkJoinTask<T> **submit** (ForkJoinTask<T> task) – submits a ForkJoinTask for execution.

<T> ForkJoinTask<T> **submit** (Runnable task, T result) – submits a Runnable task for execution and returns a Future representing that task.

Документація класу ForkJoinTask

java.util.concurrent

Class ForkJoinTask<V>

java.lang.Object

 java.util.concurrent.ForkJoinTask<V>

All Implemented Interfaces:

Serializable, Future<V>

Direct Known Subclasses:

CountedCompleter, RecursiveAction, RecursiveTask

```
public abstract class ForkJoinTask<V>
extends Object
implements Future<V>, Serializable
```

Основні методи класу ForkJoinTask

ForkJoinTask<V> **fork** () – arranges to asynchronously execute this task in the pool the current task is running in, if applicable, or using the ForkJoinPool.commonPool () if not inForkJoinPool () .

V **get** () - waits if necessary for the computation to complete, and then retrieves its result.

 V **get** (long timeout, TimeUnit unit) - waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.

 V **invoke** () - commences performing this task, awaits its completion if necessary, and returns its result, or throws an (unchecked) RuntimeException or Error if the underlying computation did so.

static <T extends ForkJoinTask<?>> Collection<T> **invokeAll** (Collection<T> tasks) - forks all tasks in the specified collection, returning when isDone holds for each task or an (unchecked) exception is encountered, in which case the exception is rethrown.

V **join** () - returns the result of the computation when it is done.

Приклад розробки ForkJoinTask-задачі

```
public class ForkBlur extends RecursiveAction {  
    ...  
    protected void computeDirectly() {  
        ... //обчислення для задачі  
    }  
    ...  
    @Override  
    protected void compute() {  
        if (mLength < sThreshold) {  
            computeDirectly();  
            return;  
        }  
        int split = mLength / 2;  
        invokeAll(  
            new ForkBlur(mSource, mStart, split, mDestination),  
            new ForkBlur(mSource, mStart + split,  
                         mLength - split, mDestination)  
        );  
    }  
}
```

Структура программы паралельных обчислень в ForkJoin Framework

1. Create a task that represents all of the work to be done.

```
// source image pixels are in src  
// destination image pixels are in dst  
ForkBlur fb = new ForkBlur(src, 0, src.length, dst);
```

2. Create the ForkJoinPool that will run the task.

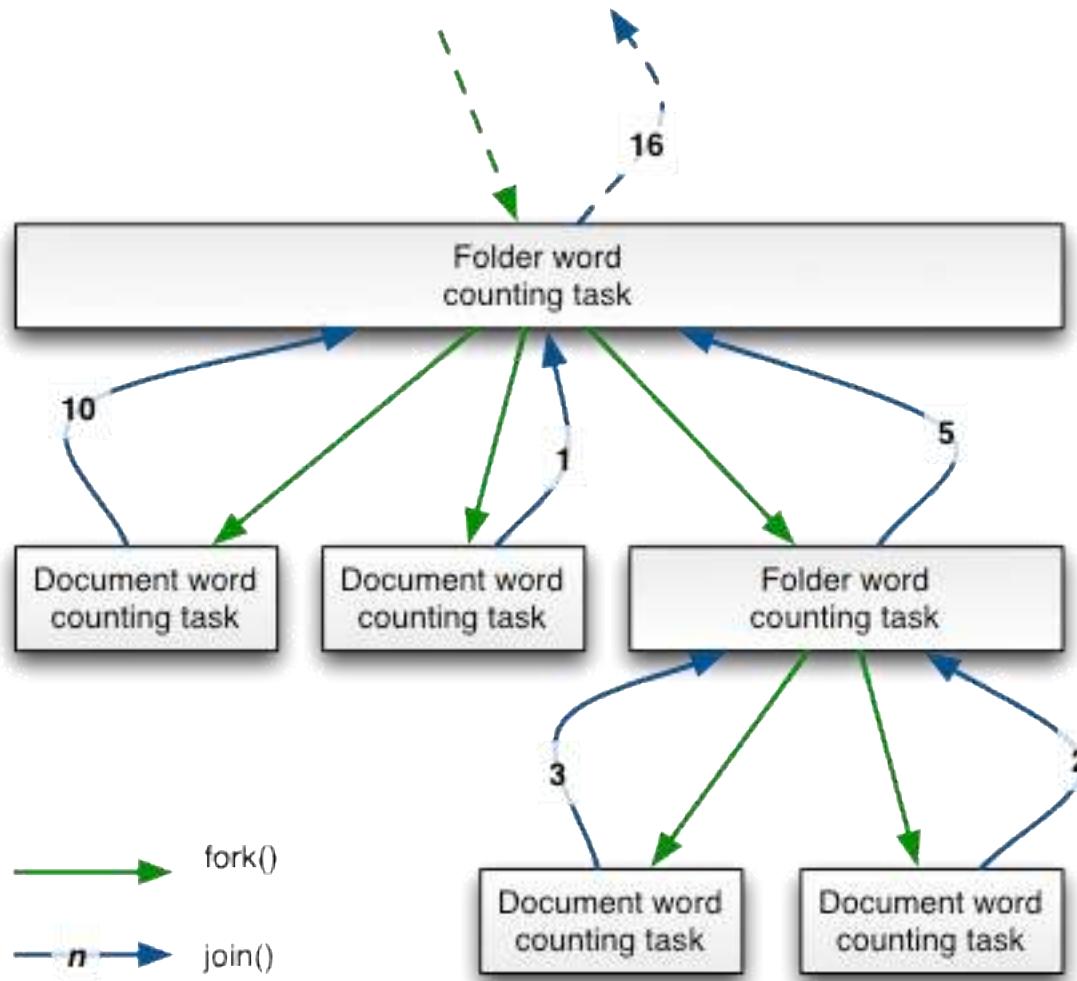
```
ForkJoinPool pool = new ForkJoinPool();
```

3. Run the task.

```
pool.invoke(fb);
```

Приклад: Counter of Searched Word

<https://www.oracle.com/technical-resources/articles/java/fork-join.html>



Приклад: Counter of Searched Word

<https://www.oracle.com/technical-resources/articles/java/fork-join.html>

```
class Document {  
    private final List<String> lines;  
    ....  
  
    static Document fromFile(File file)  
        throws IOException {  
        List<String> lines = new LinkedList<>();  
        try(BufferedReader reader =  
            new BufferedReader(new FileReader(file))) {  
            String line = reader.readLine();  
            while (line != null) {  
                lines.add(line);  
                line = reader.readLine();  
            }  
        }  
        return new Document(lines);  
    }  
}
```

Приклад: Counter of Searched Word

```
public class Folder {  
    private final List<Folder> subFolders;  
    private final List<Document> documents;  
    ...  
  
    static Folder fromDirectory(File dir) throws IOException {  
        List<Document> documents = new LinkedList<>();  
        List<Folder> subFolders = new LinkedList<>();  
  
        for (File entry : dir.listFiles()) {  
            if (entry.isDirectory()) {  
                subFolders.add(Folder.fromDirectory(entry));  
            } else {  
                documents.add(Document.fromFile(entry));  
            }  
        }  
        return new Folder(subFolders, documents);  
    }  
}
```

Приклад: Counter of Searched Word

```
public class WordCounter {  
    private final ForkJoinPool forkJoinPool =  
        new ForkJoinPool();  
    String[] wordsIn(String line) {  
        return line.trim().split("(\\s|\\p{Punct})+");  
    }  
  
    Long occurrencesCount(Document document,  
                          String searchedWord) {  
        long count = 0;  
        for (String line : document.getLines()) {  
            for (String word : wordsIn(line)) {  
                if (searchedWord.equals(word)) {  
                    count++;  
                }  
            }  
        }  
        return count;  
    }  
}
```

Приклад: Counter of Searched Word

```
Long countOccurrencesOnSingleThread( Folder folder,
                                         String searchedWord) {
    long count = 0;
    for (Folder subFolder : folder.getSubFolders()) {
        count = count + countOccurrencesOnSingleThread(
                                         subFolder, searchedWord);
    }
    for (Document document : folder.getDocuments()) {
        count = count + occurrencesCount(document,
                                         searchedWord);
    }
    return count;
}
```

Приклад: Counter of Searched Word

```
class DocumentSearchTask extends RecursiveTask<Long> {  
    .....  
}  
class FolderSearchTask extends RecursiveTask<Long> {  
    .....  
}  
Long countOccurrencesInParallel(Folder folder,  
                           String searchedWord) {  
    return forkJoinPool.invoke(  
        new FolderSearchTask(folder, searchedWord));  
}  
}
```

```
class DocumentSearchTask extends RecursiveTask<Long> {  
    private final Document document;  
    private final String searchedWord;  
    .....  
    @Override  
    protected Long compute() {  
        return occurrencesCount(document, searchedWord);  
    }  
}
```

```
class FolderSearchTask extends RecursiveTask<Long> {
    private final Folder folder;
    private final String searchedWord;
    .....
    @Override
    protected Long compute() {
        long count = 0L;
        List<RecursiveTask<Long>> tasks = new LinkedList<>();

        for (Document document : folder.getDocuments()) {
            DocumentSearchTask task =
                new DocumentSearchTask(document, searchedWord);
            tasks.add(task);
            task.fork();
        }

        for (RecursiveTask<Long> task : tasks) {
            count = count + task.join();
        }
        return count;
    }
}
```

```
public class WordCounter {  
    public static void main(String[] args) throws IOException {  
        Folder folder =  
            Folder.fromDirectory(new File("D:/TextFolder"));  
        WordCounter wordCounter = new WordCounter();  
        String searchedWord = "synchronized";  
        final int repeatCount = 4;  
        long cstartTime, stopTime, counts=0, averTime = 0;  
        long averTime = 0;  
        for (int i = 0; i < repeatCount; i++) {  
            startTime = System.currentTimeMillis();  
            counts = wordCounter.countOccurrencesInParallel(folder,  
                searchedWord);  
            stopTime = System.currentTimeMillis();  
            averTime+=stopTime - startTime;  
        }  
        System.out.println(counts +  
            " words are fined. Fork / join search took " +  
            averTime/repeatCount+ "ms");  
    }  
}
```

```
averTime = 0;
for (int i = 0; i < repeatCount; i++) {
    startTime = System.currentTimeMillis();
    counts = wordCounter.countOccurrencesOnSingleThread(  

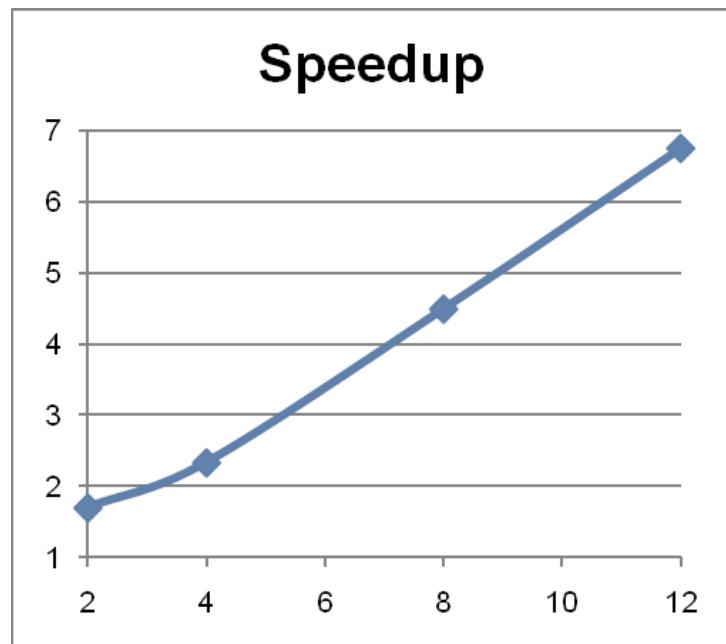
                           folder, searchedWord)
    stopTime = System.currentTimeMillis();
    averTime+=stopTime - startTime;
}
System.out.println(counts +
    " words are fined. Single thread search took " +
    averTime/repeatCount+ "ms");
}
```

Ефективність ForkJoin Framework

Результат дослідження ефективності алгоритму пошуку слів в документах при запуску на різній кількості ядер Sun Fire T2000 server from Oracle [<https://www.oracle.com/technical-resources/articles/java/fork-join.html>]:

Number of Cores	Single-Thread Execution Time (ms)	Fork/Join Execution* Time (ms)	Speedup
2	18798	11026	1.704879376
4	19473	8329	2.337975747
8	18911	4208	4.494058935
12	19410	2876	6.748956885

*Час виконання алгоритму оцінювався за найменшим значенням серед кількох запусків.



Приклад: результат запуску

```
14
15     long averTime = 0;
16     for (int i = 0; i < repeatCount; i++) {
17         startTime = System.currentTimeMillis();
18         counts = wordCounter.countOccurrencesInParallel(folder, searchedWord);
19         stopTime = System.currentTimeMillis();
20         averTime+=stopTime - startTime;
21     }
22     System.out.println(counts + " words are fined. Fork / join search took " +averTime);
23
24     averTime = 0;
25     for (int i = 0; i < repeatCount; i++) {
```

III

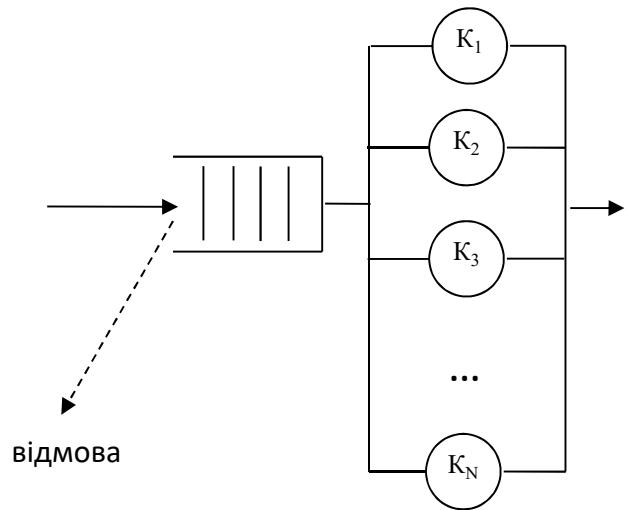
ForkJoinExample.WordCounter > main > for (int i = 0; i < repeatCount; i++) >

вод - Threads (run) ✘

```
run:
36 words are fined. Fork / join search took 134ms
36 words are fined. Single thread search took 42ms
СВОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 1 секунда)
```

Моделювання паралельних обчислень

Завдання до комп'ютерного практикуму 5



- Для отримання статистично значимої оцінки середньої довжини черги та ймовірності відмови, їх значення оцінюють за кількома прогонами «моделі» (не менше 4)
- Виведення результатів (для того, щоб отримувати поточну інформацію про перебіг імітації) виконують в окремому потоці через задану часову затримку.

Правила обслуговування в багатоканальній СМО з обмеженою чергою:

- Об'єкти надходять на обслуговування в СМО через випадкові інтервали часу з заданим середнім значенням (розподілених, наприклад, за рівномірним законом розподілу).
- Якщо є вільний канал обслуговування, то об'єкт займає його на час обслуговування.
- В протилежному випадку, об'єкт намагається стати в чергу. Якщо місце в черзі є, то об'єкт займає його. В протилежному випадку, об'єкт залишає СМО необслугованим (збільшується кількість відмов в обслуговуванні).
- Після завершення обслуговування в каналі об'єкт вважається таким, що завершив обслуговування в СМО (збільшується кількість обслугованих об'єктів).
- Моделювання здійснюють протягом заданого інтервалу часу (іншим способом є завершення за кількістю об'єктів, які завершили обслуговування). Час моделювання має бути достатньо великим, щоб обслуговування здійснилось для не менш як 1000 об'єктів.
- За результатом імітації підраховують: ймовірність відмови, середня кількість об'єктів в черзі.

Задачі моделювання

Оцінювання ефективності паралельних обчислень застосовується у таких формах:

- 1) Оцінювання прискорення процесу обчислень
- 2) Оцінювання максимально можливого прискорення процесу обчислень

Модель обчислень у вигляді графу « операції-операнди»

Припущення:

- усі обчислювальні операції виконуються з однаковою з часовою затримкою рівною 1,
- передача даних між обчислювальними пристроями здійснюється миттєво (!вірно, якщо використовується спільна модель пам'яті в розподіленій системі)

Структура графу

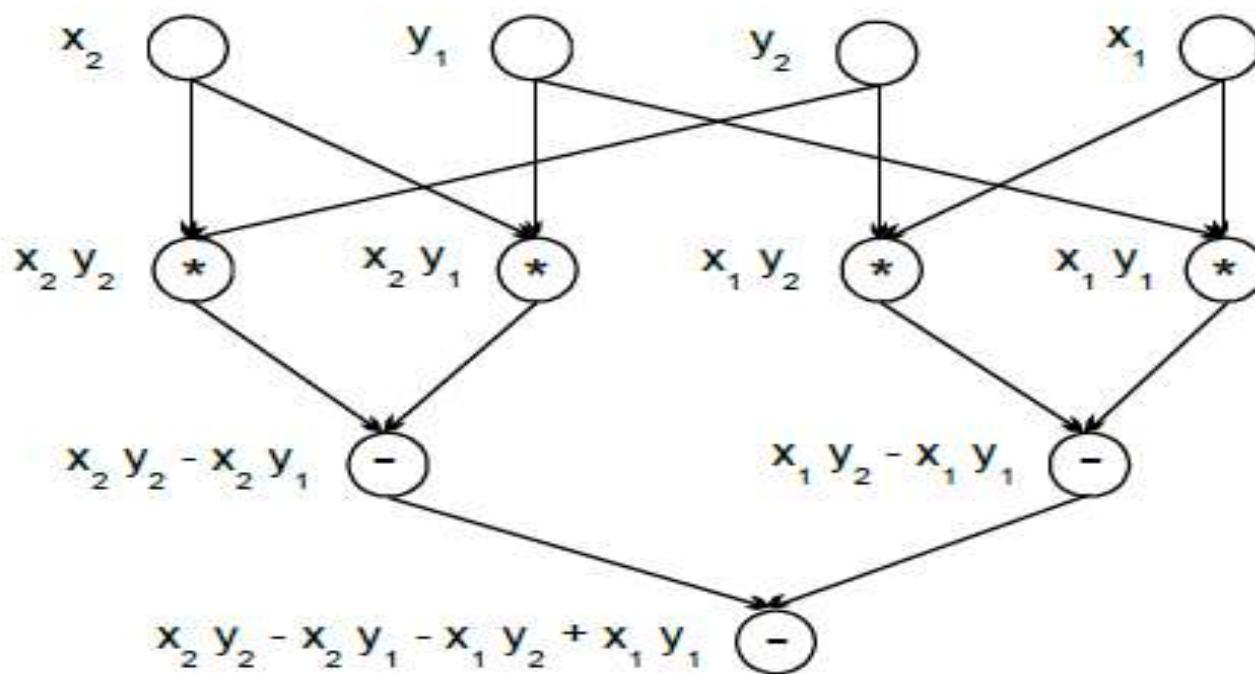
Вершини графу представляють операції алгоритму.

Дуга між вершинами графу існує, якщо операція – кінець дуги використовує результат обчислень операції – старт дуги.

Приклад

(x_2, y_2)
 x_1, y_1)

$$S = ((x_2 - x_1)(y_2 - y_1)) =$$
$$= x_2 y_2 - x_2 y_1 - x_1 y_2 + x_1 y_1$$



Визначення часу виконання паралельних обчислень

- Мінімально можливий час виконання паралельного алгоритму визначається довжиною максимального шляху обчислювальної схеми алгоритму:

$$T_{\infty}(G) = d(G),$$

де T_{∞} - час виконання паралельного алгоритму на **паракомп'ютері** (з нескінченно великою кількістю процесорів) $T_{\infty} = \min_p T_p$

- Мінімально можливий час виконання паралельного алгоритму, за умови, що кількість входів у кожну вершину графу не перевищує 2, обмежений знизу величиною

$$T_{\infty}(G) \geq \log_2 n,$$

де n – оцінка складності алгоритму (кількість вхідних даних, наприклад).

Визначення часу виконання паралельних обчислень

- При зменшенні кількості використовуваних процесорів час виконання алгоритму збільшується пропорційно:

$$0 < c < 1 \quad T_{cp} \leq \frac{T_p}{c}$$

Наприклад, при $p=16$, $c=0,5$: $T_8 \leq 2T_{16}$

Окремий випадок: якщо $cp = 1$, то $T_1 \leq pT_p$, або $T_p \geq \frac{T_1}{p}$.

Тобто час виконання алгоритму на p процесорах обмежений знизу величиною $\frac{T_1}{p}$.

- Часу виконання паралельного алгоритму на p процесорах обмежений зверху:

$$\forall p \quad T_p < T_\infty + \frac{T_1}{p},$$

де T_1 - час виконання послідовного алгоритму

Визначення часу виконання паралельних обчислень

- Час виконання алгоритму, який наблизений до мінімально можливого часу T_∞ , можна досягнути при такій кількості процесорів:

$$p \sim \frac{T_1}{T_\infty}. \text{ (Дійсно, оскільки } T_\infty \rightarrow \frac{T_1}{p}, \text{ то } p \rightarrow \frac{T_1}{T_\infty})$$

При цьому,

$$\text{якщо } p \geq \frac{T_1}{T_\infty}, \text{ то } T_p \leq 2T_\infty,$$

$$\text{якщо } p < \frac{T_1}{T_\infty}, \text{ то } \frac{T_1}{p} \leq T_p \leq 2 \frac{T_1}{p}$$

Рекомендації до складання паралельних алгоритмів

- 1) При виборі обчислюальної схеми повинен використовуватись граф з мінімально можливим діаметром
- 2) Для паралельного виконання доцільно кількість процесорів визначати величиною $p \sim \frac{T_1}{T_\infty}$
- 3) Час виконання паралельного алгоритму обмежується зверху величинами:

$$\forall p \quad T_p < T_\infty + \frac{T_1}{p}$$

$$if \ p \geq \frac{T_1}{T_\infty} \Rightarrow \ T_p \leq 2T_\infty;$$

$$if \ p < \frac{T_1}{T_\infty} \Rightarrow \ T_p \leq 2 \frac{T_1}{p}$$

Показники ефективності паралельного алгоритму

- **Прискорення**, одержуване при використанні p процесорів для реалізації паралельного алгоритму складності n , оцінюється величиною:

$$S_p(n) = T_1(n)/T_p(n)$$

- **Ефективність** використання процесорів при паралельній реалізації алгоритму оцінюється величиною:

$$E_p(n) = T_1(n)/(p \cdot T_p(n)) = S_p(n)/p$$

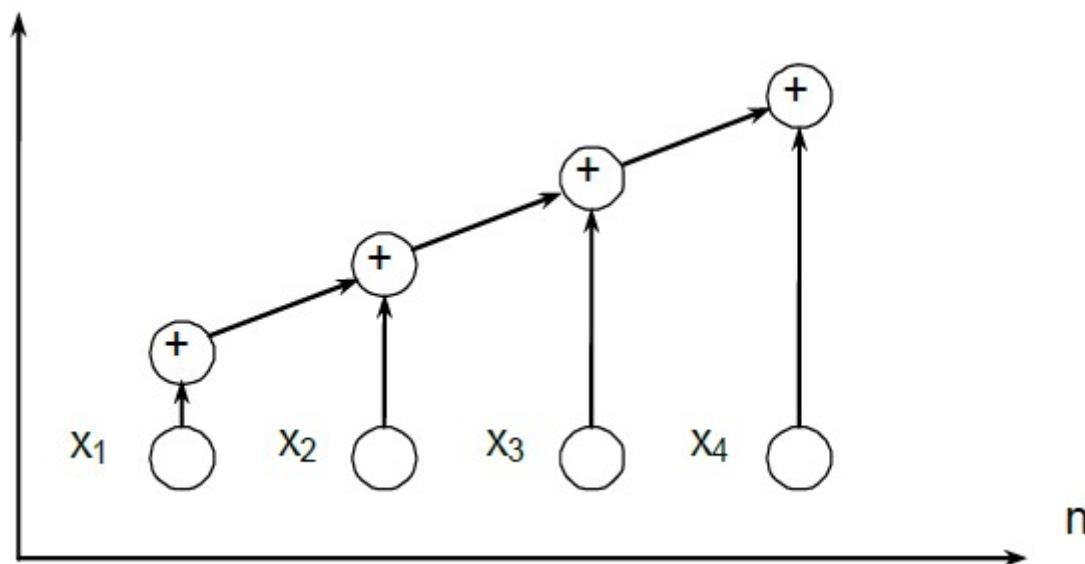
Зauważення: $E_p(n) \leq 1$, $S_p(n) \leq p$. Якщо досягається $S_p(n) > p$, то говорять про суперлінійне прискорення. Це можливо для алгоритмів з нелінійною складністю

- **Вартість** обчислень

$$C_p = pT_p$$

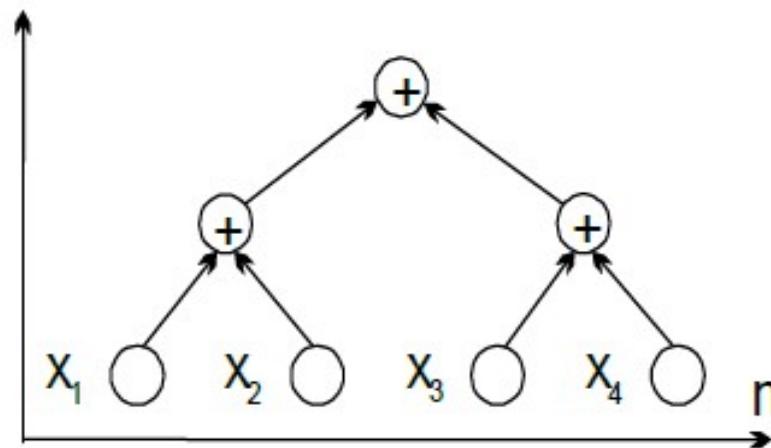
Приклад: алгоритм обчислення суми

Сума $S=\sum_{i=1}^n x_i$ послідовно розраховується у такі кроки: $S=0, S=S+x_1, \dots$ Схема цих обчислень має такий вид і не може бути розпаралелена:



Приклад: алгоритм обчислення суми

Каскадна схема паралельного алгоритму
сумування:



$$T_p = \log_2 n \qquad p = \frac{n}{2}$$

Ефективність алгоритму обчислення суми

Кількість операцій сумування:

$$T_1 = n - 1$$

Кількість паралельних операцій сумування:

$$T_p = \log_2 n$$

Маємо такі оцінки ефективності:

$$S_p = T_1/T_p = (n - 1)/\log_2 n,$$

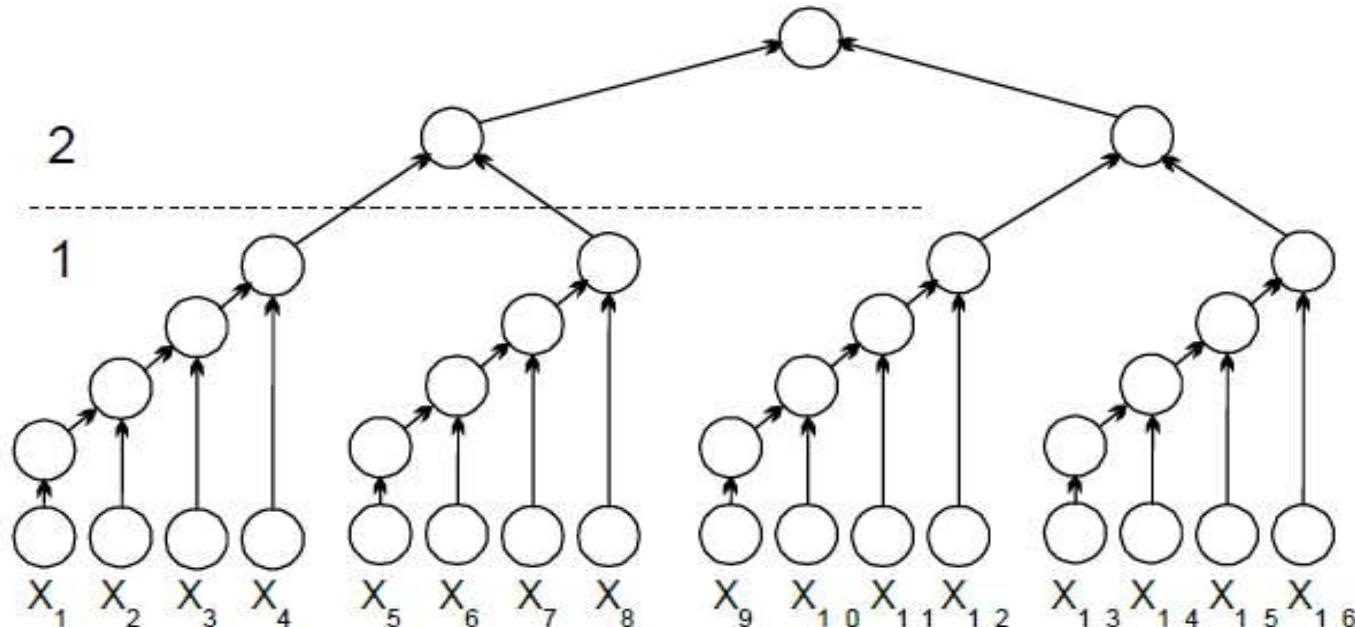
$$\begin{aligned} E_p &= S_p/p = \\ &= (n - 1)/(p \log_2 n) = (n - 1)/\left(\frac{n}{2} \log_2 n\right) \end{aligned}$$

При зростанні кількості даних: $\lim_{n \rightarrow \infty} E_p \rightarrow 0$

Модифікована каскадна схема паралельного обчислення сум

$n = 2^k$, кількість груп = $n/k = n/\log_2 n$

$T_p \leq 2\log_2 n$, p = кількість груп = $n/k = n/\log_2 n$



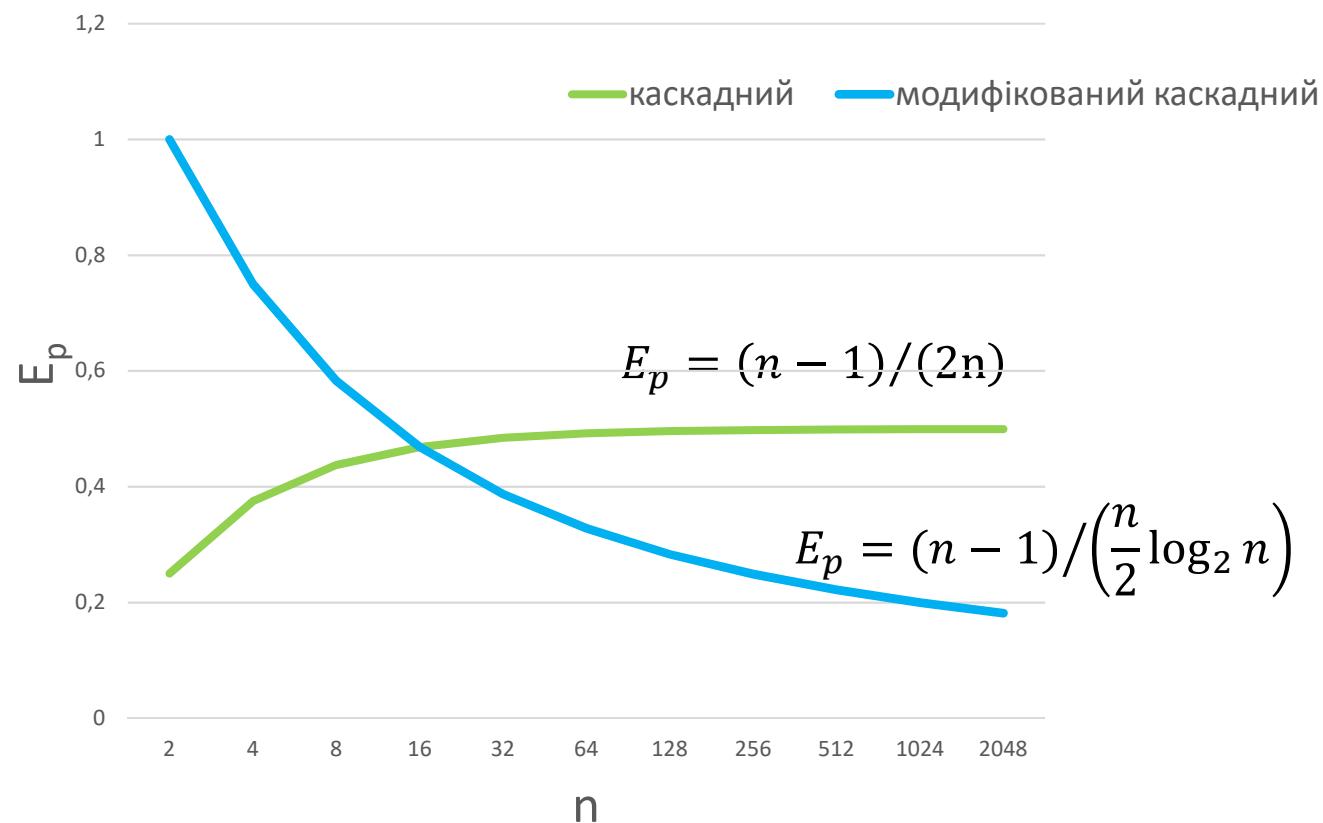
$$T_p = (k - 1) + \log_2 \frac{n}{k} = \log_2 n - 1 + \log_2 \frac{n}{k} = 2 \log_2 n - 1 - \log_2 k \leq 2 \log_2 n$$

Ефективність модифікованого каскадного сумування

- $S_p = T_1/T_p = (n - 1)/(2\log_2 n)$
- $E_p = T_1/(p \cdot T_p) =$
 $(n - 1)/(n/\log_2 n \cdot 2\log_2 n) = (n - 1)/(2n)$

Маємо (!!): $\lim_{n \rightarrow \infty} E_p \rightarrow 0,5$

Порівняння ефективності звичайного та модифікованого каскадних алгоритмів



Виведення закону Амдала

f – частка коду, який не може бути розпаралелений

$$S_p = \frac{T_1}{T_p} \leq \frac{fT_1 + (1-f)T_1}{fT_1 + \frac{(1-f)T_1}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$$

Послідовна
частина коду

Паралельна
частина коду

Максимально досяжний паралелизм

- **Закон Амдала**

$$S_p \leq \frac{1}{f + \frac{(1-f)}{p}} \leq \frac{1}{f} ,$$

де f – частка послідовних обчислень в алгоритмі.

Наприклад, для каскадної схеми $f = \frac{k}{n} = \log_2 n / n$, тому прискорення обмежене величиною

$$S_p \leq \frac{1}{f} = n / \log_2 n$$

- **Ефект Амдала:** оскільки $f(n)$ спадає при зростанні n , то $S_p(n)$ є зростаючою функцією.

Аналіз масштабованості паралельних обчислень

- Паралельний алгоритм називають **масштабованим** (*scalable*), якщо при зростанні кількості процесорів він забезпечує збільшення прискорення при збереженні постійного рівня ефективності використання процесорів.
- Накладні витрати, які мають місце при виконанні паралельного алгоритму, оцінюються величиною

$$T_0 = pT_p - T_1$$

Аналіз масштабованості паралельних обчислень

- Для бажаного значення ефективності E виконано співвідношення:

$$\frac{T_0}{T_1} = \frac{1 - E}{E} = \frac{1}{K} \Rightarrow T_1 = KT_0, \text{де } K = \frac{E}{1 - E}$$

- З останнього співвідношення може бути одержана залежність виду $n = F(p)$ між складністю задачі та кількістю процесорів. Цю залежність називають функцією **ізоекспективності**.

Наприклад, для алгоритму обчислення суми маємо:

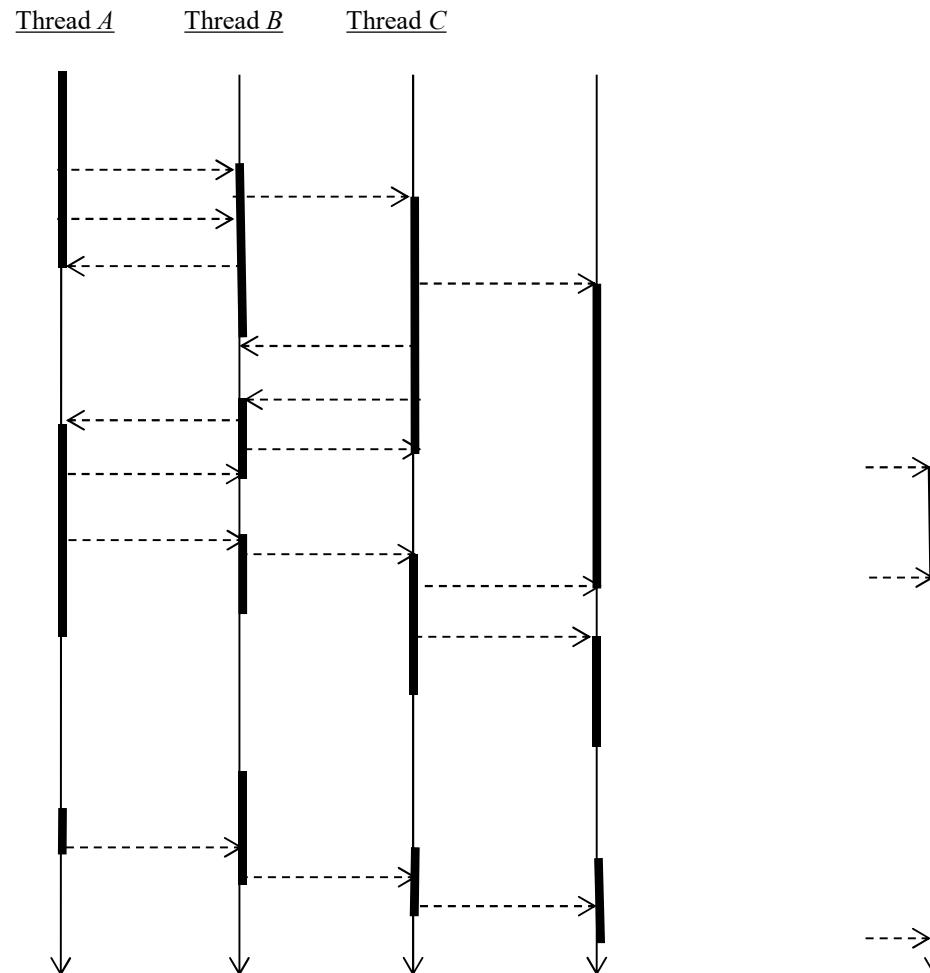
$$T_0 = pT_p - T_1 = p \left(\frac{n}{p} + \log_2 p \right) - n = p \log_2 p$$
$$T_1 = KT_0 \Rightarrow n = Kp \log_2 p$$

Наприклад, при $E = 0,5$ (тобто $K = 1$) та $p = 16$ з останньої формули отримуємо $n = 64$.

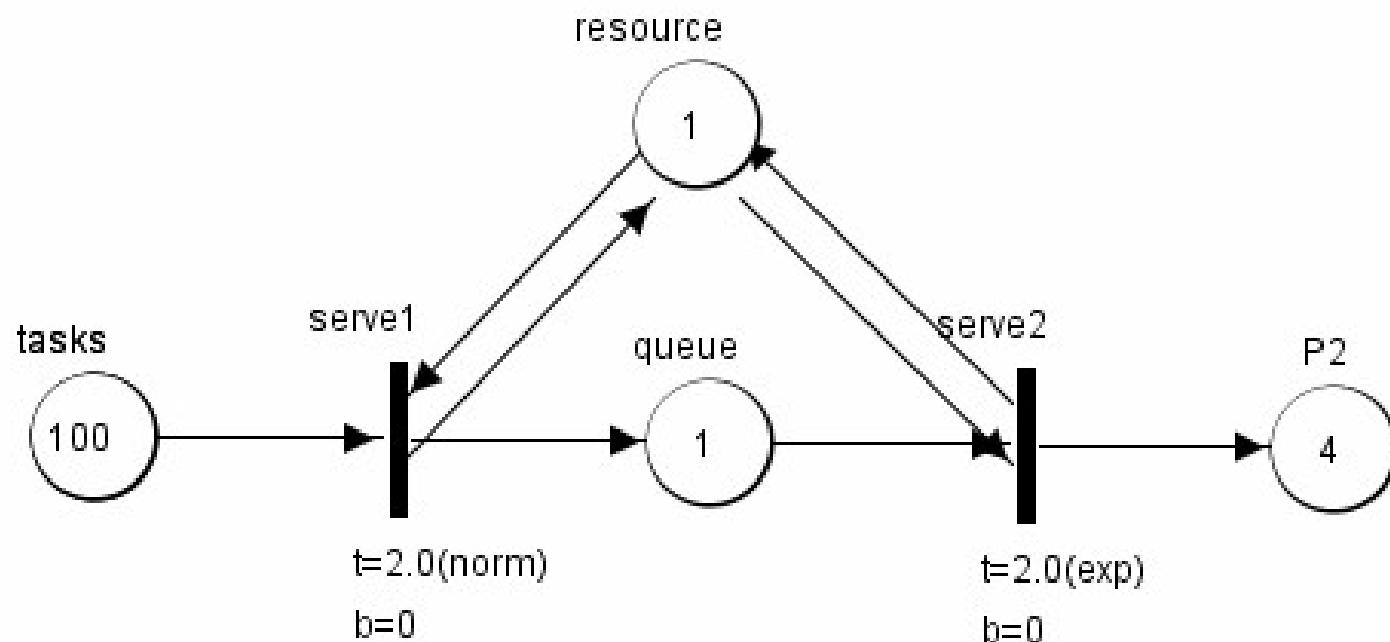
Імітаційне моделювання багатопочної програми засобами Петрі-об'єктного моделювання

Стеценко І.В.
Дифучина О.Ю.
КПІ ім. Ігоря Сікорського

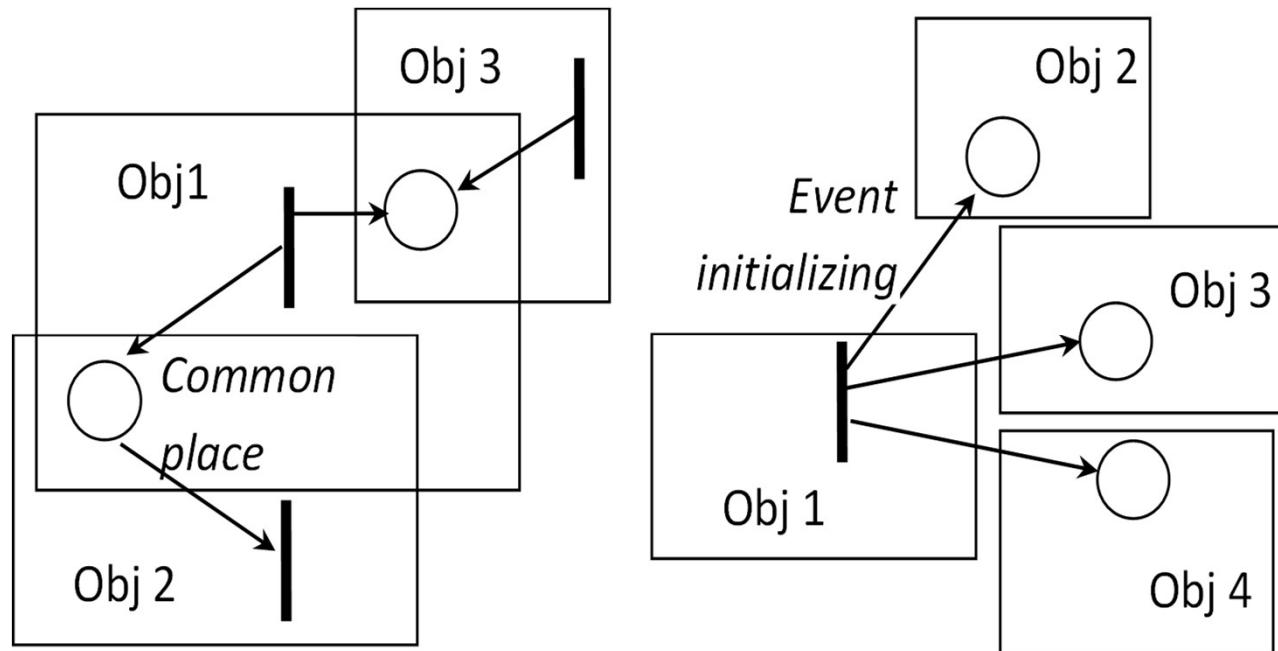
Діаграма потоків



Стохастична мережа Петрі



Петрі-об'єктна модель

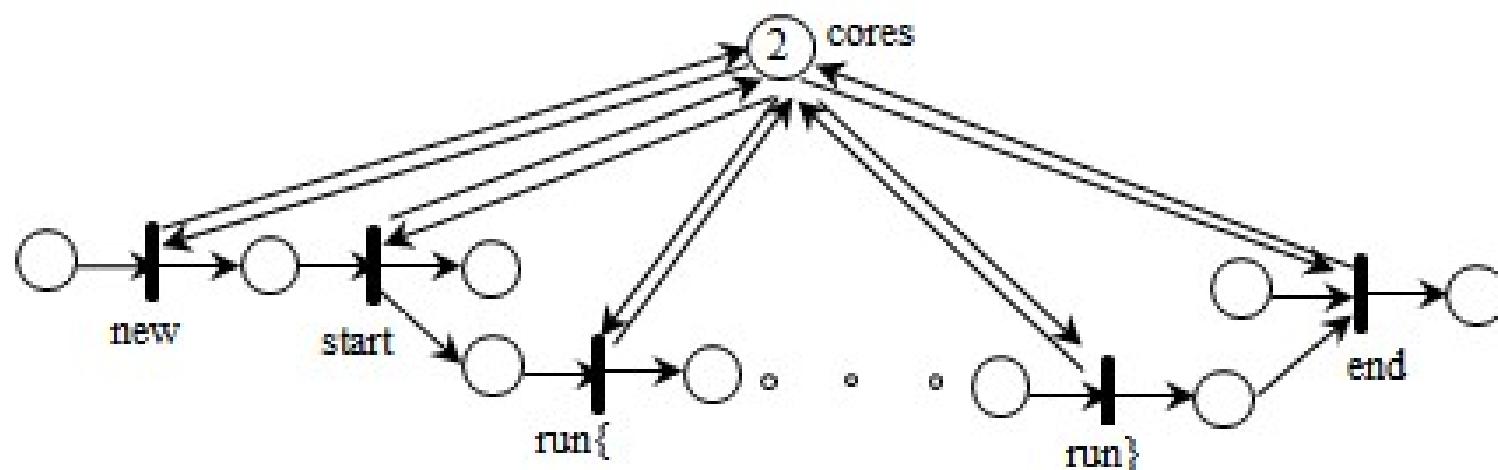


Література

- Stetsenko I.V., Dydychyna O. (2019) Simulation of Multithreaded Algorithms Using Petri-Object Models. In: Hu Z., Petoukhov S., Dychka I., He M. (eds) Advances in Computer Science for Engineering and Education. ICCSEEA 2018. Advances in Intelligent Systems and Computing, vol 754. - Springer, Cham. - P.391-401.
- Stetsenko I.V., Dydychyna O. (2020) Thread Pool Parameters Tuning Using Simulation. In: Hu Z., Petoukhov S., Dychka I., He M. (eds) Advances in Computer Science for Engineering and Education II. ICCSEEA 2019. Advances in Intelligent Systems and Computing, vol 938. Springer, Cham. - P.78-89. (Scopus) https://doi.org/10.1007/978-3-030-16621-2_8
- Stetsenko I.V., Pavlov A.A., Dydychyna O. (2021) Parallel algorithm development and testing using Petri-object simulation. International Journal of Parallel, Emergent and Distributed Systems. Taylor & Francis. 1-16. <https://doi.org/10.1080/17445760.2021.1955113> ISSN 1744-5760

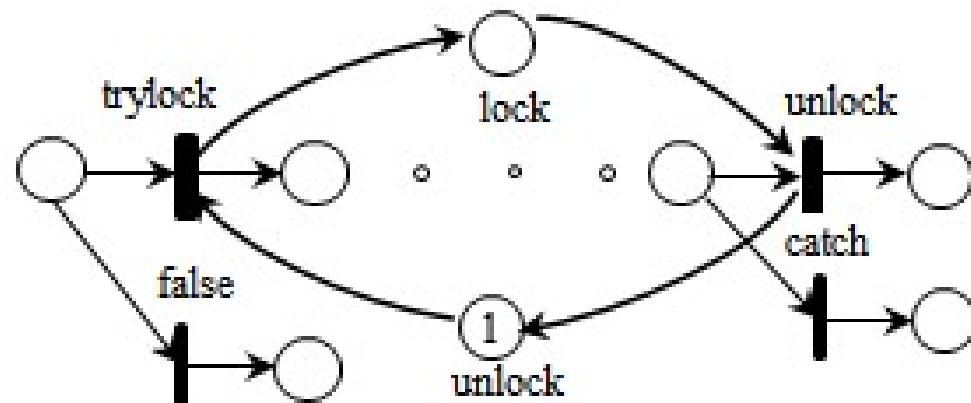
Implementation of thread's creating, starting and ending

```
public static void main(String[] args) {  
    Thread thread = new Thread(new Runnable());  
    thread.start();  
}
```



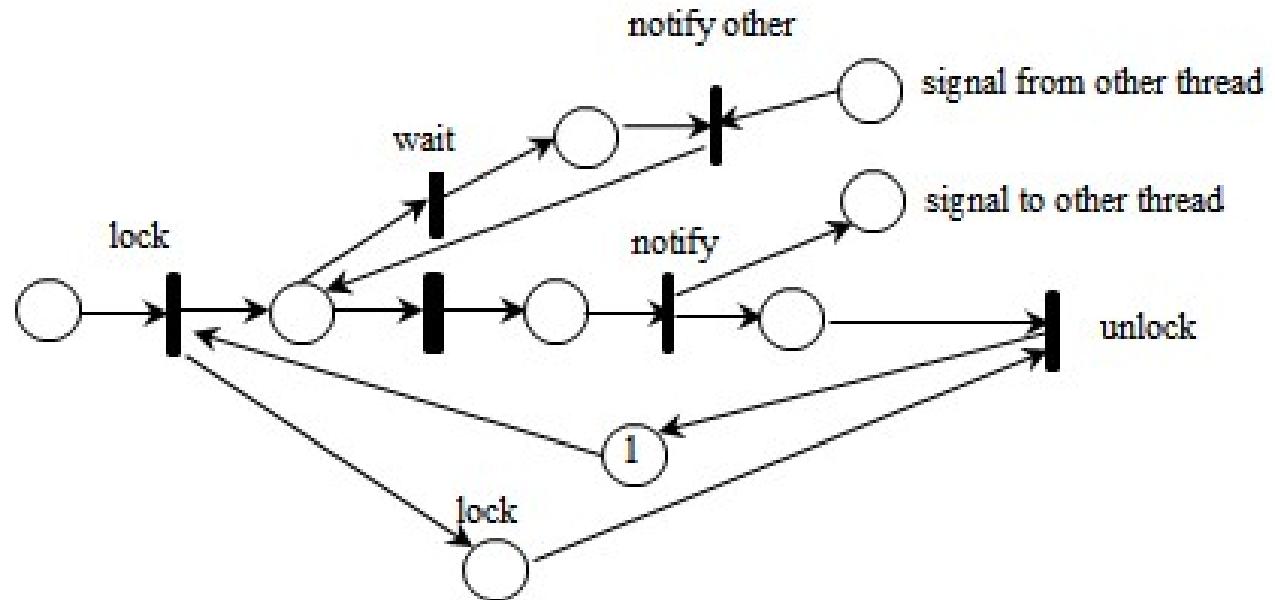
Implementation of thread's locking

```
private final Lock lock = new ReentrantLock();
try{
    lock = lock.tryLock();
    ...// synchronized actions
} finally {
    lock.unlock();
}
```



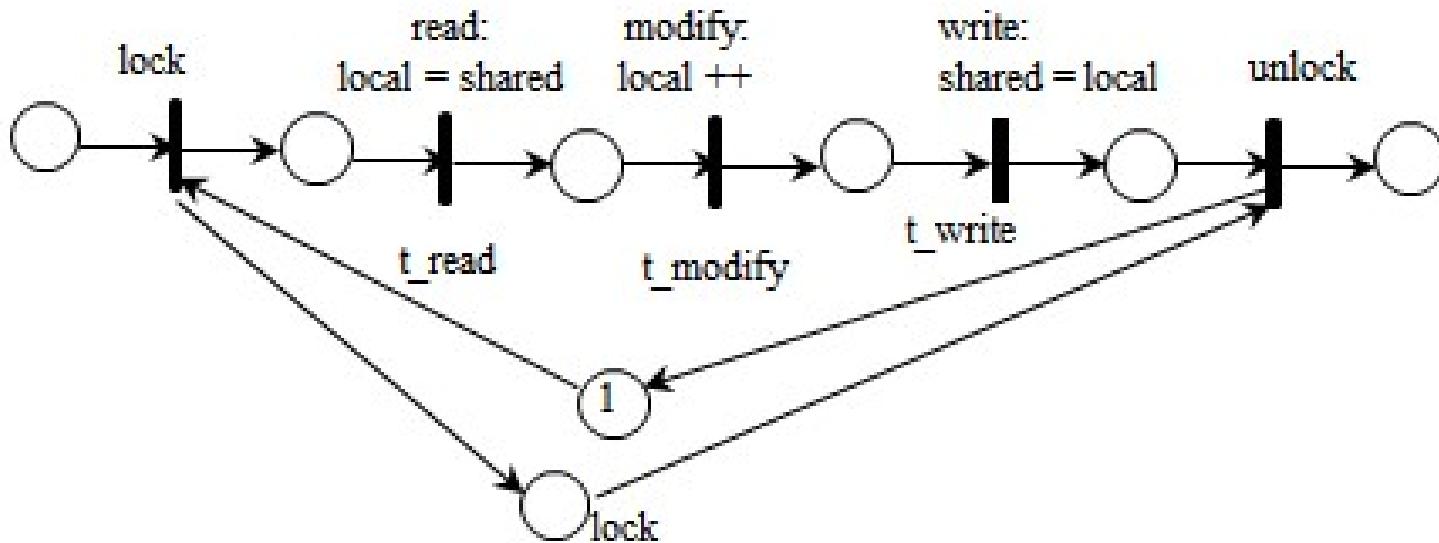
Implementation of thread's guarded block

```
public synchronized void method() throws InterruptedException {  
    while (!condition()) {  
        wait();  
    }  
    ...// some actions  
    notifyAll();  
}
```

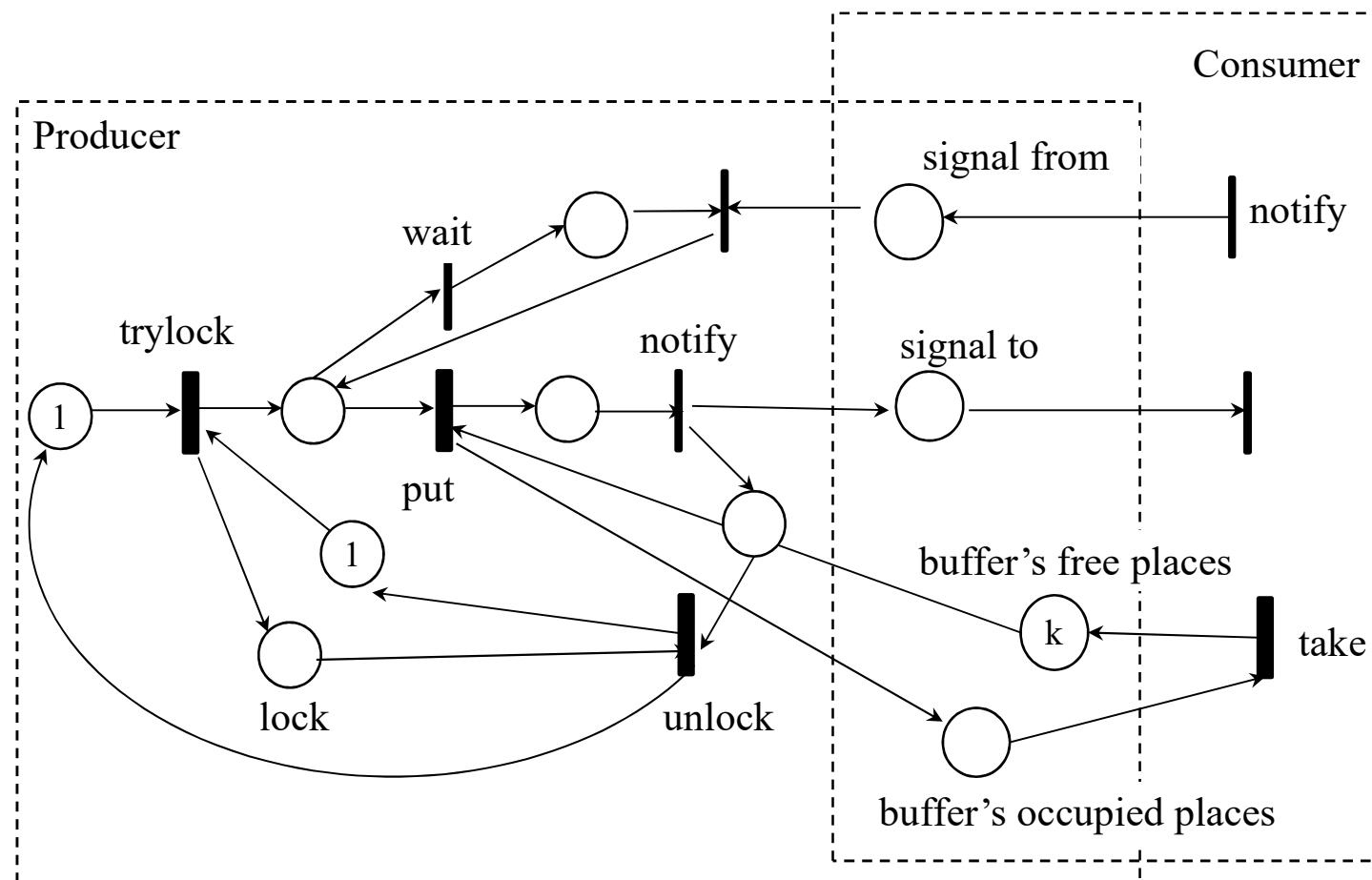


Implementation of thread's shared data access

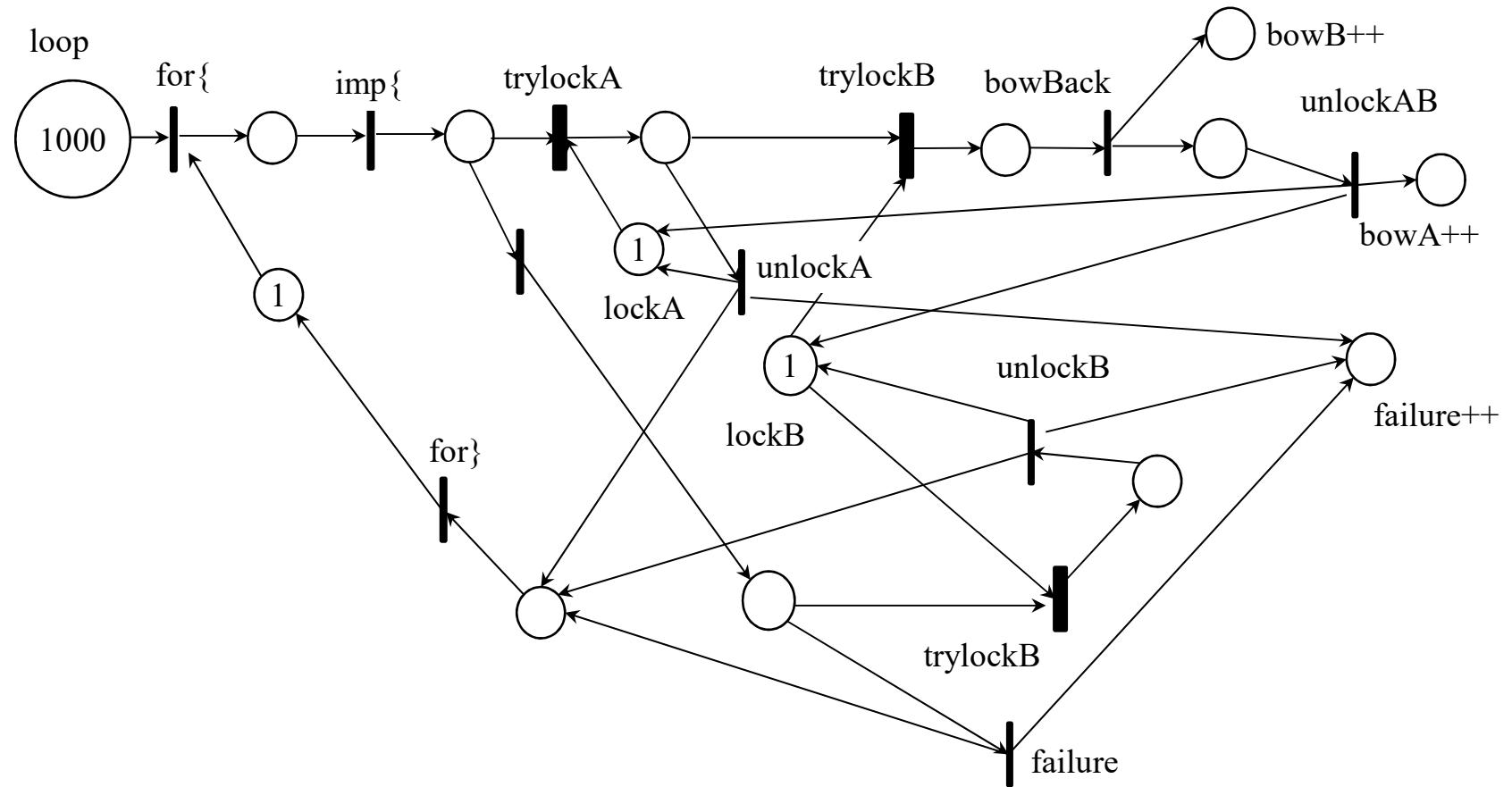
```
public synchronized void incMethod(){  
    local++;  
}
```



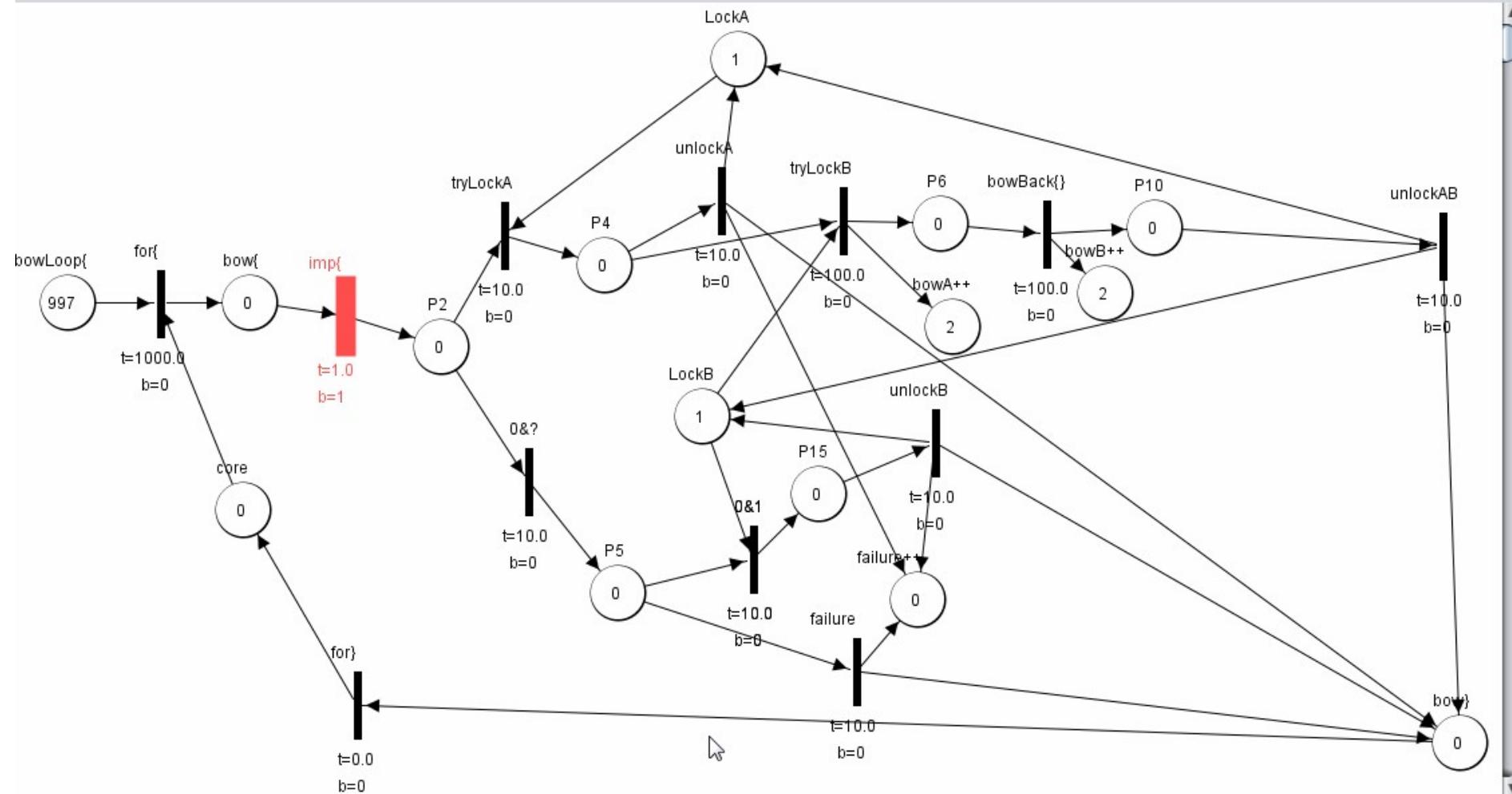
The net of Petri-object Producer in connection with the net of Petri-object Consumer



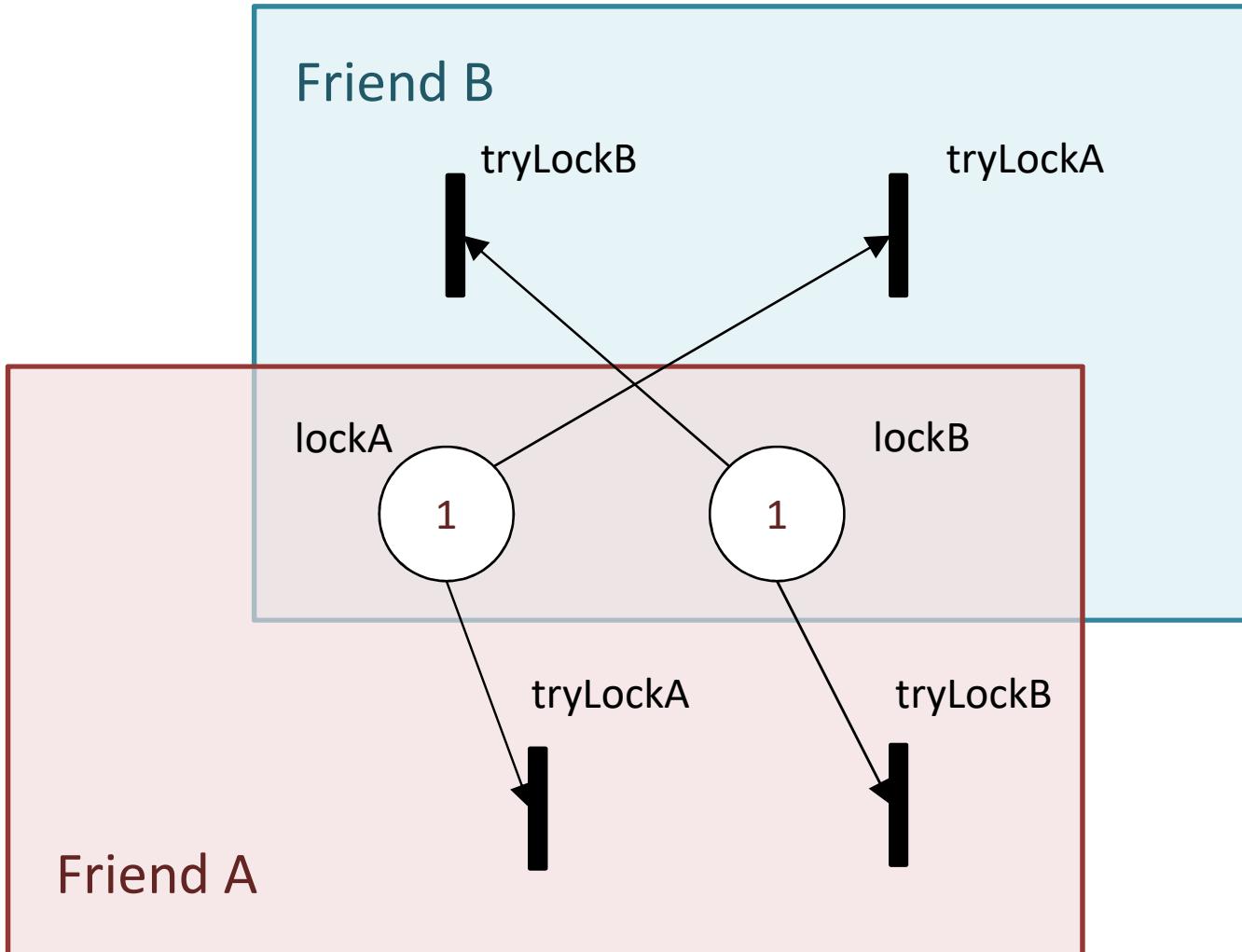
The net of Petri-object Friend



Transition Arc



Connections between two Petri-objects



Код конструювання зв'язків Петрі-об'єктів

```
public static void main(String[] args) throws ExceptionInvalidNetStructure {
//...
    class Friend extends PetriSim {
        public Friend(String name, int loop) throws ExceptionInvalidNetStructure {
            super(NetLibrary.CreateNetFriendUsingCores(name, loop, 2)); // 2 cores
        }
        public void addFriend(Friend other){
            this.getNet().getListP()[7] = other.getNet().getListP()[2]; //lockOther = lock
            this.getNet().getListP()[15] = other.getNet().getListP()[15]; // coresOther = cores
        }
    }
    Friend friendA = new Friend("A", 1000);
    Friend friendB = new Friend("B", 1000);
    Friend friendC = new Friend("C", 1000);
    Friend friendD = new Friend("D", 1000);

    friendA.addFriend(friendB);
    friendA.addFriend(friendC);
    friendA.addFriend(friendD);

    friendB.addFriend(friendA);
    friendB.addFriend(friendC);
    friendB.addFriend(friendD);

    friendC.addFriend(friendA);
    friendC.addFriend(friendB);
    friendC.addFriend(friendD);

    friendD.addFriend(friendA);
    friendD.addFriend(friendB);
    friendD.addFriend(friendC);
}
}
```

Код конструювання Петрі-об'єктної моделі з Петрі-об'єктів

```
public static void main(String[] args) throws ExceptionInvalidNetStructure {  
    ArrayList<PetriSim> list = new ArrayList<>();  
  
    //Petri-objects creation  
  
    list.add(friendA);  
    list.add(friendB);  
    list.add(friendC);  
    list.add(friendD);  
  
    PetriObjModel model = new PetriObjModel(list);  
  
    model.setIsProtokol(false);  
    model.go(100000000);  
}
```

Експериментальне дослідження на моделі ймовірності виникнення конфлікту потоків в залежності від співвідношення часових затримок

$4 \bullet 3 = 12$ threads

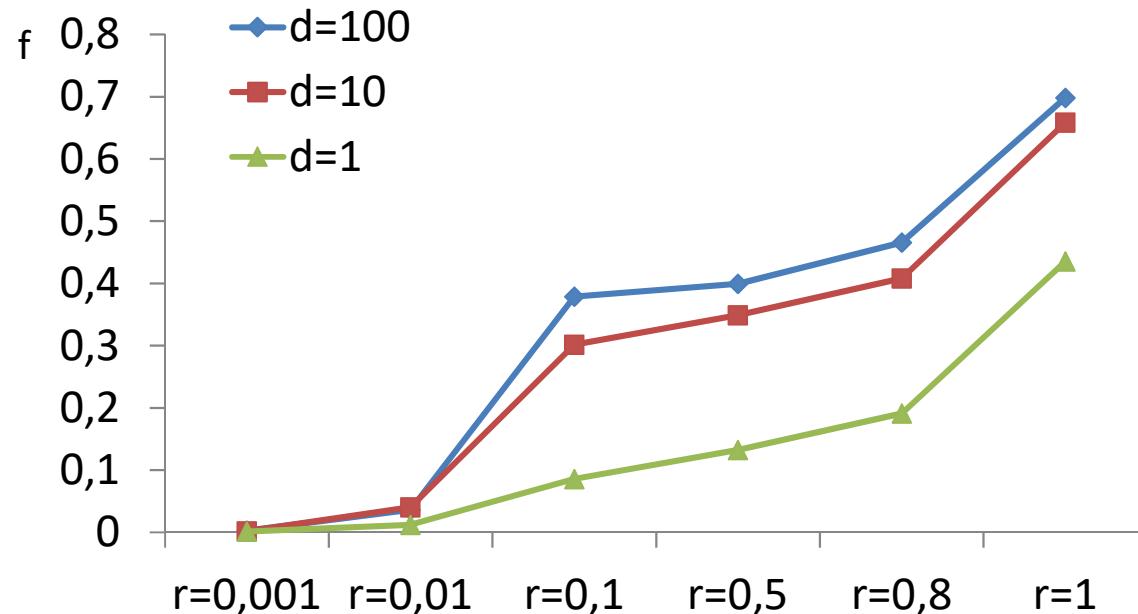
d часова затримка переходу “for{”,

що відповідає затримці в методі sleep() у програмі

$d \bullet r$ часова затримка інших переходів, r - співвідношення часових затримок

що відповідає затримци на виконання елементарної опреації обчислень іs

f частота виникнення конфлікту потоків



Точність моделі

Number of threads	Multithreaded program	Simulation model (d=100, r=0.01)	Error
2	0.981450	0.978650	0.29%
12	0.959042	0.963417	0.46%
10 (90 tasks)	0.987777	0.979080	0.88%
20 (380 tasks)	0.988047	0.980910	0.72%
50 (2450 tasks)	0.995212	0.981585	1.37%

Лекція 1

Розподілені обчислювальні системи:
визначення, класифікація,
архітектура, програмне забезпечення

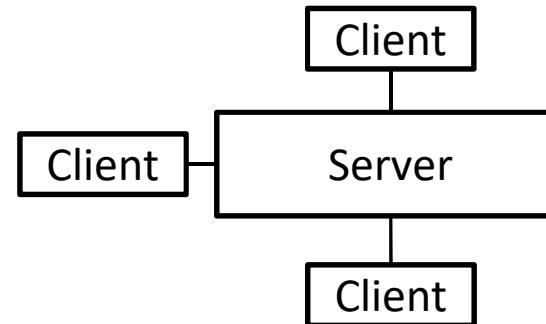
Розподілені обчислювальні системи

Розподілена обчислювальна система – набір обчислювальних вузлів (компонент), що координують свою роботу через обмін повідомленнями (синхронний чи асинхронний)

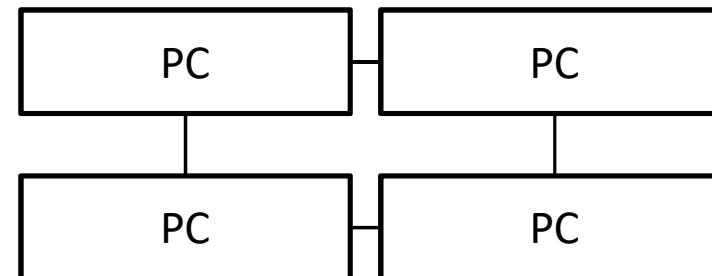
- Системи з розподіленими обчислювальними ресурсами
- Системи з розподіленою пам'яттю (розподілені бази даних)
 - ❖ Грід-системи
 - ❖ Хмарні обчислення

Архітектура розподілених систем

- Клієнт-серверна архітектура:



- Розподілена архітектура з рівноправними процесорами (peer to peer):

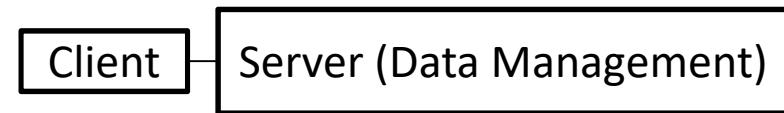


Клієнт-серверна рхітектура

– Тонкий клієнт



– Товстий клієнт

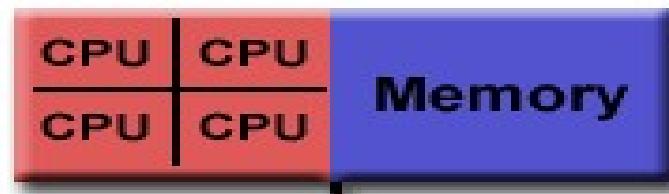


– Трирівнева
архітектура

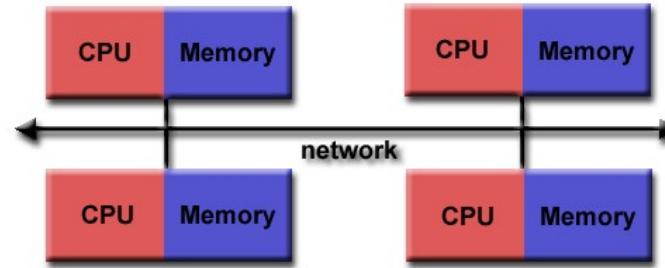


Архітектура з рівноправними процесорами

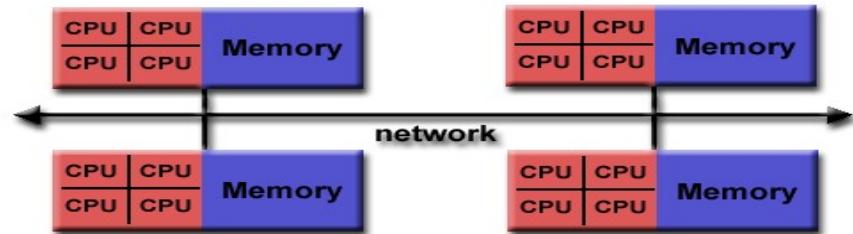
– Системи зі спільним модулем пам'яті



– Системи з розподіленими блоками пам'яті



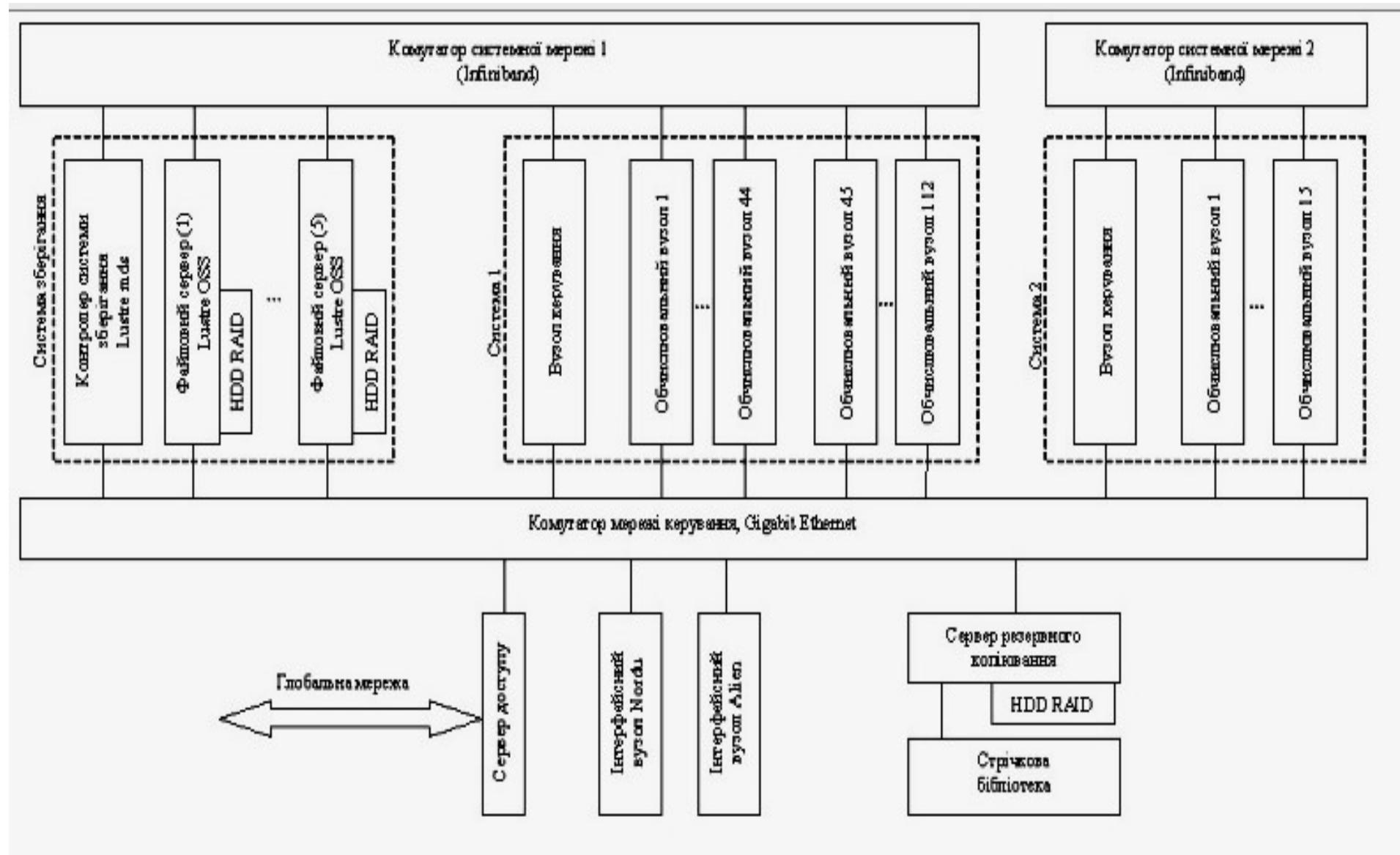
– Гібридні системи



Технології та програмне забезпечення розподілених систем, що вивчатимуться

- Стандарт обміну повідомленнями Message Passing Interface (MPI)
 - OpenMPI
 - MPJ Express
- Програмний інтерфейс виклику віддалених методів Java (Remote Method Invocation) RMI

Кластер Центру суперкомп'ютерних обчислень НТУУ «КПІ імені Ігоря Сікорського»



Кластер Центру суперком'ютерних обчислень НТУУ «КПІ імені Ігоря Сікорського»

Система 1:

Вузли: 44 з двома чотирьох-ядерними процесорами Intel Xeon E5440 @ 2.83ГГц та 8 Гб оперативної пам'яті у кожному

68 з двома двох-ядерними процесорами Intel Xeon 5160 @ 3.00ГГц та 4 Гб оперативної пам'яті у кожному

Мережа обміну даними: InfiniBand

Дисковий простір: 6 Тб на базі розподіленої файлової системи LustreFS

Продуктивність: пікова 7 ТФлопс, linpack 5.7 ТФлопс

ОС: CentOS release 6.4, **MPI:** OpenMPI 1.6.4, MVAPICH2 1.9rc1

Локальний менеджер ресурсів: Slurm, **Протокол доступу:** SSH

Система 2:

Вузли: 16 з двома чотирьох-ядерними процесорами Intel Xeon E5345 @ 2.33 ГГц, 8 Гб оперативної пам'яті та диском ємністю 500 Гб у кожному

Мережа обміну даними: InfiniBand

ОС: MS Windows Server 2008 HPC Edition, **MPI:** MS MPI 2.0.1551

Локальний менеджер ресурсів: HPC Job Manager, **Протокол доступу:** RDP

Материнські плати вузлів підтримують інтерфейс IPMI для віддаленого керування.

Система зберігання даних включає наступні складові:

6Тб + 6,5 Тб простору на базі системи зберігання FalconStor

20 Тб простору у стрічковому архіві

Лекція

Message Passing Interface (MPI)
*by Blaise Barney,
Lawrence Livermore National
Laboratory*

An Interface Specification

- MPI is a *specification* for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.
- MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.
- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
 - Practical
 - Portable
 - Efficient
 - Flexible
- The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.
- Interface specifications have been defined for C and Fortran90 language bindings:
 - C++ bindings from MPI-1 are removed in MPI-3
 - MPI-3 also provides support for Fortran 2003 and 2008 features
- Actual MPI library implementations differ in which version and features of the MPI standard they support. Developers/users will need to be aware of this.

Reasons for Using MPI

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability** - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.
- **Functionality** - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
- **Availability** - A variety of implementations are available, both vendor and public domain.

Documentation

- Documentation for all versions of the MPI standard is available at:

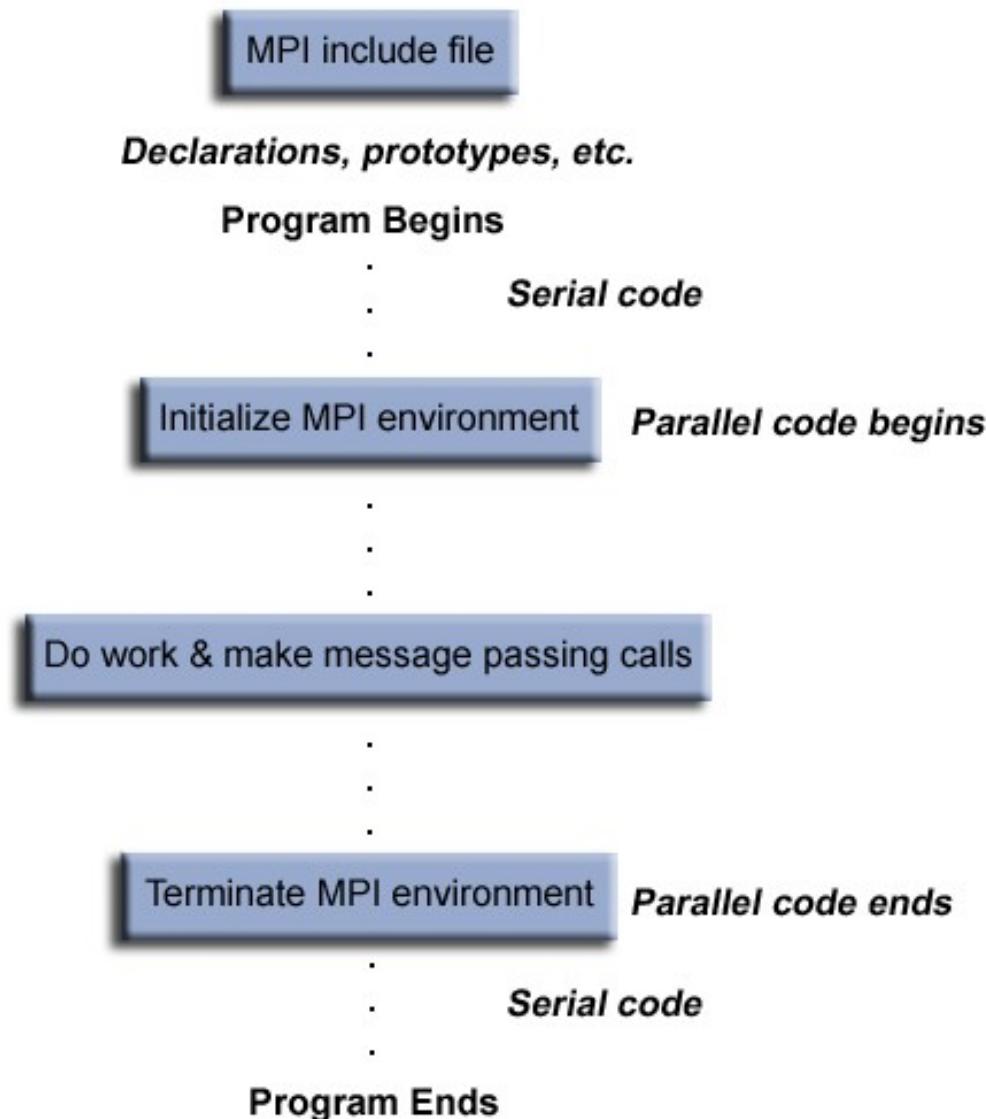
<http://www mpi-forum.org/docs/>

Open MPI

- Open MPI is a thread-safe, open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners.
- Open MPI is available on most LC Linux clusters. You'll need to load the desired dotkit package using the **use** command. For example:**use -l** (*list available packages*) **use openmpi-gnu-1.4.3** (*use the package of interest*)
- This ensures that LC's MPI wrapper scripts point to the desired version of Open MPI.
- More info about Open MPI in general:

www.open-mpi.org

General MPI Program Structure



Communicators and Groups

- MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.
- Most MPI routines require you to specify a communicator as an argument.
- Communicators and groups will be covered in more detail later. For now, simply use **MPI_COMM_WORLD** whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.

Rank

- Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero.
- Used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if $\text{rank}=0$ do this / if $\text{rank}=1$ do that).

Environment Management Routines

- [MPI Init](#)

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, `MPI_Init` may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

`MPI_Init (&argc,&argv)`
`MPI_INIT (ierr)`

- [MPI Comm size](#)

Returns the total number of MPI processes in the specified communicator, such as `MPI_COMM_WORLD`. If the communicator is `MPI_COMM_WORLD`, then it represents the number of MPI tasks available to your application.

`MPI_Comm_size (comm,&size)`
`MPI_COMM_SIZE (comm,size,ierr)`

- [MPI Comm rank](#)

Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator `MPI_COMM_WORLD`. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

`MPI_Comm_rank (comm,&rank)`
`MPI_COMM_RANK (comm,rank,ierr)`

- [**MPI Abort**](#)

Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

MPI_Abort (comm,errorcode)

MPI_ABORT (comm,errorcode,ierr)

- [**MPI Get processor name**](#)

Returns the processor name. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands.

MPI_Get_processor_name (&name,&resultlength)

MPI_GET_PROCESSOR_NAME (name,resultlength,ierr)

- [**MPI Get version**](#)

Returns the version and subversion of the MPI standard that's implemented by the library.

MPI_Get_version (&version,&subversion)

MPI_GET_VERSION (version,subversion,ierr)

- [**MPI Initialized**](#)

Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary. MPI_Initialized solves this problem.

MPI_Initialized (&flag)

MPI_INITIALIZED (flag,ierr)

- [**MPI_Wtime**](#)

Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

MPI_Wtime ()

MPI_WTIME ()

- [**MPI_Wtick**](#)

Returns the resolution in seconds (double precision) of MPI_Wtime.

MPI_Wtick ()

MPI_WTICK ()

- [**MPI_Finalize**](#)

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

MPI_Finalize ()

MPI_FINALIZE (ierr)

Hello, world

```
/** FILE: mpi_hello.c
 * DESCRIPTION: MPI tutorial example code: Simple hello world program
 • AUTHOR: Blaise Barney ***/
```

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define MASTER 0
int main (int argc, char *argv[]) {
    int numtasks, taskid, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    MPI_Get_processor_name(hostname, &len);
    printf ("Hello from task %d on %s!\n", taskid, hostname);
    if (taskid == MASTER)
        printf("MASTER: Number of MPI tasks is: %d\n", numtasks);
    MPI_Finalize();
}
```

Point to Point Communication Routines

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- There are different types of send and receive routines used for different purposes. For example:
 - **Blocking send / blocking receive**
 - **Non-blocking send / non-blocking receive**
 - **Buffered send**
 - **Synchronous send**
 - **"Ready" send**
 - **Combined send/receive**
- Any type of send routine can be paired with any type of receive routine.
- MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

Point to Point Communication Routines

MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

Blocking sends

`MPI_Send(buffer,count,type,dest,tag,comm)`

Non-blocking sends

`MPI_Isend(buffer,count,type,dest,tag,comm,request)`

Blocking receive

`MPI_Recv(buffer,count,type,source,tag,comm,status)`

Non-blocking receive

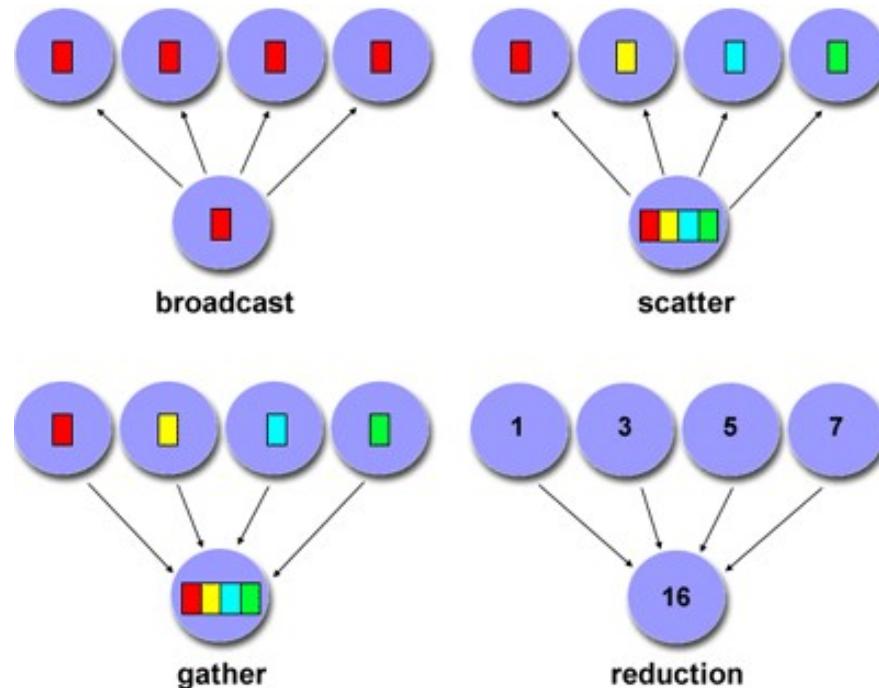
`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

Collective Communication Routines

- Collective communication routines must involve **all** processes within the scope of a communicator.
 - All processes are by default, members in the communicator `MPI_COMM_WORLD`.
 - Additional communicators can be defined by the programmer.
- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Collective Operations

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.
- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective Computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.



Collective Communication Routines

[MPI_Barrier](#)

Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed.

MPI_Barrier (comm)

MPI_BARRIER (comm,ierr)

[MPI_Bcast](#)

Data movement operation. Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

MPI_Bcast (&buffer,count,datatype,root,comm)

MPI_BCAST (buffer,count,datatype,root,comm,ierr)

[MPI_Scatter](#)

Data movement operation. Distributes distinct messages from a single source task to each task in the group.

MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf, recvbuf, recvtype,root,comm)

MPI_SCATTER (sendbuf,sendcnt,sendtype,recvbuf, recvbuf, recvtype,root,comm,ierr)

[MPI_Gather](#)

Data movement operation. Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf, recvbuf, recvtype,root,comm)

MPI_GATHER (sendbuf,sendcnt,sendtype,recvbuf, recvbuf, recvtype,root,comm,ierr)

[MPI_Allgather](#)

Data movement operation. Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.

MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf, recvbuf, recvtype,comm)

MPI_ALLGATHER (sendbuf,sendcount,sendtype,recvbuf, recvbuf, recvtype,comm,info)

[MPI_Reduce](#)

Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task.

MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)

MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)

Лекція

Блокуюча передача повідомлень
в MPI та реалізація паралельного
алгоритму множення матриць

Розрізняють:

- ✓ блокуючу передачу повідомлень,
- ✓ неблокуючу передачу повідомлень.

Передача повідомлень в MPI

- Передача повідомлення відбувається в три кроки:
 - 1) відправник читує дані з буфера та формує повідомлення;
 - 2) повідомлення передається від відправника до отримувача;
 - 3) отримувач записує дані з повідомлення до буфера.
- Функції відправки та прийому повідомлень MPI оперують масивами однотипних елементів, які називають **буферами**. Розмір буферів вказується не в байтах, а в елементах.
- Програміст має забезпечити, щоб фактичний тип змінних та тип, вказаний в аргументах функції MPI, збігались. Компілятор такої перевірки не виконує та не видає попереджень, якщо в аргументах функції MPI вказано тип, що не відповідає типу змінної.
- Програміст має забезпечити відповідність розміру буфера отримувача кількості даних, що надходять від відправника.

Передача повідомлень в MPI

- Тег – це ціле число від 0 до 32767, що використовують для того, щоб ідентифікувати повідомлення. Відправник встановлює тег повідомлення, а отримувач може обрати такий режим прийому, щоб отримувати тільки повідомлення з заданим тегом. `MPI_ANY_TAG` означає прийом повідомлення з будь-яким тегом.
- `MPI_ANY_SOURCE` означає прийом повідомлення від будь-якого процесу.
- Статус – посилання на структуру `MPI_STATUS`, в яку буде записано інформацію про прийняте повідомлення. Структура `MPI_STATUS` містить такі поля: `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`. Інформація про повідомлення може бути потрібна програмі, якщо під час прийому використовувались `MPI_ANY_TAG` або `MPI_ANY_SOURCE`, тобто отримувач не знає відправника або тегу повідомлення. `MPI_STATUS_IGNORE` означає, що програмі інформація про повідомлення не потрібна.

Блокуючий обмін повідомленнями

- Метод `MPI_Send` повертає управління програмі тоді, коли всі дані скопійовані з буфера відправника у буфер каналу зв'язку.
- В момент, коли метод `MPI_Send` повертає управління програмі, не гарантується, що повідомлення вже отримано отримувачем.
- Функція `MPI_Recv` обирає повідомлення за вказаними аргументами: ранг відправника, тег, комунікатор. Якщо відповідність встановлена, то повідомлення приймається повністю, а функція `MPI_Recv` повертає управління програмі.
- Коли `MPI_Recv` повернула управління програмі, гарантується, що всі дані, які отримуються, повністю скопійовані з каналу зв'язку в буфер отримувача.

Point to Point Communication: blocking send/receive

```
int MPI_Send(      void* buf,  
                  int count,  
                  MPI_Datatype datatype,  
                  int dest,  
                  int tag,  
                  MPI_Comm comm          );
```

```
int MPI_Recv(      void* buf,  
                  int count,  
                  MPI_Datatype datatype,  
                  int source,  
                  int tag,  
                  MPI_Comm comm,  
                  MPI_Status *status     );
```

Приклад: тестування передачі повідомень з заданим тегом

```
#include <stdio.h>
#include <mpi.h>
const int TAG_DATA1 = 10;
const int TAG_DATA2 = 20;
const int TAG_DATA3 = 25;
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0) {
        int x;
        MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, TAG_DATA3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Got %d with tag TAG_DATA3\n", x);
        MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, TAG_DATA1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Got %d with tag TAG_DATA1\n", x);
        MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, TAG_DATA2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Got %d with tag TAG_DATA2\n", x);
    }
    else if(rank == 1) {
        int x = 1000;
        MPI_Send(&x, 1, MPI_INT, 0, TAG_DATA1, MPI_COMM_WORLD);
    }
    else if(rank == 2) {
        int x = 2000;
        MPI_Send(&x, 1, MPI_INT, 0, TAG_DATA2, MPI_COMM_WORLD);
    }
    else {
        int x = 3000;
        MPI_Send(&x, 1, MPI_INT, 0, TAG_DATA3, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Паралельне обчислення суми ряду

за Стіренко С.Г. «Засоби паралельного програмування»

Нехай є p процесів.

Задача з номером id буде обчислювати члени ряду з номерами $id, p + id, 2p + id, \dots$

$$f(x_0) = \underbrace{a_1 + a_2 + \cdots + a_p}_{\text{крок 1}} + \underbrace{a_{p+1} + a_{p+2} + \cdots + a_{2p}}_{\text{крок 2}} + \dots$$

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

const double EPSILON = 1E-100; // Точність обчислення суми ряду
const int VALUE_TAG = 1; // Тег показника ступеня числа Е
const int NUM_TAG = 2; // Тег номера поточного члену ряду
const int TERM_TAG = 3; // Тег значення поточного члену ряду
const int BREAK_TAG = 4; // Тег завершення обчислень
const char *input_file_name = "in.txt";
const char *output_file_name = "out.txt";

double factorial(int value) { ... }

double calc_series_term(int term_number, double value) { ... }
```

Паралельне обчислення суми ряду

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank; /*id процесу*/
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int p; /* кількість процесів*/
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    double parametr; /* параметр суми ряду */
    if(rank == 0) /*MASTER*/
        FILE *input_file = fopen(input_file_name, "r");
        if(!input_file) {
            fprintf(stderr, "Can't open input file!\n\n");
            MPI_Abort(MPI_COMM_WORLD, 1);
            return 1;
        }
        fscanf(input_file, "%lf", &parametr);
        fclose(input_file);

        for(int id = 1; id < p; id++) {
            MPI_Send(&parametr, 1, MPI_DOUBLE, id, VALUE_TAG, MPI_COMM_WORLD);
        }
    }
    else /*WORKER*/
        MPI_Recv(&parametr, 1, MPI_DOUBLE, 0, VALUE_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Паралельне обчислення суми ряду

```
int last_num = 0;    /*номер останнього обчислениого члену ряду */
double sum = 0.0;   /*сума членів ряду */

for(int step = 0; step < 1000; step++) {
    int num;    /* номер члену ряду, що обчислюється */
    if(rank == 0) {/*MASTER*/
        num = last_num++;
        int curr_num = last_num;
        for(int id = 1; id < p; id++) {
            MPI_Send(&curr_num, 1, MPI_INT, id, NUM_TAG, MPI_COMM_WORLD);
            curr_num++;
        }
        last_num = curr_num;
    }
    else{ /*WORKER*/
        MPI_Recv(&num, 1, MPI_INT, 0, NUM_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    double term = calc_series_term(num, parametr);
    int need_break = false; /* умова «досягнуто необхідну точність» */
}
```

Паралельне обчислення суми ряду

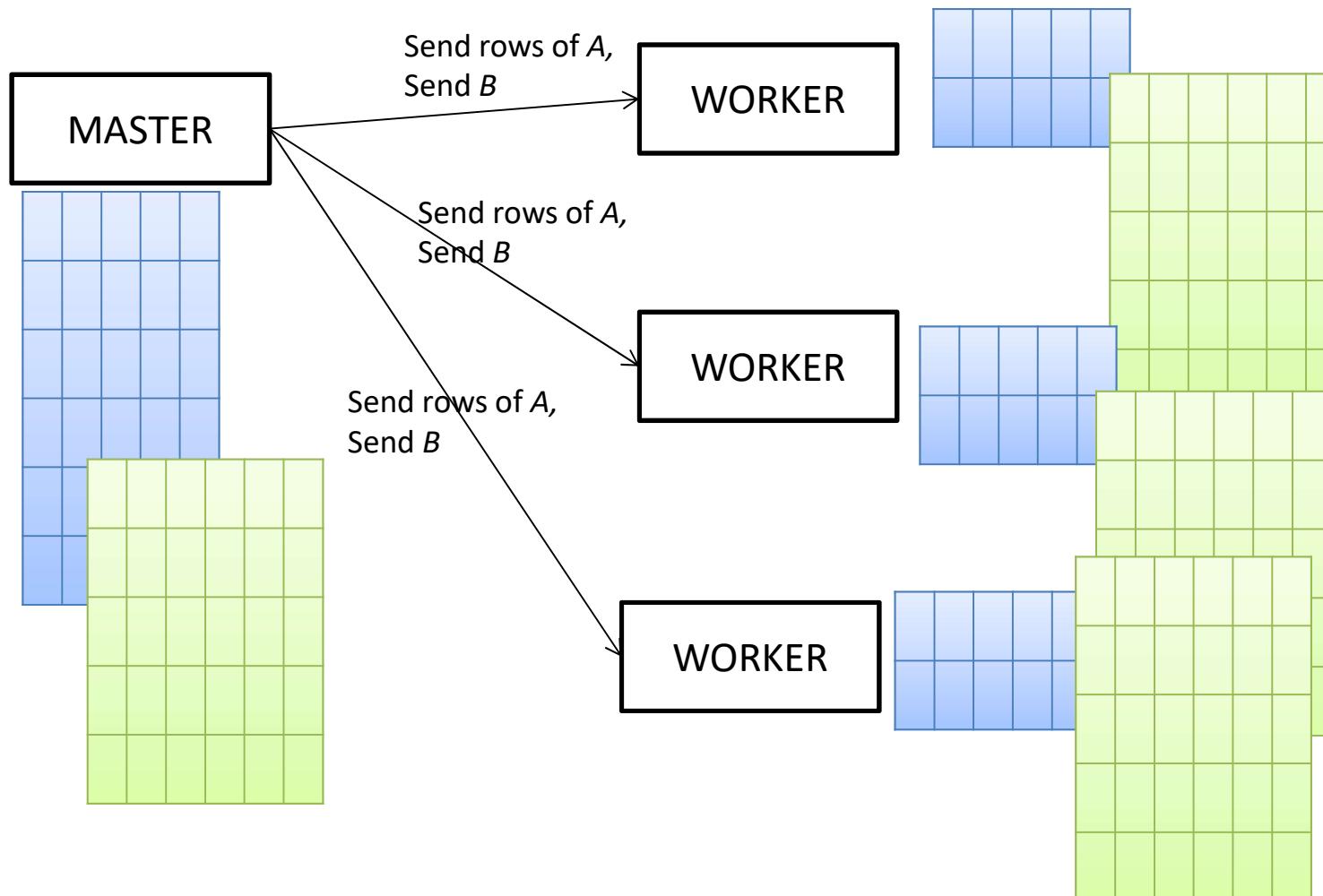
```
if(rank == 0) /*MASTER*/
    double curr_term = term;
    sum += curr_term;
    if(curr_term < EPSILON) {
        need_break = true;
    }
    for(int id = 1; id < p; id++) {
        MPI_Recv(&curr_term, 1, MPI_DOUBLE, id, TERM_TAG, MPI_COMM_WORLD, MPI_STATUS_
        sum += curr_term;
        if(curr_term < EPSILON) {
            need_break = true;
        }
        MPI_Send(&need_break, 1, MPI_INT, id, BREAK_TAG, MPI_COMM_WORLD);
    }
}
else { /*WORKER*/
    MPI_Send(&term, 1, MPI_DOUBLE, 0, TERM_TAG, MPI_COMM_WORLD);
    MPI_Recv(&need_break, 1, MPI_INT, 0, BREAK_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
}
if(need_break) {
    break;
}
/* завершення циклу for step*/
```

Паралельне обчислення суми ряду

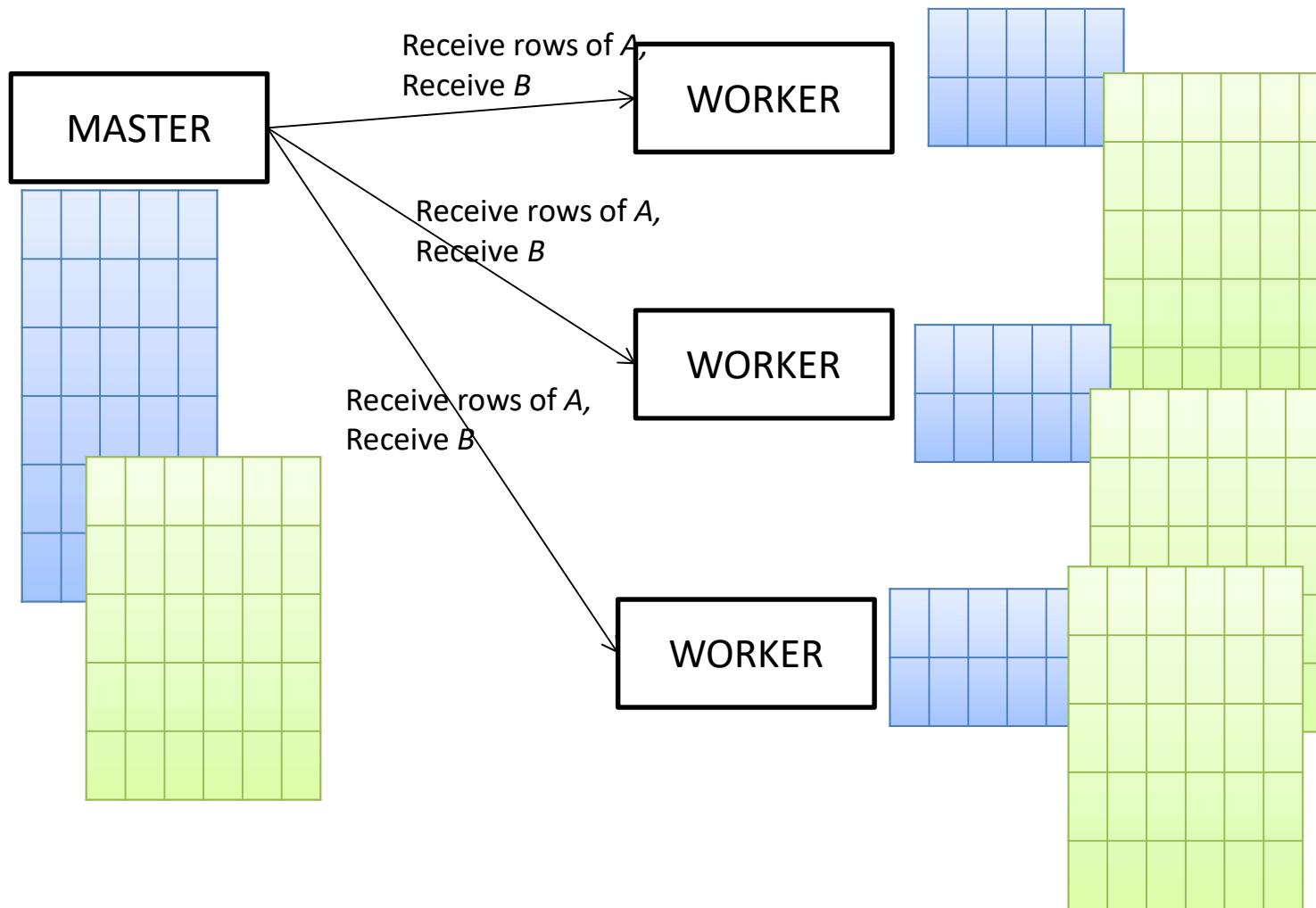
```
if(rank == 0) {  
    FILE *output_file = fopen(output_file_name, "w");  
    if(!output_file) {  
        fprintf(stderr, "Can't open output file!\n\n");  
        MPI_Abort(MPI_COMM_WORLD, 2);  
        return 2;  
    }  
    fprintf(output_file, "%.15lf\n", sum);  
}  
MPI_Finalize();  
return 0;  
}
```

Matrix multiply

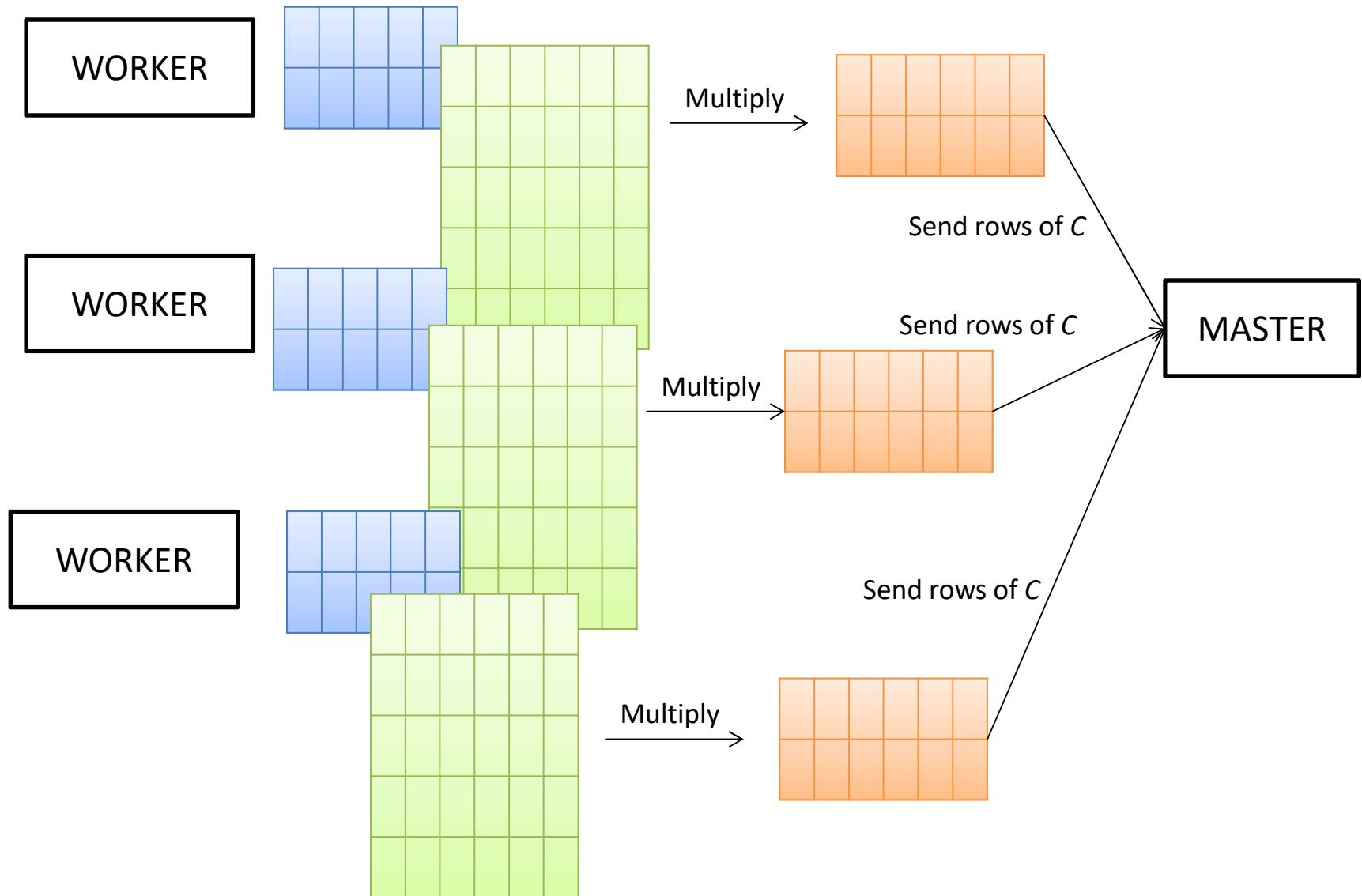
MASTER does:



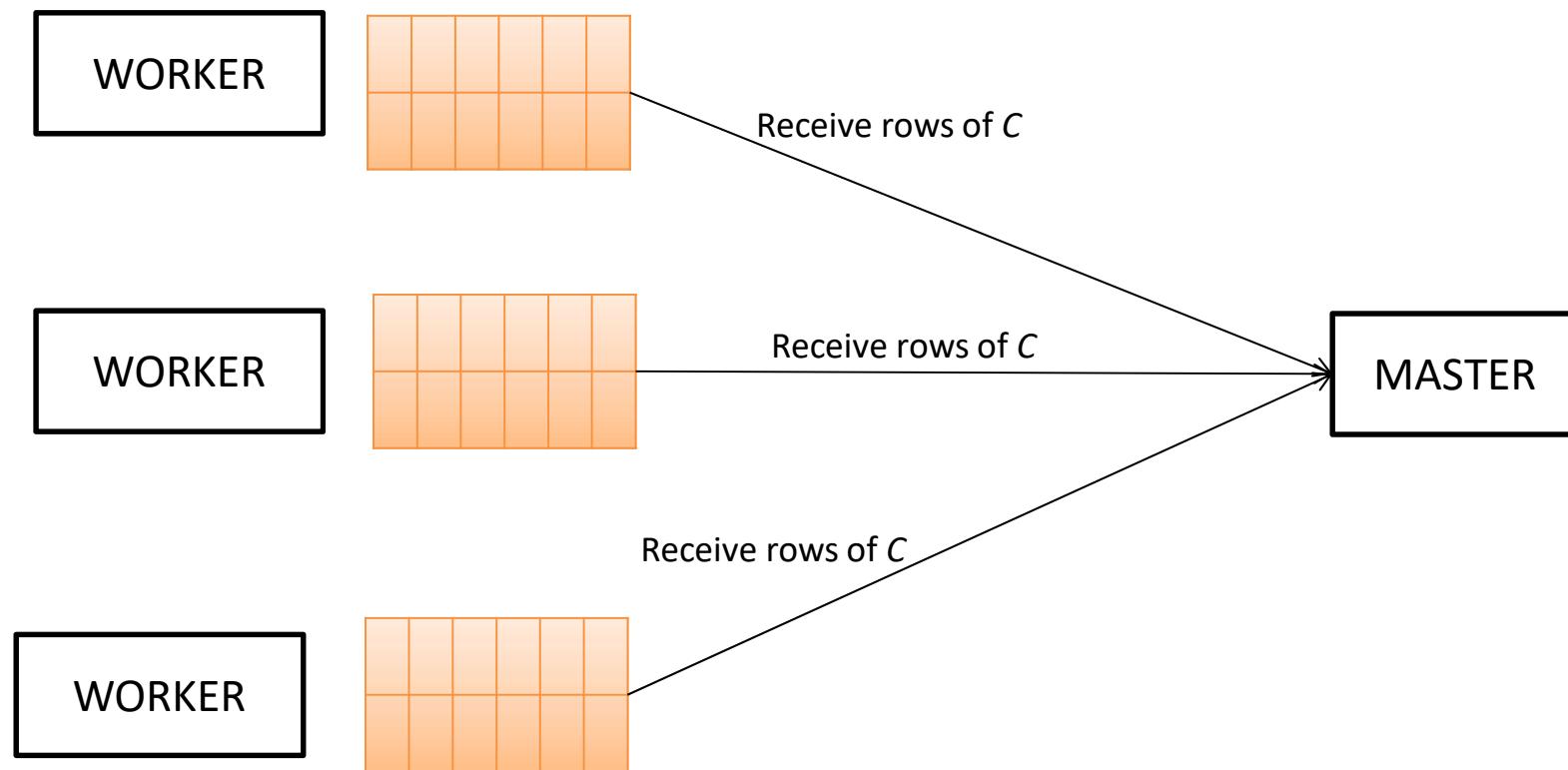
WORKERS do:



WORKERS do:



MASTER does:



Matrix multiply

```
/* FILE: mpi_mm.c  DESCRIPTION: MPI Matrix Multiply - C Version
 * In this code, the master task distributes a matrix multiply
 * operation to numtasks-1 worker tasks.
 * AUTHOR: Blaise Barney.
 * Adapted from Ros Leibensperger, Cornell Theory
 */
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define NRA 62           /* number of rows in matrix A */
#define NCA 15          /* number of columns in matrix A */
#define NCB 7           /* number of columns in matrix B */
#define MASTER 0         /* taskid of first task */
#define FROM_MASTER 1   /* setting a message type */
#define FROM_WORKER 2   /* setting a message type */
int main (int argc, char *argv[]) {
    int numtasks,
    taskid,
    numworkers,
    source,
    dest,
    mtype,             /* message type */
    rows,              /* rows of matrix A sent to each worker */
    averow, extra, offset, /* used to determine rows sent to each worker*/
    i, j, k, rc;
    double a[NRA][NCA], /* matrix A to be multiplied */
           b[NCA][NCB], /* matrix B to be multiplied */
           c[NRA][NCB]; /* result matrix C */
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numtasks );
    MPI_Comm_rank( MPI_COMM_WORLD, &taskid );
```

```

if (numtasks < 2 ) {
    printf("Need at least two MPI tasks. Quitting...\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
    exit(1);
}
numworkers = numtasks-1;
if (taskid == MASTER) {
    printf("mpi_mm has started with %d tasks.\n", numtasks);
    for (i=0; i<NRA; i++)
        for (j=0; j<NCA; j++)
            a[i][j]= 10;
    for (i=0; i<NCA; i++)
        for (j=0; j<NCB; j++)
            b[i][j]= 10;

    averow = NRA/numworkers;
    extra = NRA%numworkers;
    offset = 0;
    for (dest=1; dest<=numworkers; dest++) {
        rows = (dest <= extra) ? averow+1 : averow;
        printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);
        MPI_Send(&offset, 1, MPI_INT, dest, FROM_MASTER, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, dest, FROM_MASTER, MPI_COMM_WORLD);
        MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, FROM_MASTER,
                   MPI_COMM_WORLD);
        MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, FROM_MASTER, MPI_COMM_WORLD);
        offset = offset + rows;
    }
}

```

```
/* Receive results from worker tasks */
for (source=1; source<=numworkers; source++) {
    MPI_Recv(&offset, 1, MPI_INT, source, FROM_WORKER,
              MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, FROM_WORKER,
              MPI_COMM_WORLD, &status);
    MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, FROM_WORKER,
              MPI_COMM_WORLD, &status);
    printf("Received results from task %d\n", id);
}
/* Print results */
printf("****\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++) {
    printf("\n");
    for (j=0; j<NCB; j++)
        printf("%6.2f ", c[i][j]);
}
printf("\n*****\n");
printf ("Done.\n");
}
```

```

***** worker task *****/
else{ /* if (taskid > MASTER) */
    MPI_Recv(&offset, 1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD,
                           &status);
    MPI_Recv(&rows, 1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD, &status)
    MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, FROM_MASTER, MPI_COMM_WORLD,
                           &status)
    MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, FROM_MASTER, MPI_COMM_WORLD,
                           &status)

    for (k=0; k<NCB; k++)
        for (i=0; i<rows; i++) {
            c[i][k] = 0.0;
            for (j=0; j<NCA; j++)
                c[i][k] = c[i][k] + a[i][j] * b[j][k];
        }

        MPI_Send(&offset, 1, MPI_INT, MASTER, FROM_WORKER, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, MASTER, FROM_WORKER, MPI_COMM_WORLD);
        MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, FROM_WORKER, MPI_COMM_WORLD);
    }

    MPI_Finalize();
}

```

Лекція

Неблокуюча передача
повідомлень в MPI

Неблокуючий обмін повідомленнями

- Виконання програми не призупиняється для отримання чи відправлення даних. Такий спосіб при правильному використанні може зменшити втрату ефективності програми через повільну передачу повідомлень. Методи обміну повідомленнями починаються з літери I (Immediate).
- Метод MPI_Irecv не містить параметра status.
- Додатковий параметр request методів неблокуючого обміну повідомленнями означає дескриптор операції. Цей параметр можна передати методам MPI_Wait або MPI_Waitall, щоб примусити програму дочекатись завершення операції обміну повідомленнями.
- Усі буфери, що використовуються в неблокуючих обмінах повідомленнями, стають недоступними користувачу після запуску операції. Буфери даних стануть знову доступними після виконання методів MPI_Wait або MPI_Waitall для цих операцій обміну повідомленнями.
- !!! Вибір способу відправки повідомлень (блокуючий/неблокуючий) не впливає на вибір способу отримання. Повідомлення, які були відправлені **блокуючими** функціями, можна отримувати **неблокуючими** функціями, і навпаки.

Point to Point Communication: non-blocking send/receive

```
int MPI_Isend(          void *buf,  
                    int count,  
                    MPI_Datatype datatype,  
                    int dest,  
                    int tag,  
                    MPI_Comm comm,  
                    MPI_Request *request      );  
  
int MPI_Irecv(          void *buf,  
                    int count,  
                    MPI_Datatype datatype,  
                    int source,  
                    int tag,  
                    MPI_Comm comm,  
                    MPI_Request *request      );  
  
int MPI_Wait(           MPI_Request *request,  
                    MPI_Status *status       );  
  
int MPI_Waitall(        int count,  
                    MPI_Request array_of_requests[],  
                    MPI_Status array_of_statuses[] );
```

Приклад: тестування неблокуючого обміну повідомленнями

```
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
const int MY_TAG = 42;
int main(int argc, char* argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0) {
        sleep(5);
        int data[] = { 1, 2, 3, 4, 5};
        MPI_Request send_req;
        MPI_Isend(&data, 5, MPI_INT, 1, MY_TAG, MPI_COMM_WORLD, &send_req);
        sleep(5); /* Процес 0 виконує дії, які не змінюють data */
        MPI_Wait(&send_req, MPI_STATUS_IGNORE);
    }
    if(rank == 1){/*Процес 1 починає отримувати дані раніше ніж процес 0 почав їх передавати*/
        int data[5];
        MPI_Request recv_req;
        MPI_Irecv(&data, 5, MPI_INT, 0, MY_TAG, MPI_COMM_WORLD, &recv_req);
        sleep(2); /* Процес 1 виконує дії, які не використовують data */
        MPI_Wait(&recv_req, MPI_STATUS_IGNORE);
        for(int i = 0; i < 5; i++) {
            printf(" %d", data[i]);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Інші методи неблокуючого обміну повідомленнями

- Використовуються літери B, S, or R для позначення buffered, synchronous або ready режиму обміну повідомленнями:

```
int MPI_IBsend(), int MPI_IBrecv(),
int MPI_ISsend(), int MPI_ISrecv(),
int MPI_IRsend(), int MPI_IRrecv(),
```

Лекція

Методи колективного обміну
повідомленнями в MPI та їх
застосування для матричних
обчислень

Методи взаємодії процесів

- Рівноправний (peer to peer)
- Майстер-робітник (master/slave)

Метод MPI_Scatter

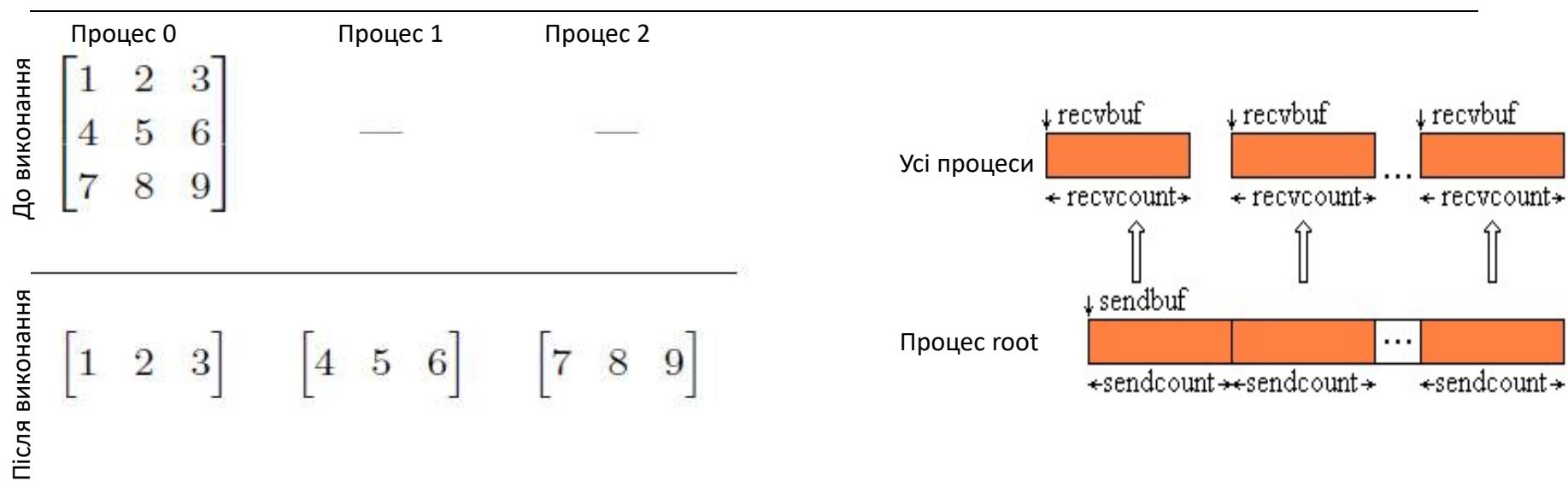
```
MPI_Scatter( void *sendbuf,  
             int sendcnt,  
             MPI_Datatype sendtype,  
             void *recvbuf,  
             int recvcnt,  
             MPI_Datatype recvtype,  
             int root, /* процес-відправник*/  
             MPI_Comm comm);
```

- Для виконання розсилки даних метод потрібно викликати в кожному процесі з однаковими значеннями `root` та `comm` та відповідними значеннями `recvcnt` та `recvtype`. Аргументи `sendbuf`, `sendcnt`, `sendtype` необхідно вказувати тільки для процеса-відправника даних.
- Перші три аргументи в процесах, що приймають дані, ігноруються, а кількість та тип отримуваних елементів, номер процесу-відправника та комунікатор збігаються.

Приклад:

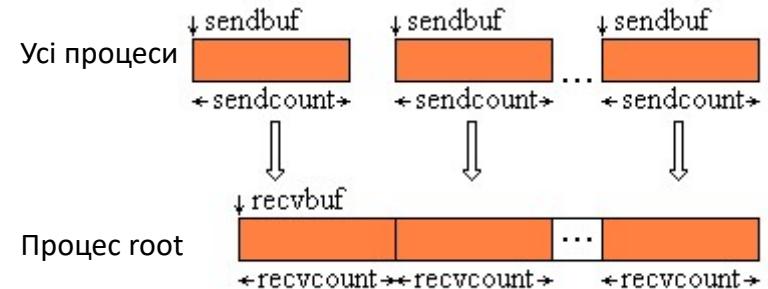
```
/*MASTER*/
MPI_Scatter(    MA,  k*N, MPI_INT,
                 MAh, k*N, MPI_INT,
                 0, MPI_COMM_WORLD);

/*WORKER*/
MPI_Scatter(    NULL, 0, MPI_DATATYPE_NULL,
                 MAh, k*N, MPI_INT,
                 0, MPI_COMM_WORLD);
```



Метод MPI_Gather

```
MPI_Gather( void *sendbuf,  
            int sendcnt,  
            MPI_Datatype sendtype,  
            void *recvbuf, int recvcnt,  
            MPI_Datatype recvtype,  
            int root, MPI_Comm comm);
```

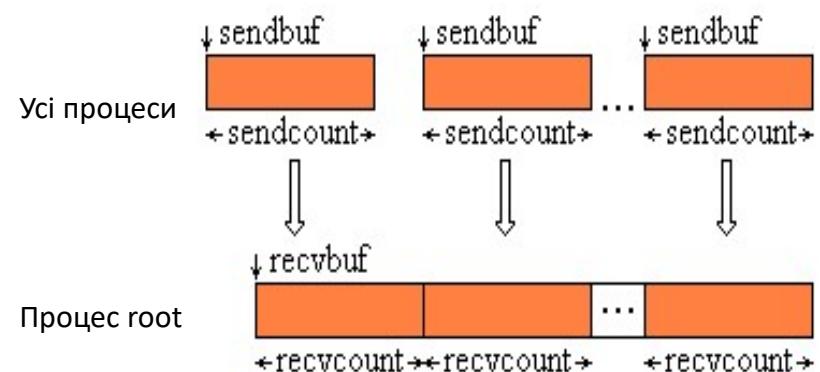


- Параметри `recvbuf`, `recvcnt`, `recvtype` мають значення тільки для задачі, що отримує дані. Буфер відправки в кожній задачі має бути розміром не менше за `sendcnt`, буфер прийому в приймаючій задачі не менше за `recvcnt` Р, де Р - кількість задач.
- Метод може використовувати один і той самий буфер як буфер відправки та прийому у відправляючій задачі. Для цього необхідно аргумент `sendbuf` встановити рівним константі `MPI_IN_PLACE`. При цьому аргументи `sendcnt` та `sendtype` ігноруються. Бажано дотримуватися попередніх рекомендацій щодо встановлення значень ігнорованих аргументів.

Приклад:

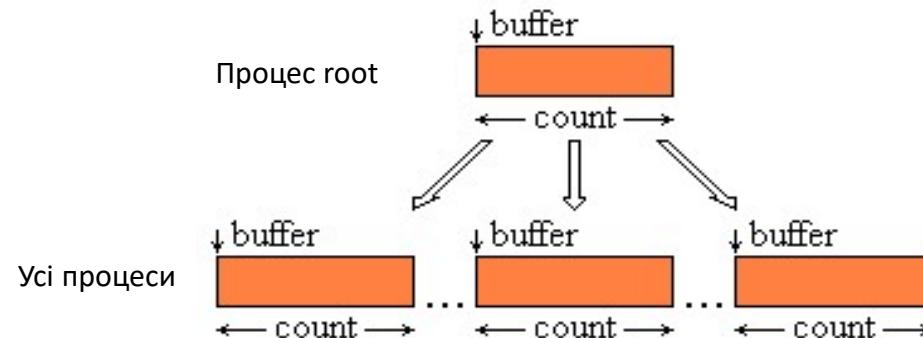
```
/*MASTER*/  
MPI_Gather(      MAh,  k*N, MPI_INT,  
                  MR,  k*N, MPI_INT,  
                  0, MPI_COMM_WORLD);  
  
/*WORKER*/  
MPI_Gather(      MAh,  k*N, MPI_INT,  
                  NULL, 0, MPI_INT,  
                  0, MPI_COMM_WORLD);
```

Процес 0	Процес 1	Процес 2
[1 2 3]	[4 5 6]	[7 8 9]
Після виконання	—	—



Метод MPI_Bcast

```
MPI_Bcast ( void *buffer,  
            int count,  
            MPI_Datatype type,  
            int root,  
            MPI_Comm comm)
```



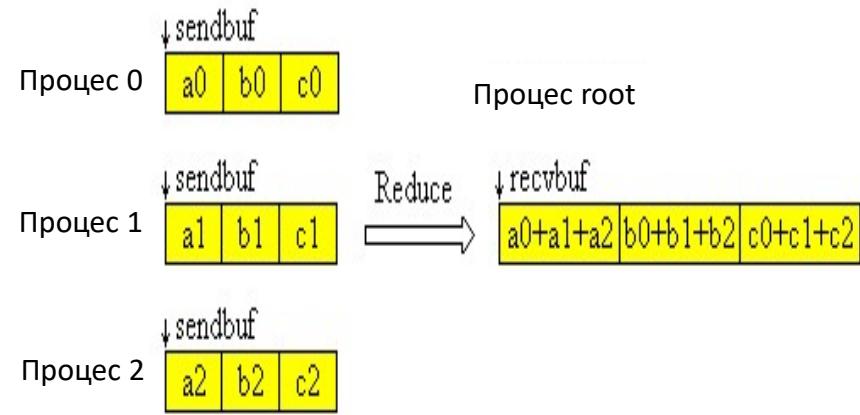
Приклад:

```
MPI_Bcast( &MR, N*N, MPI_INT,  
           1, MPI_COMM_WORLD);
```

Процес 0	Процес 1	Процес 2
До виконання	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$	
Після виконання	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

Метод MPI_Reduce

```
MPI_Reduce ( void *sendbuf,  
              void *recvbuf,  
              int count,  
              MPI_Datatype datatype,  
              MPI_Op op,  
              int root,  
              int comm)
```



Операції:

MPI_MAX	Максимум	int, float
MPI_MIN	Мінімум	int, float
MPI_SUM	Сума	int, float
MPI_PROD	Добуток	int, float
MPI_LAND	Логічне «І»	int
MPI_BAND	Побітове «І»	int
MPI_LOR	Логічне «АБО»	int
MPI_BOR	Побітове «АБО»	int
MPI_LXOR	Логічне виключне «АБО»	int
MPI_BXOR	Побітове виключне «АБО»	int

Приклад:

```
/*MASTER*/  
MPI_Reduce( MA, MR, N*N,  
             MPI_INT, MPI_SUM,  
             0, MPI_COMM_WORLD);  
  
/*WORKER*/  
MPI_Reduce( MA, NULL, N*N,  
             MPI_INT, MPI_SUM,  
             0, MPI_COMM_WORLD);
```

	Процес 0	Процес 1	Процес 2
До виконання	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	$\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$	$\begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}$
Після виконання	<hr/> $\begin{bmatrix} 15 & 18 \\ 21 & 24 \end{bmatrix}$		

Метод MPI_AllReduce

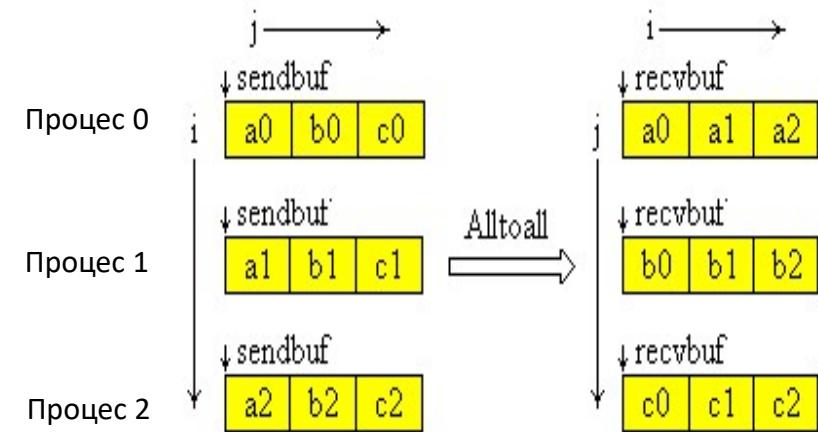
```
MPI_Allreduce(      void *sendbuf,  
                    void *recvbuf,  
                    int count,  
                    MPI_Datatype datatype,  
                    MPI_op op,  
                    MPI_Comm comm);
```

Приклад: MPI_Allreduce(MA, MR,
 N*N, MPI_INT,
 MPI_SUM, MPI_COMM_WORLD);

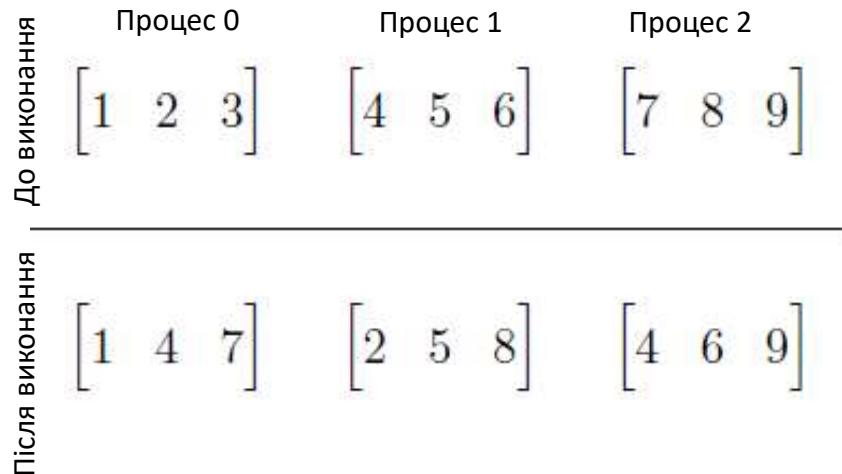
	Процес 0	Процес 1	Процес 2
До виконання	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	$\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$	$\begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}$
<hr/>			
Після виконання	$\begin{bmatrix} 15 & 18 \\ 21 & 24 \end{bmatrix}$	$\begin{bmatrix} 15 & 18 \\ 21 & 24 \end{bmatrix}$	$\begin{bmatrix} 15 & 18 \\ 21 & 24 \end{bmatrix}$

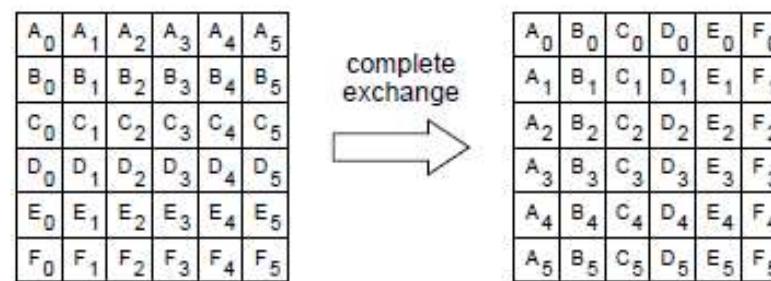
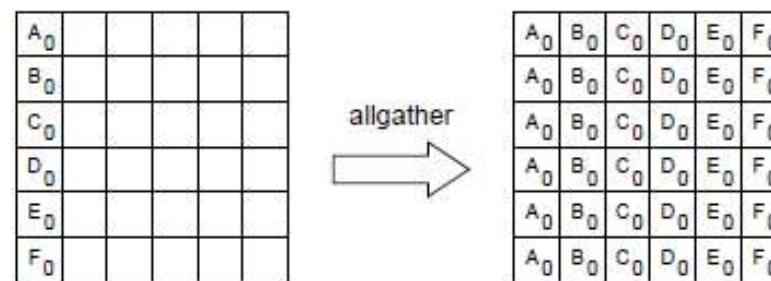
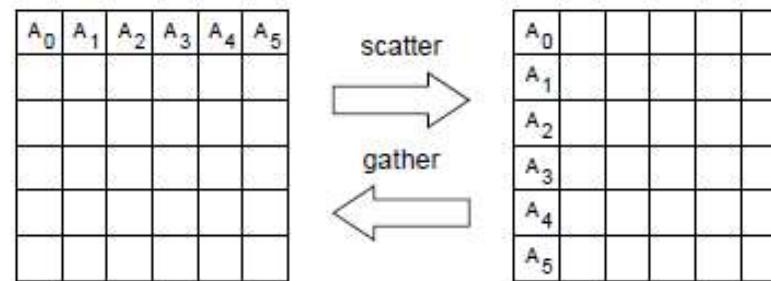
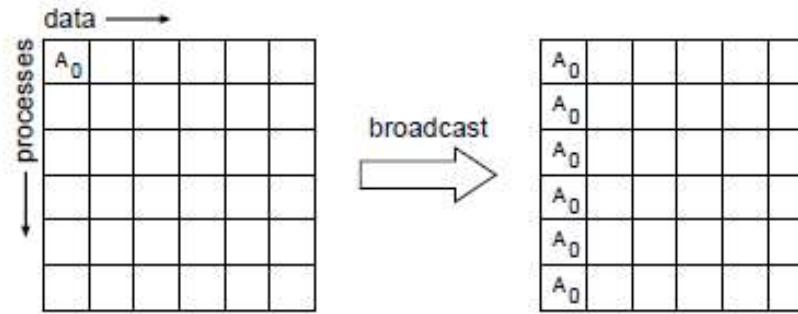
Метод MPI_Alltoall

```
MPI_Alltoall (void *sendbuf,  
              int sendcount,  
              MPI_Datatype sendtype,  
              void *recvbuf,  
              int recvcount,  
              MPI_Datatype recvtype,  
              MPI_Comm comm);
```



Приклад: MPI_Alltoall(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
 MA, 1, MPI_INT, MPI_COMM_WORLD);





Example 1

Gather 100 ints from every process in group to root.

```
MPI_Comm comm;  
int gsize, sendarray[100];  
int root, *rbuf;  
...  
MPI_Comm_size(comm, &gsize);  
rbuf = (int *)malloc(gsize*100*sizeof(int));  
MPI_Gather(sendarray, 100, MPI_INT,  
            rbuf, 100, MPI_INT,  
            root, comm);
```

Example 2

Previous example modified | only the root allocates memory for the receive buer.

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank(comm, &myrank);
if (myrank == root) {
    MPI_Comm_size(comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100,
           MPI_INT, root, comm);
```

Example 3

Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of $gsize * 100$ ints since type matching is denied pairwise between the root and each process in the gather.

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size(comm, &gsize);
MPI_Type_contiguous(100, MPI_INT, &rtype);
MPI_Type_commit(&rtype);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1,
           rtype, root, comm);
```

Example 4

Now have each process send 100 ints to root, but place each set (of 100) stride ints apart at receiving end. Use MPI_GATHERV and the displs argument to achieve this effect. Assume stride≥100.

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf, stride;
int *displs, i, *rcounts;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs,
MPI_INT,
root, comm);
```

Example 5

Same as Example 5.5 on the receiving side, but send the 100 ints from the 0th column of a 100x150 int array, in C.

```
MPI_Comm comm;
int gsize, sendarray[100][150];
int root, *rbuf, stride;
MPI_Datatype stype;
int *displs, i, *rcounts;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
/* Create datatype for 1 column of array */
MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
MPI_Gatherv(sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
root, comm);
```

Example 6

Process i sends (100-i) ints from the i-th column of a 100 150 int array, in C.
It is received into a buer with stride, as in the previous two examples.

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs, i, *rcounts;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100-i; /* note change from previous example */
}
/* Create datatype for the column we are sending */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
/* sptr is the address of start of "myrank" column
*/
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT, root, comm);
```

Example 7

The reverse of Example 5.2. Scatter sets of 100 ints from the root to each process in the group.

```
MPI_Comm comm;  
int gsize,*sendbuf;  
int root, rbuf[100];  
...  
MPI_Comm_size(comm, &gsize);  
sendbuf = (int *)malloc(gsize*100*sizeof(int));  
...  
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100,  
MPI_INT, root, comm);
```

Example 8

The starting order of collective operations on a particular communicator defines their matching. The following example shows an erroneous matching of different collective operations on the same communicator

```
MPI_Request req;
switch(rank) {
    case 0:
        /* erroneous matching */
        MPI_Ibarrier(comm, &req);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
    case 1:
        /* erroneous matching */
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Ibarrier(comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
}
```

Example 9

Collective and point-to-point requests can be mixed in functions that enable multiple completions. If started with two processes, the following program is valid

```
MPI_Request reqs[2];
switch(rank) {
    case 0:
        MPI_Ibarrier(comm, &reqs[0]);
        MPI_Send(buf, count, dtype, 1, tag, comm);
        MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
        break;
    case 1:
        MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
        MPI_Ibarrier(comm, &reqs[1]);
        MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
        break;
}
```

Example 10

Multiple nonblocking collective operations can be outstanding on a single communicator and match in order

```
MPI_Request reqs[3];
compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
compute(buf3);
MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
```

Current Practice #2

```
int main(int argc, char *argv[]) {
    int me, count;
    void *data;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if(me == 0) {
        /* get input, create buffer ``data'' */
        ...
    }
    MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
    return 0;
}
```

Example. Create group of processors

```
int main(int argc, char *argv[]){
    int me, count, count2;
    void *send_buf, *recv_buf, *send_buf2, *re
    MPI_Group group_world, grpren;
    MPI_Comm commslave;
    static int ranks[] = {0};
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);
    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
    MPI_Group_excl(group_world, 1, ranks, &grpren); /* local */
    MPI_Comm_create(MPI_COMM_WORLD, grpren, &commslave);
    if(me != 0) {
        /* compute on slave */
        ...
        MPI_Reduce(send_buf, recv_buf, count, MPI_INT, MPI_SUM, 1, commslave);
        ...
        MPI_Comm_free(&commslave);
    }
    /* zero falls through immediately to this reduce, others do later... */
    MPI_Reduce(send_buf2, recv_buf2, count2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Group_free(&group_world);
    MPI_Group_free(&grpren);
    MPI_Finalize();
    return 0;
}
```

Example #4

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT 50
int main(int argc, char *argv[]) {
    int me;
    MPI_Request request[2];
    MPI_Status status[2];
    MPI_Group group_world, subgroup;
    int ranks[] = {2, 4, 6, 8};
    MPI_Comm the_comm;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);
    MPI_Group_incl(group_world, 4, ranks, &subgroup); /* local */
    MPI_Group_rank(subgroup, &me); /* local */
    MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
    if(me != MPI_UNDEFINED) {
        MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                  the_comm, request);
        MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                  the_comm, request+1);
        for(i = 0; i < SOME_COUNT; i++)
            MPI_Reduce(..., the_comm);
        MPI_Waitall(2, request, status);
        MPI_Comm_free(&the_comm);
    }
    MPI_Group_free(&group_world);
    MPI_Group_free(&subgroup);
    MPI_Finalize();
    return 0;
}
```

Programming Parallel Hardware using MPJ Express

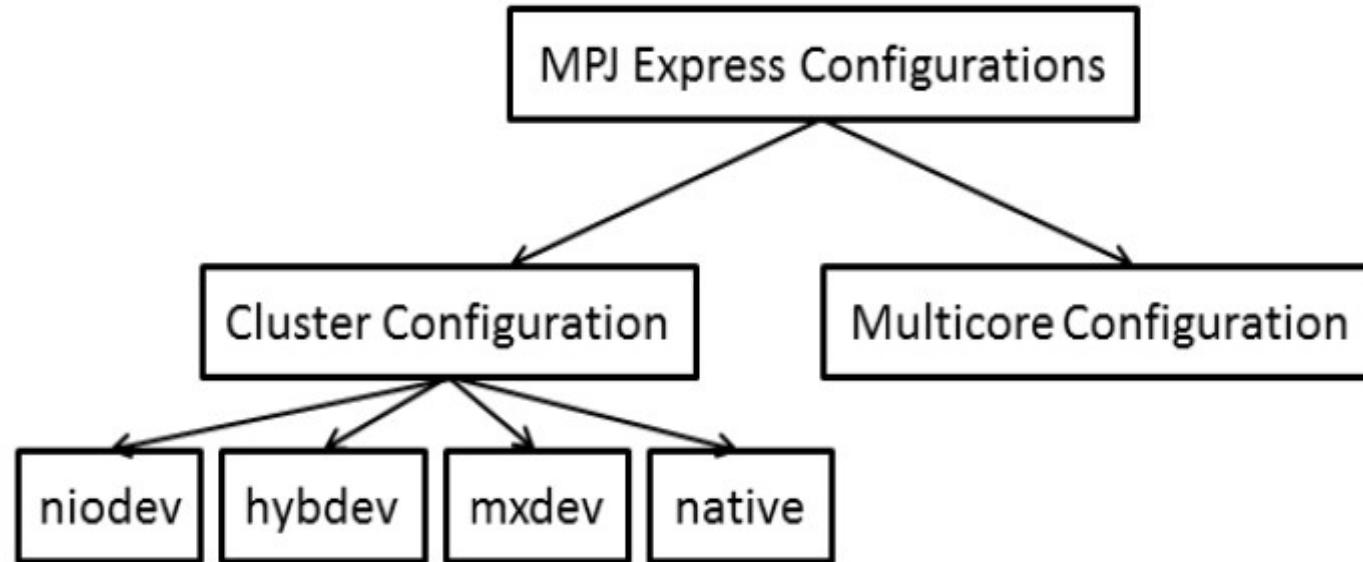
By A. Shafi

Why Java?

- Portability
- A popular language in colleges and software industry:
 - Large pool of software developers
 - A useful educational tool
- Higher programming abstractions including OO features
- Improved compile and runtime checking of the code
- Automatic garbage collection
- Support for multithreading
- Rich collection of support libraries

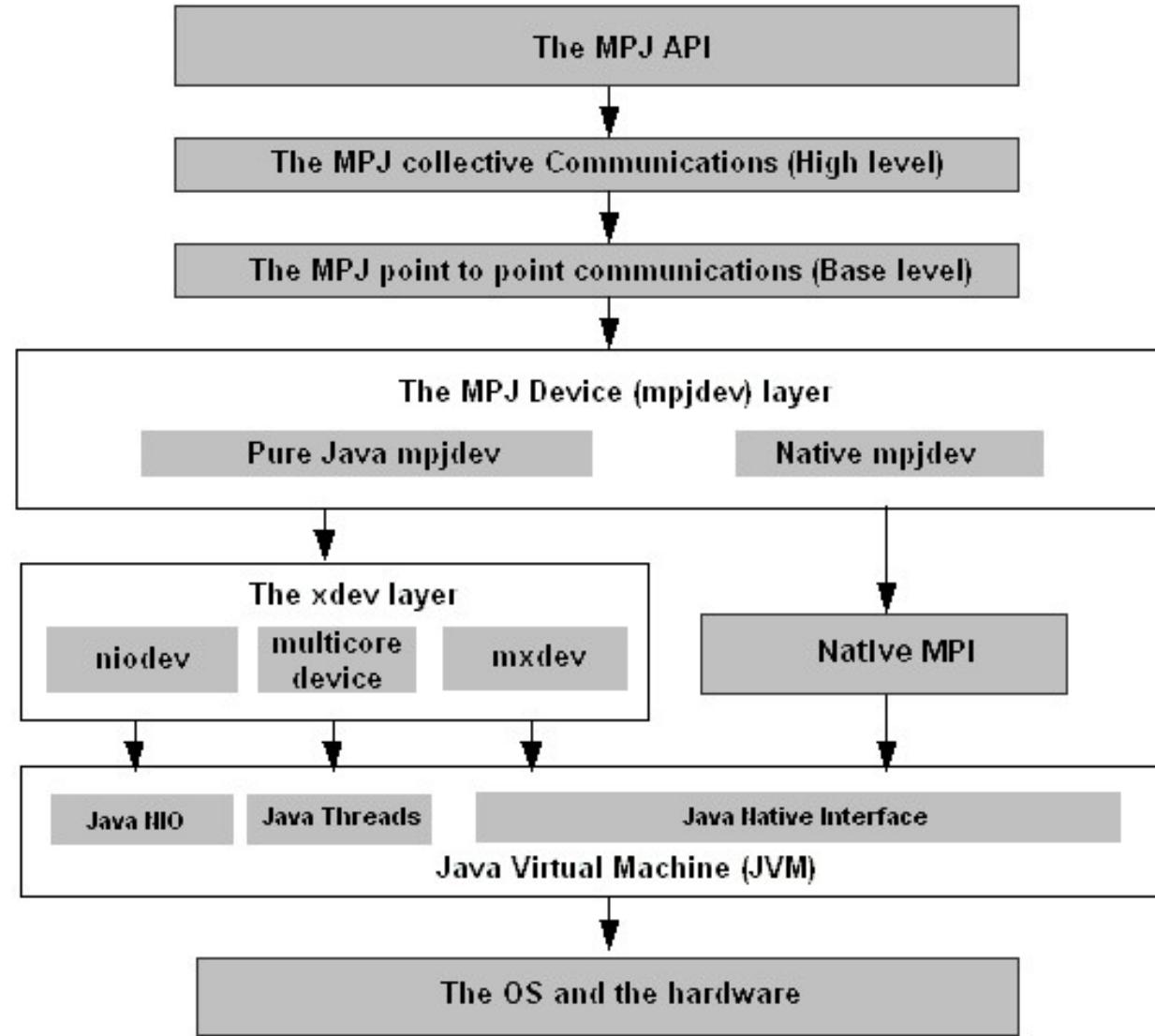
- MPJ Express is an MPI-like library that supports execution of parallel Java applications
- Three existing approaches to Java messaging:
 - Pure Java (Sockets based)
 - Java Native Interface (JNI)
 - Remote Method Invocation (RMI)
- Motivation for a new Java messaging system:
 - Maintain compatibility with Java threads by providing thread-safety
 - Handle contradicting issues of high-performance and portability
 - Requires no change to the native standard JVM

MPJ Express configuration

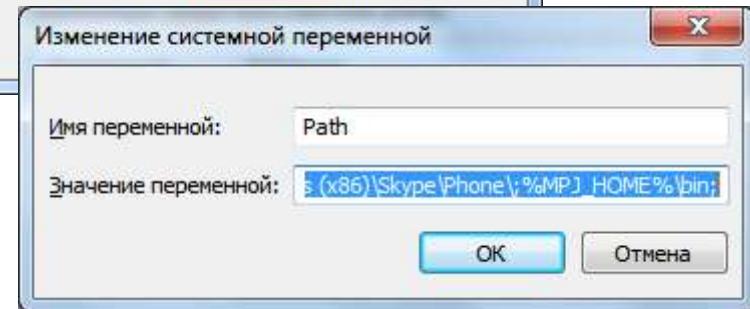
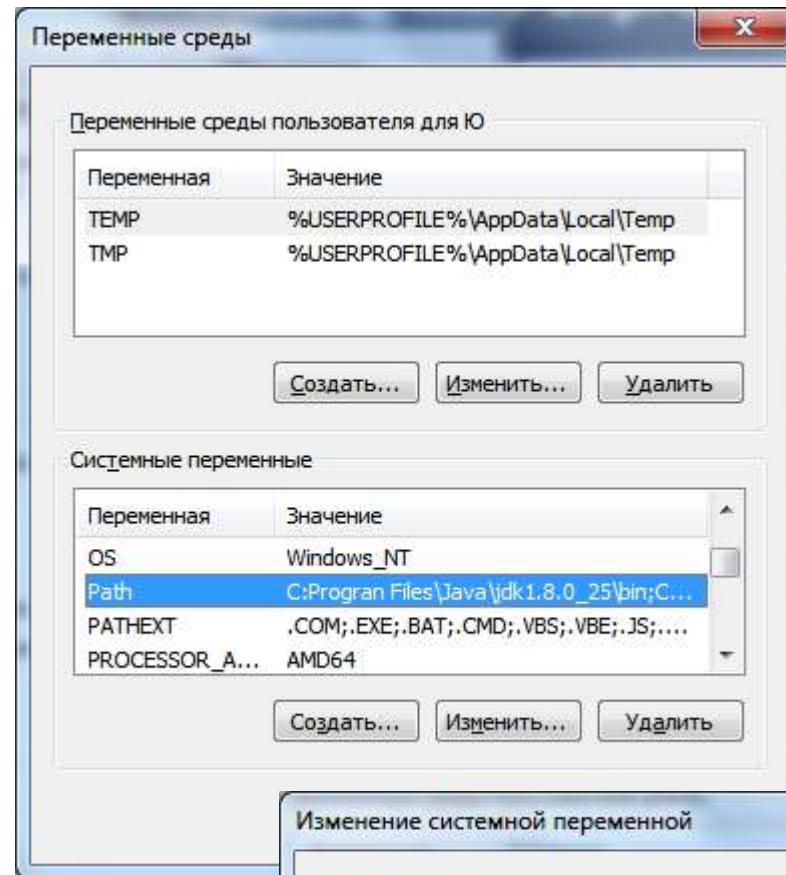
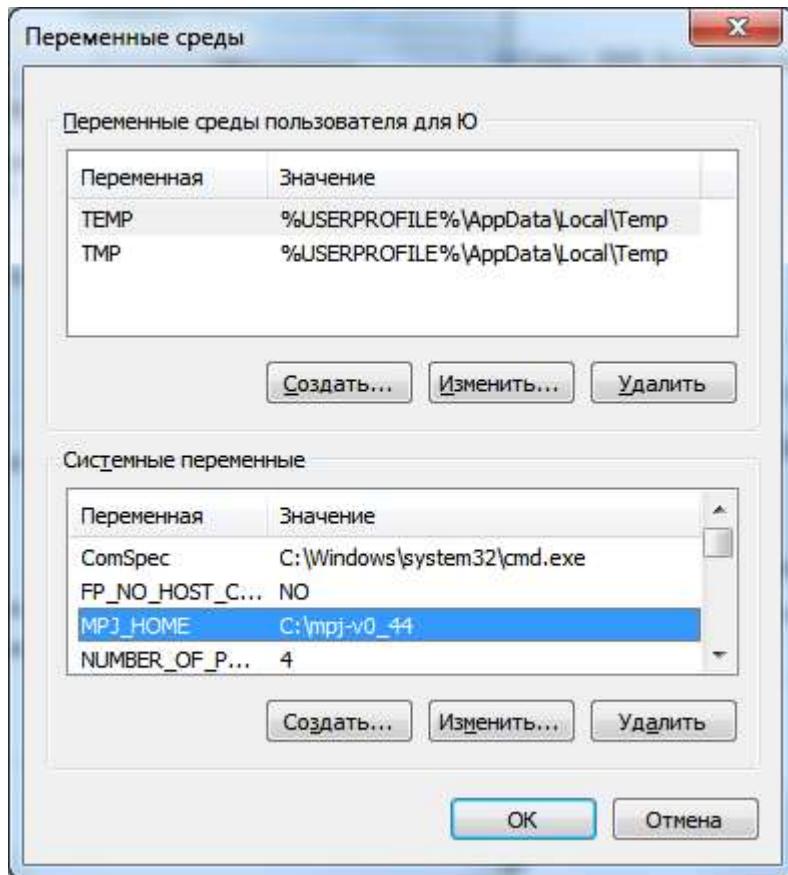


1. Java New I/O (NIO) device known as `niodev`: `niodev` is used to execute MPJ Express user programs on clusters using Ethernet.
2. Myrinet device known as `mxdev`: `mxdev` is used to execute MPJ Express user programs on clusters connected by Myrinet express interconnects. Currently `mxdev` is not supported under windows.
3. Hybrid device known as `hybdev`: `hybdev` is used to execute MPJ Express user programs on clusters of multicore computers.
4. Native device known as `native`: `native` is used to execute MPJ Express user programs on top of a native MPI library (MPICH, Open MPI or MS-MPI).

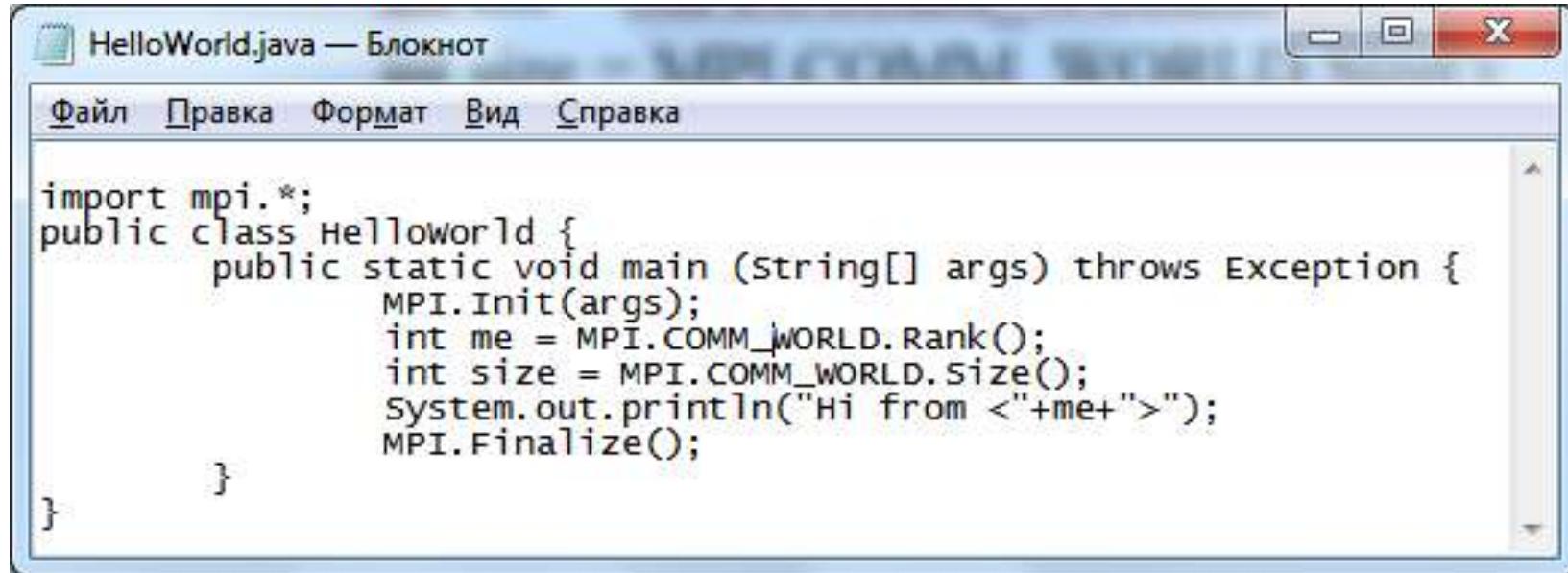
MPJ Express Design



Installing MPJ Express



HelloWorld program

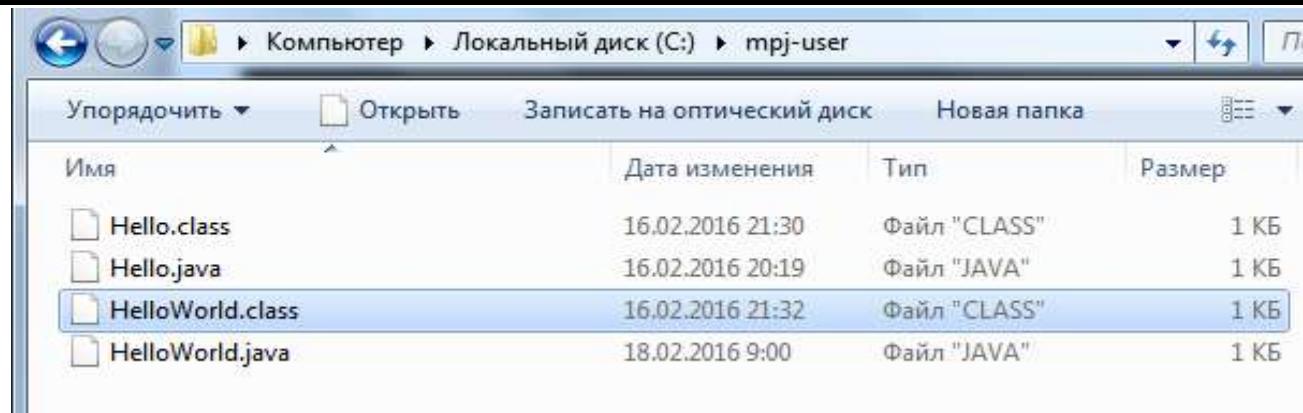


The screenshot shows a Windows Notepad window with the title "HelloWorld.java — Блокнот". The menu bar includes "Файл", "Правка", "Формат", "Вид", and "Справка". The code in the editor is:

```
import mpi.*;
public class HelloWorld {
    public static void main (String[] args) throws Exception {
        MPI.Init(args);
        int me = MPI.COMM_WORLD.Rank();
        int size = MPI.COMM_WORLD.Size();
        System.out.println("Hi from <" + me + ">");
        MPI.Finalize();
    }
}
```

HelloWorld program

```
C:\mpj-user>"C:\Program Files\Java\jdk1.8.0_25\bin\javac.exe" -classpath .;%MPJ_HOME%\lib\mpj.jar HelloWorld.java
```



```
C:\mpj-user>mpjrun.bat -np 2 HelloWorld
MPJ Express <0.44> is started in the multicore configuration
Hi from <0>
Hi from <1>
```

```
C:\mpj-user>mpjrun.bat -np 4 HelloWorld
MPJ Express <0.44> is started in the multicore configuration
Hi from <2>
Hi from <3>
Hi from <0>
Hi from <1>
```

Documentation: class Comm



mpjexpress.org/docs/javadocs/mpi/Comm.html

Class Comm

java.lang.Object
mpi.Comm

Direct Known Subclasses:

Intercomm, Intracomm

```
public class Comm
extends java.lang.Object
```

Field Summary

Fields

Modifier and Type	Field and Description
Group	group
mpjdev.Comm	mpjdevComm

Method Summary

Methods

Modifier and Type	Method and Description
void	Abort(int errorcode) Abort MPI.
void	Attr_delete(int keyval) Retrieves attribute value by key.
java.lang.Object	Attr_get(int keyval) Retrieves attribute value by key.
Prequest	Bsend_init(java.lang.Object buf, int offset, int count, Datatype datatype, int dest, int tag) Creates a persistent communication request for a buffered mode send.
void	Bsend(java.lang.Object buf, int offset, int count, Datatype datatype, int dest, int tag) Send in buffered mode.
protected void	bsend(java.lang.Object buf, int offset, int count, Datatype datatype, int dest, int tag, boolean pt2p)

Documentation: Send() method

Send

```
public void Send(java.lang.Object buf,  
                 int offset,  
                 int count,  
                 Datatype datatype,  
                 int dest,  
                 int tag)  
    throws MPIException
```

Blocking send operation.

buf	send buffer array
offset	initial offset in send buffer
count	number of items to send
datatype	datatype of each item in send buffer
dest	rank of destination
tag	message tag

Java binding of the MPI operation MPI_SEND.

The actual argument associated with `buf` must be one-dimensional array. The value `offset` is a subscript in this array, defining the position of the first item of the message.

If the `datatype` argument represents an MPI basic type, its value must agree with the element type of `buf`—either a primitive type or a reference (object) type. If the `datatype` argument represents an MPI derived type, its *base type* must agree with the element type of `buf`.

Throws:

`MPIException`

Documentation: Recv() method

Recv

```
public Status Recv(java.lang.Object buf,  
                  int offset,  
                  int count,  
                  Datatype datatype,  
                  int source,  
                  int tag)  
    throws MPIException
```

Blocking receive operation.

buf	receive buffer array
offset	initial offset in receive buffer
count	number of items in receive buffer
datatype	datatype of each item in receive buffer
source	rank of source
tag	message tag
returns:	status object

Java binding of the MPI operation MPI_RECV.

The actual argument associated with `buf` must be one-dimensional array. The value `offset` is a subscript in this array, defining the position into which the first item of the incoming message will be copied.

If the `datatype` argument represents an MPI basic type, its value must agree with the element type of `buf`—either a primitive type or a reference (object) type. If the `datatype` argument represents an MPI derived type, its `base type` must agree with the element type of `buf`.

Throws:

MPIException

```

import mpi.*;

public class ToyExample {

  public static void main(String[] args) throws Exception {
    MPI.Init(args); int rank = MPI.COMM_WORLD.Rank(); int size = MPI.COMM_WORLD.Size() ;
    int unitSize=4, tag=100, master=0;

    if(rank == master) { /* master */

      int sendbuf[] = new int[unitSize*(size-1)];

      for(int i=1; i<size; i++)
        MPI.COMM_WORLD.Send(sendbuf, (i-1)*unitSize, unitSize, MPI.INT, i, tag);

      for(int i=1; i<size; i++)
        MPI.COMM_WORLD.Recv(sendbuf, (i-1)*unitSize, unitSize, MPI.INT, i, tag);

      for(int i=0 ; i<unitSize*(size-1) ; i++)
        System.out.print(sendbuf[i]+" ");

    } else { /* worker */

      int recvbuf[] = new int[uniSize];
      MPI.COMM_WORLD.Recv(recvbuf, 0, unitSize, MPI.INT, master, tag);

      for(int i=0 ; i<unitSize; i++) recvbuf[i] = rank; /* computation loop */

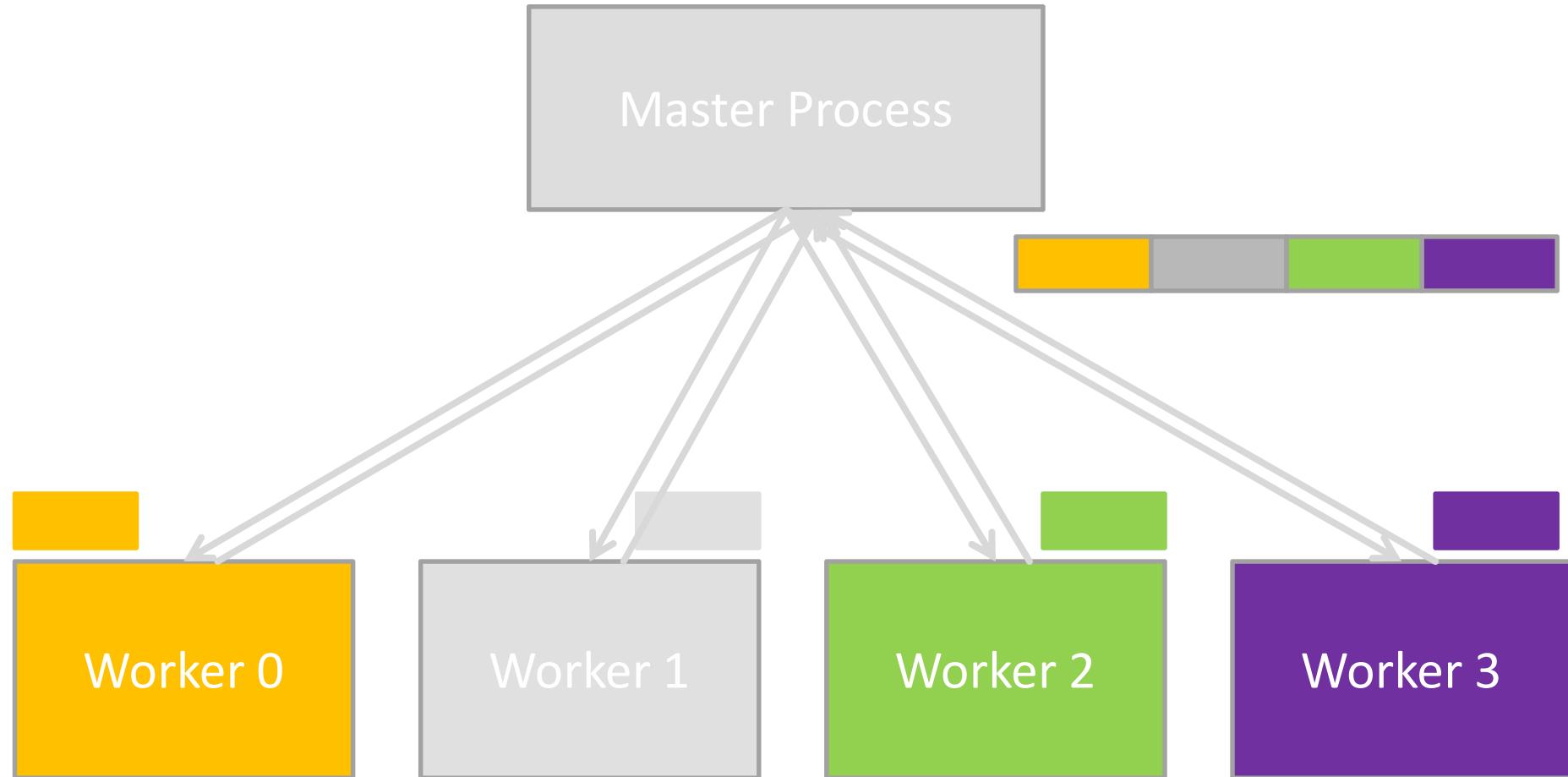
      MPI.COMM_WORLD.Send(recvbuf, 0, unitSize, MPI.INT, master, tag);
    }

    MPI.Finalize();
  }
}
  
```

aamirshafi@velour:~/work/mpj-user\$
 mpjrun.sh -np 5 ToyExample
 MPJ Express (0.38) is started in the
 multicore configuration

1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4

An Embarrassingly Parallel Toy Example

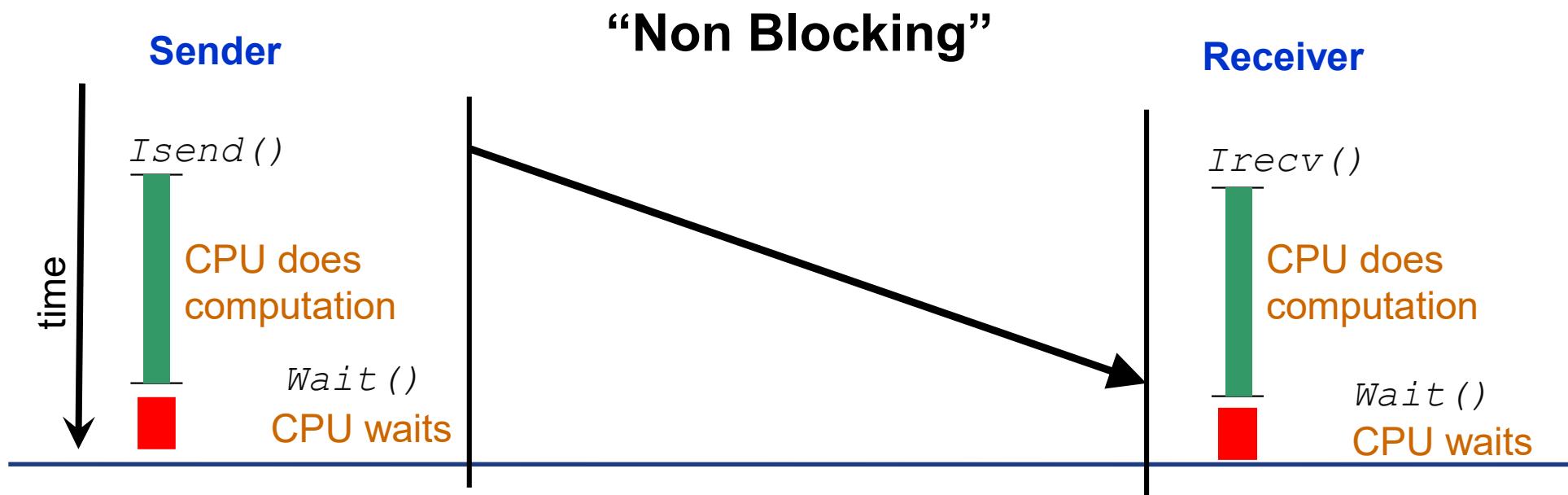
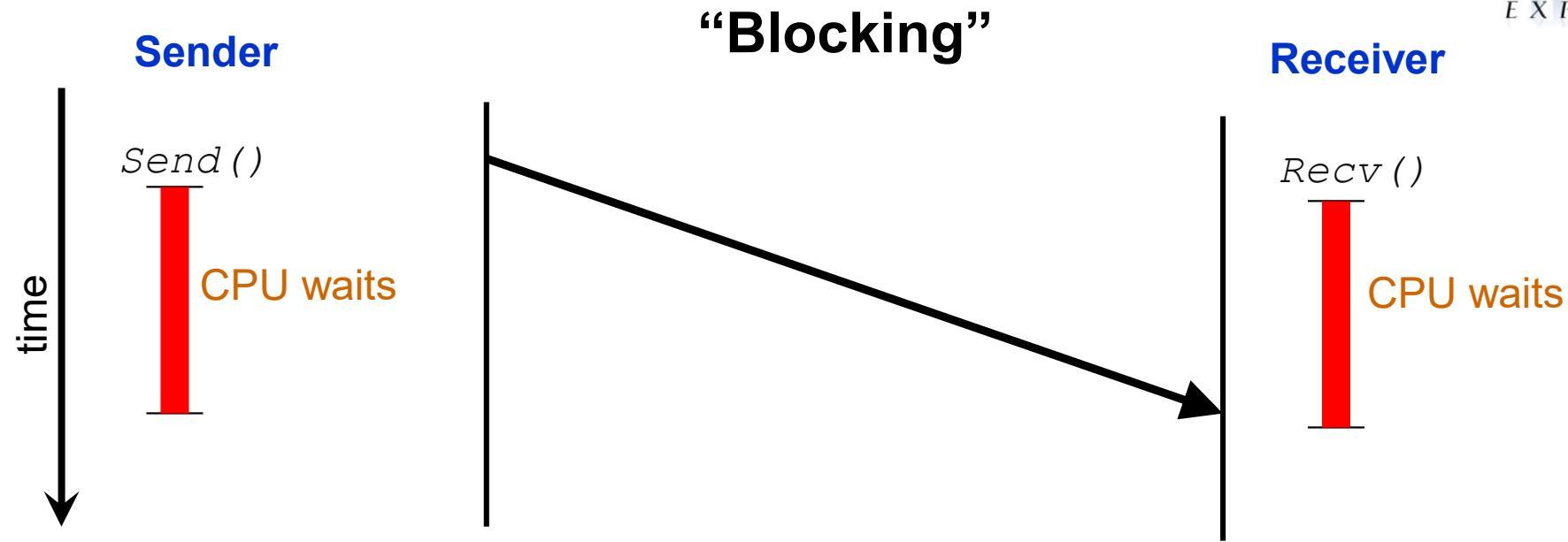


Point-to-point Communication

	Standard	Synchronous	Ready	Buffered
Blocking	Send() Recv()	Ssend()	Rsend()	Bsend()
Non-blocking	Isend() Irecv()	Issend()	Irsend()	Ibsend()

Non-blocking methods return a Request object:

- Wait() //waits until communication completes
- Test() //test if the communication has finished



Thread-safe Communication

- Thread-safe MPI libraries allow communication from multiple user threads inside a single process
- Such an implementation requires fine-grain locking:
 - Incorrect implementations can deadlock

Levels of Thread-Safety in MPI Libraries

`MPI_THREAD_SINGLE` : only one thread will execute

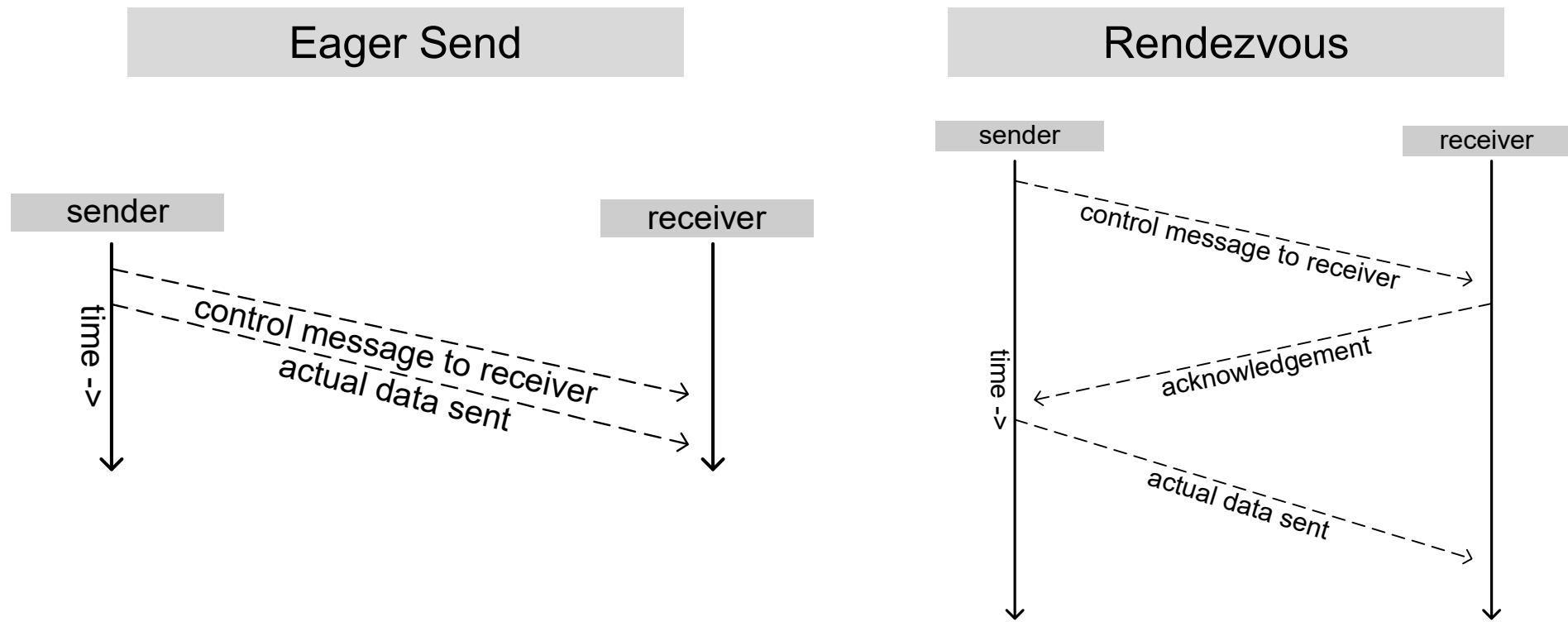
`MPI_THREAD_FUNNELED` : may be multi-threaded, but only the main thread will make MPI calls

`MPI_THREAD_SERIALIZED` : may be multi-threaded, but only the one thread will make MPI calls at a time

`MPI_THREAD_MULTIPLE` : multiple threads may call MPI with no restrictions

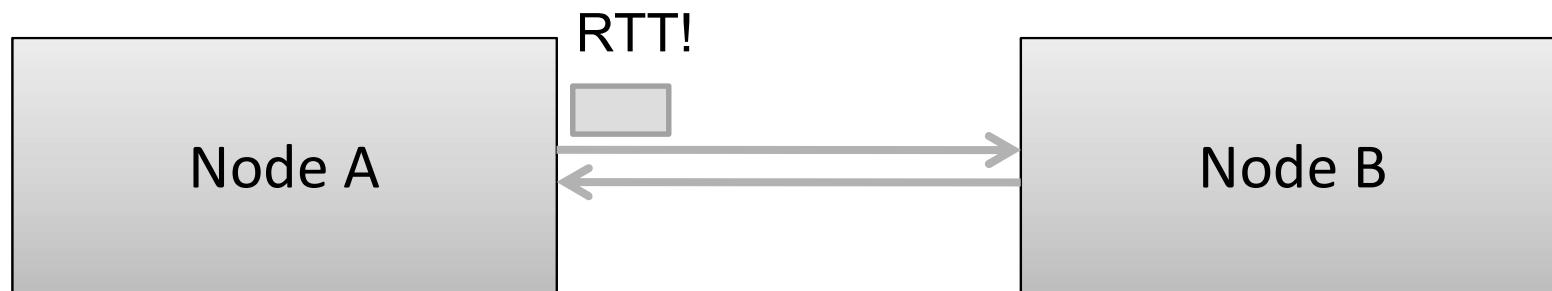
Implementation of point-to-point communication

- Various modes of blocking and non-blocking communication primitives are implemented using two protocols



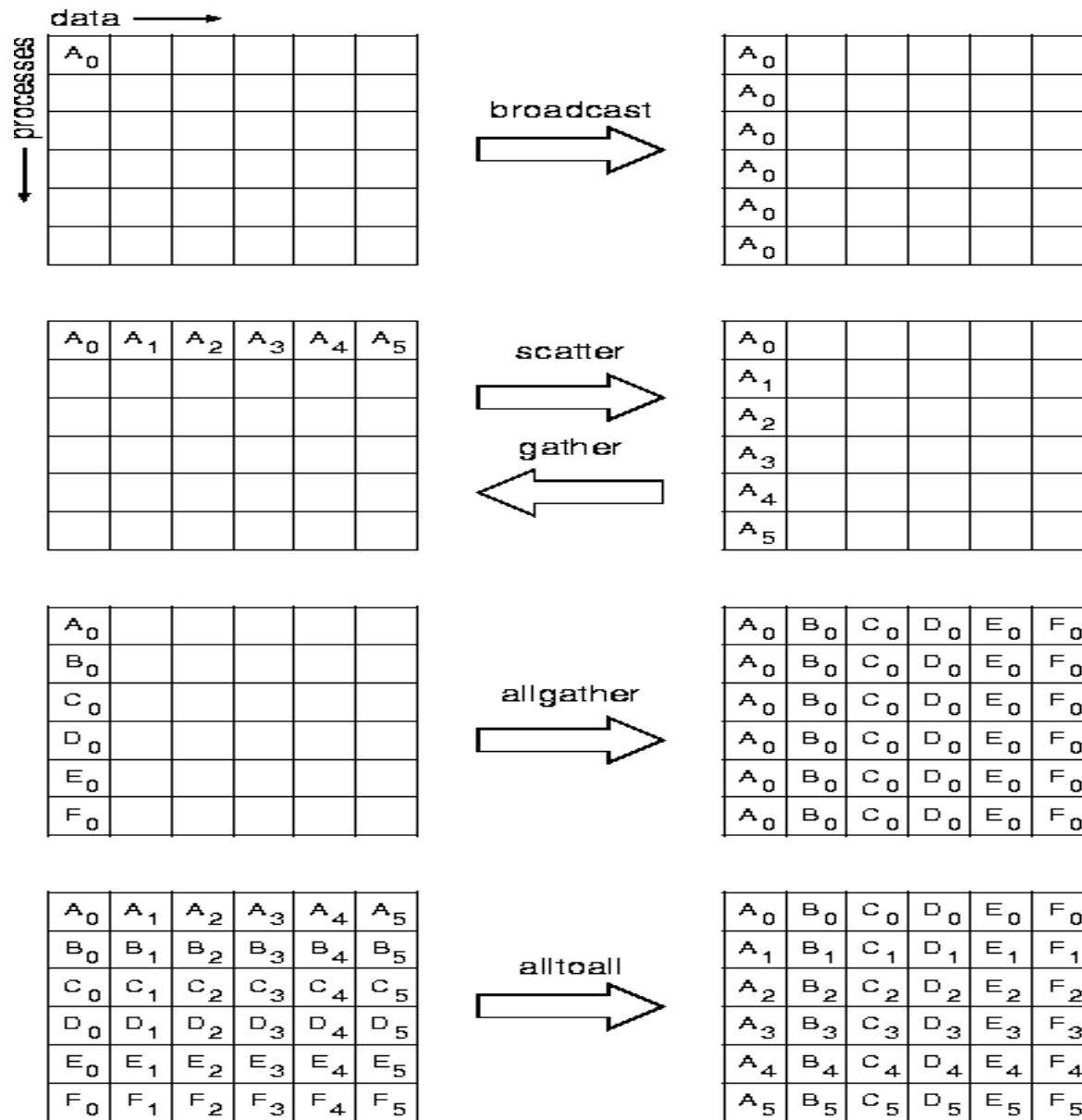
Performance Evaluation of Point to Point Communication

- Normally ping pong benchmarks are used to calculate:
 - *Latency*: How long it takes to send N bytes from sender to receiver?
 - *Throughput*: How much bandwidth is achieved?
- Latency is a useful measure for studying the performance of “small” messages
- Throughput is a useful measure for studying the performance of “large” messages

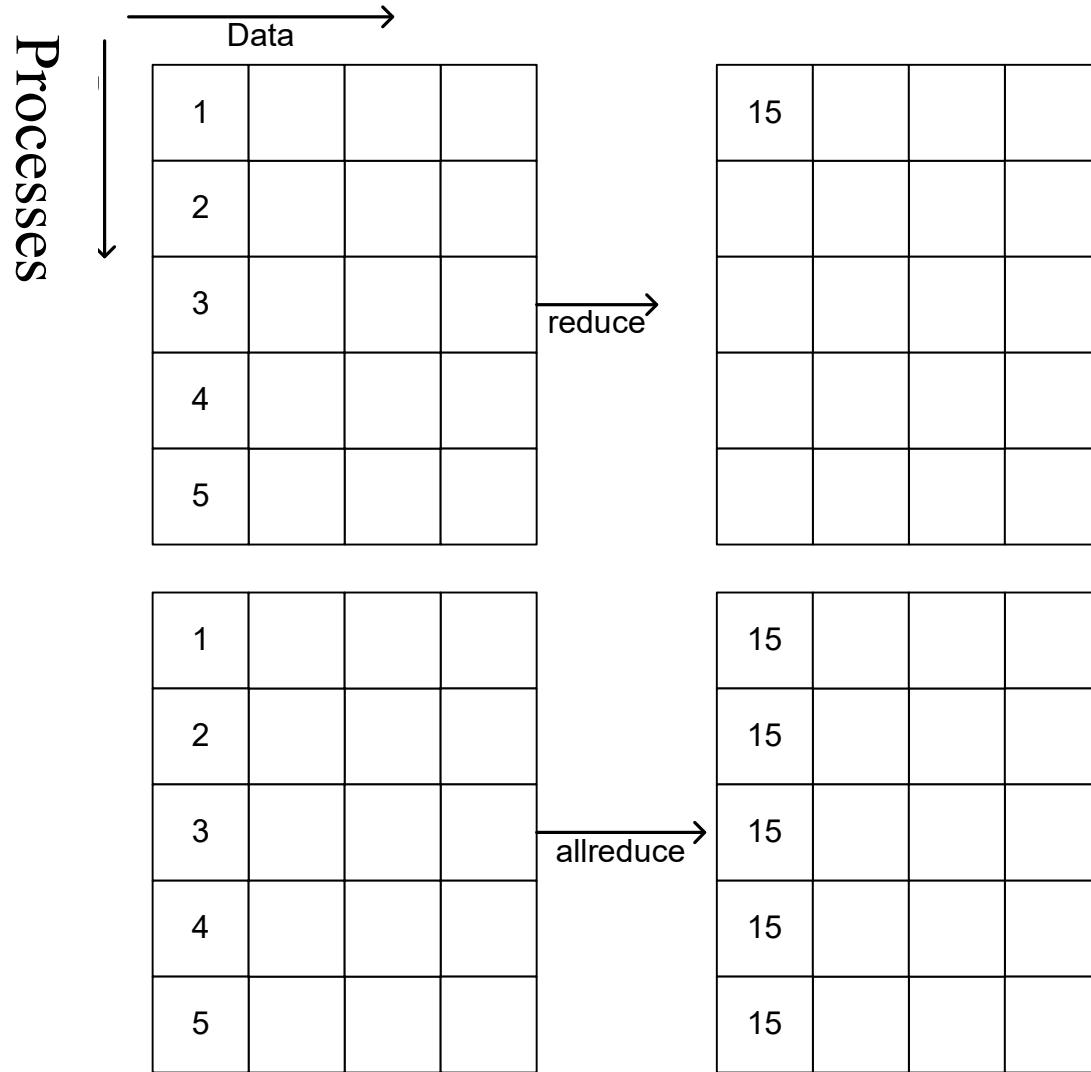


Collective communications

- Provided as a convenience for application developers:
 - Save significant development time
 - Efficient algorithms may be used
 - Stable (tested)
- Built on top of point-to-point communications



Reduce collective operations



MPI.PROD
MPI.SUM
MPI.MIN
MPI.MAX
MPI.LAND
MPI.BAND
MPI.LOR
MPI.BOR
MPI.LXOR
MPI.BXOR
MPI.MINLOC
MPI.MAXLOC

Documentation: Send() method

Scatter

```
public void Scatter(java.lang.Object sendbuf,  
                   int sendoffset,  
                   int sendcount,  
                   Datatype sendtype,  
                   java.lang.Object recvbuf,  
                   int recvoffset,  
                   int recvcount,  
                   Datatype recvtype,  
                   int root)  
    throws MPIException
```

Inverse of the operation Gather.

sendbuf	send buffer array
sendoffset	initial offset in send buffer
sendcount	number of items to send
sendtype	datatype of each item in send buffer
recvbuf	receive buffer array
recvoffset	initial offset in receive buffer
recvcount	number of items to receive
recvtype	datatype of each item in receive buffer
root	rank of sending process

Java binding of the MPI operation MPI_SCATTER.

Throws:

MPIException

Gather

```
public void Gather(java.lang.Object sendbuf,  
                   int sendoffset,  
                   int sendcount,  
                   Datatype sendtype,  
                   java.lang.Object recvbuf,  
                   int recvoffset,  
                   int recvcount,  
                   Datatype recvtype,  
                   int root)  
    throws MPIException
```

Each process sends the contents of its send buffer to the root process.

sendbuf	send buffer array
sendoffset	initial offset in send buffer
sendcount	number of items to send
sendtype	datatype of each item in send buffer
recvbuf	receive buffer array
recvoffset	initial offset in receive buffer
recvcount	number of items to receive
recvtype	datatype of each item in receive buffer
root	rank of receiving process

Java binding of the MPI operation MPI_GATHER.

Throws:

MPIException

Toy Example with Collectives

```
import mpi.*;

public class ScatterToyExample {

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

        MPI.Init(args);
        int rank = MPI.COMM_WORLD.Rank(); int size = MPI.COMM_WORLD.Size() ;
        int unitSize=4, root=0;
        int sendbuf[] = null;

        if(rank == root)
            sendbuf = new int[unitSize*size];

        int recvbuf[] = new int[unitSize];

        MPI.COMM_WORLD.Scatter(sendbuf, 0, unitSize, MPI.INT, recvbuf, 0, unitSize, MPI.INT, root);

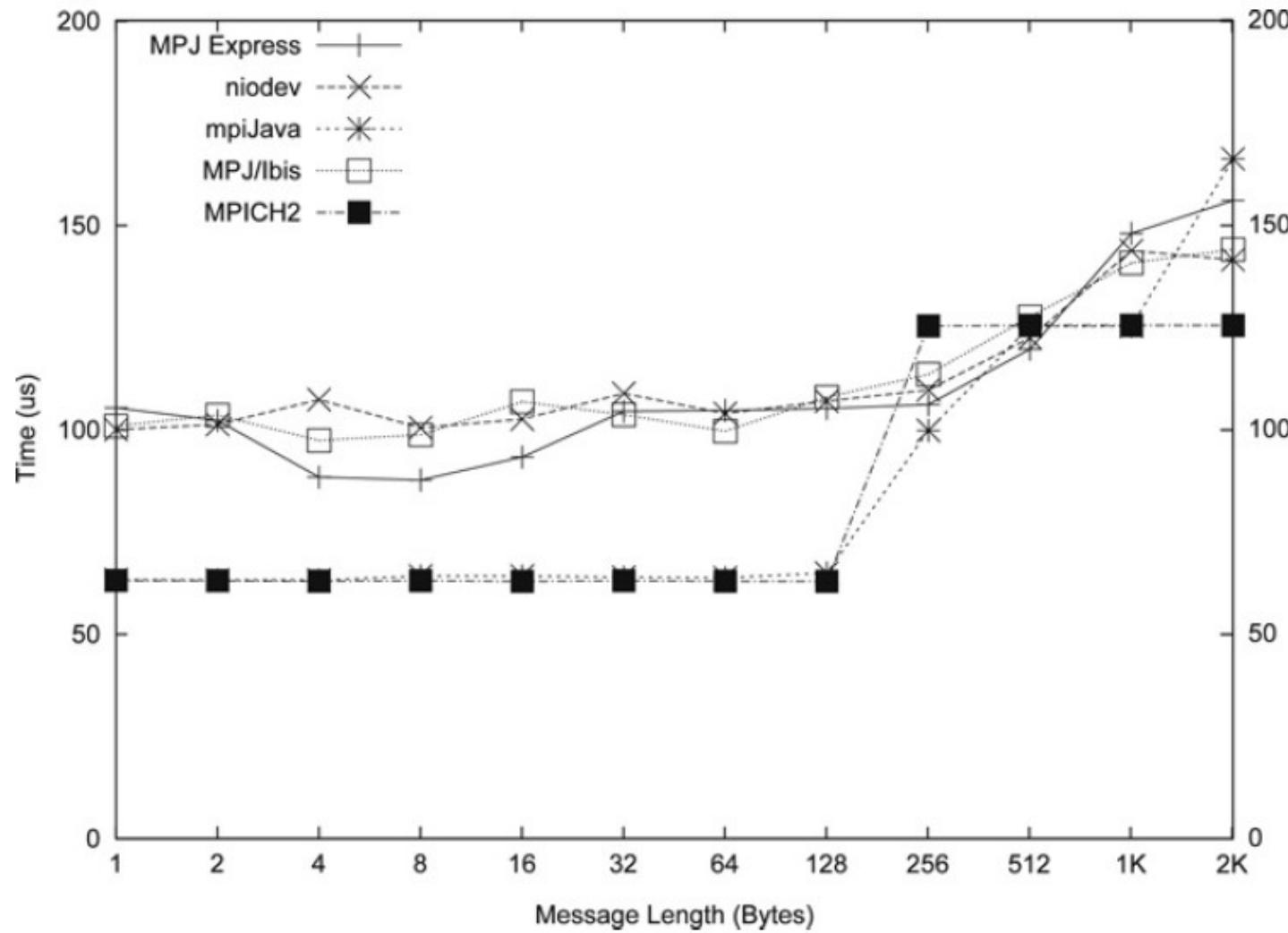
        if(rank != root)
            for(int i=0 ; i<unitSize; i++) recvbuf[i] = rank; /* computation loop */

        MPI.COMM_WORLD.Gather(recvbuf, 0, unitSize, MPI.INT, sendbuf, 0, unitSize, MPI.INT, root);

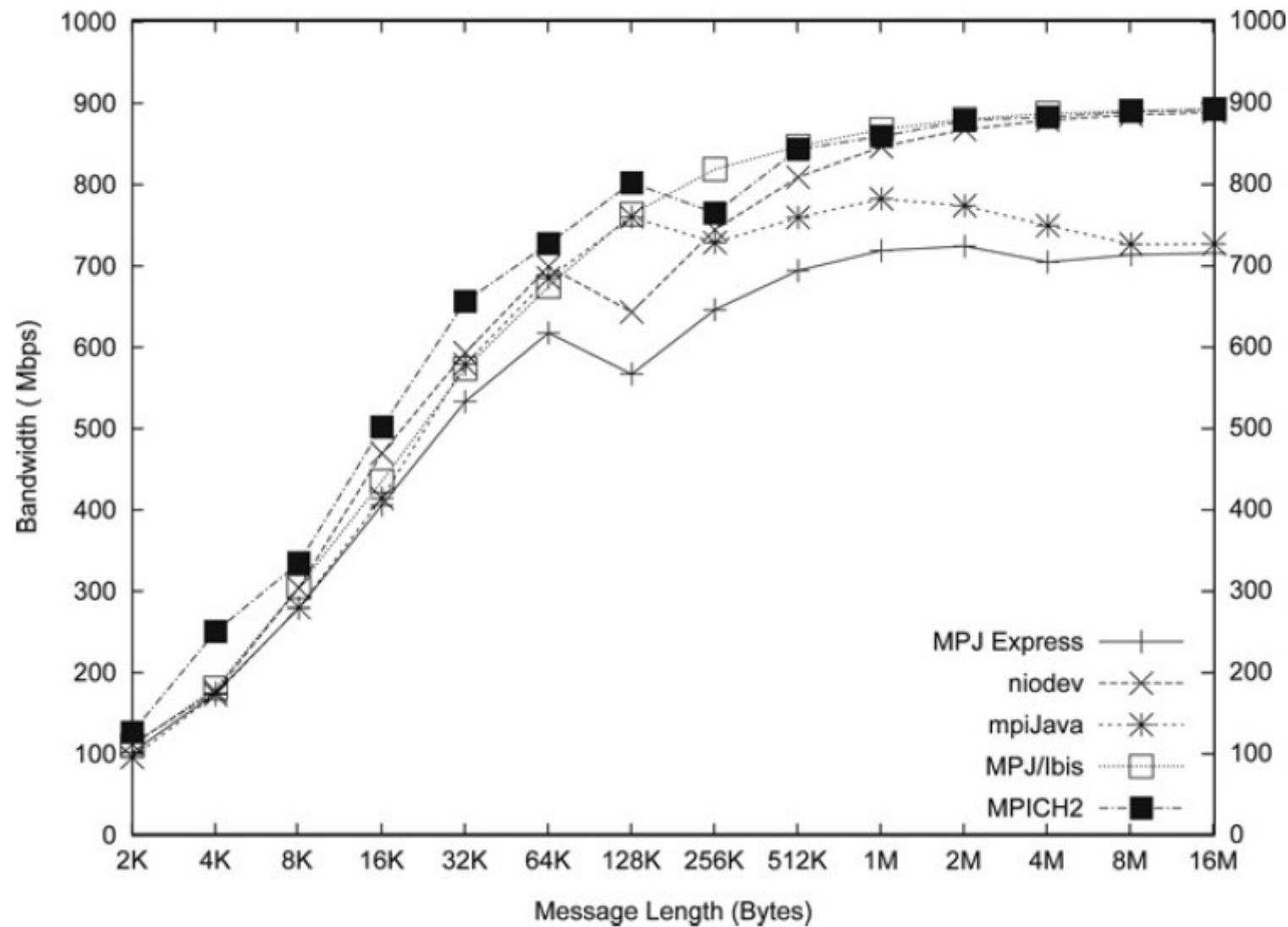
        if(rank == root)
            for(int i=0 ; i<unitSize*size ; i++)
                System.out.print(sendbuf[i]+" ");

        MPI.Finalize();
    }
}
```

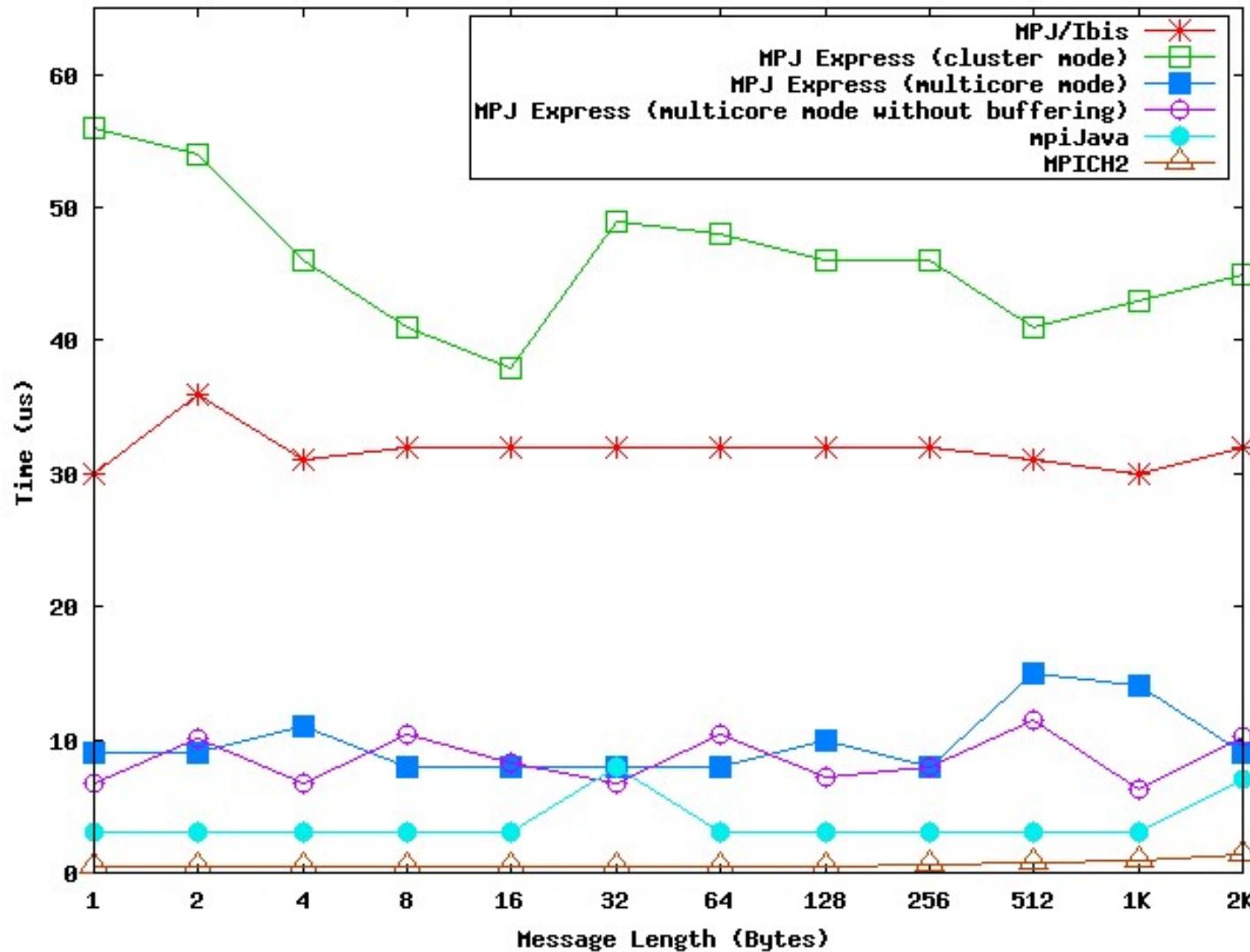
Latency Comparison



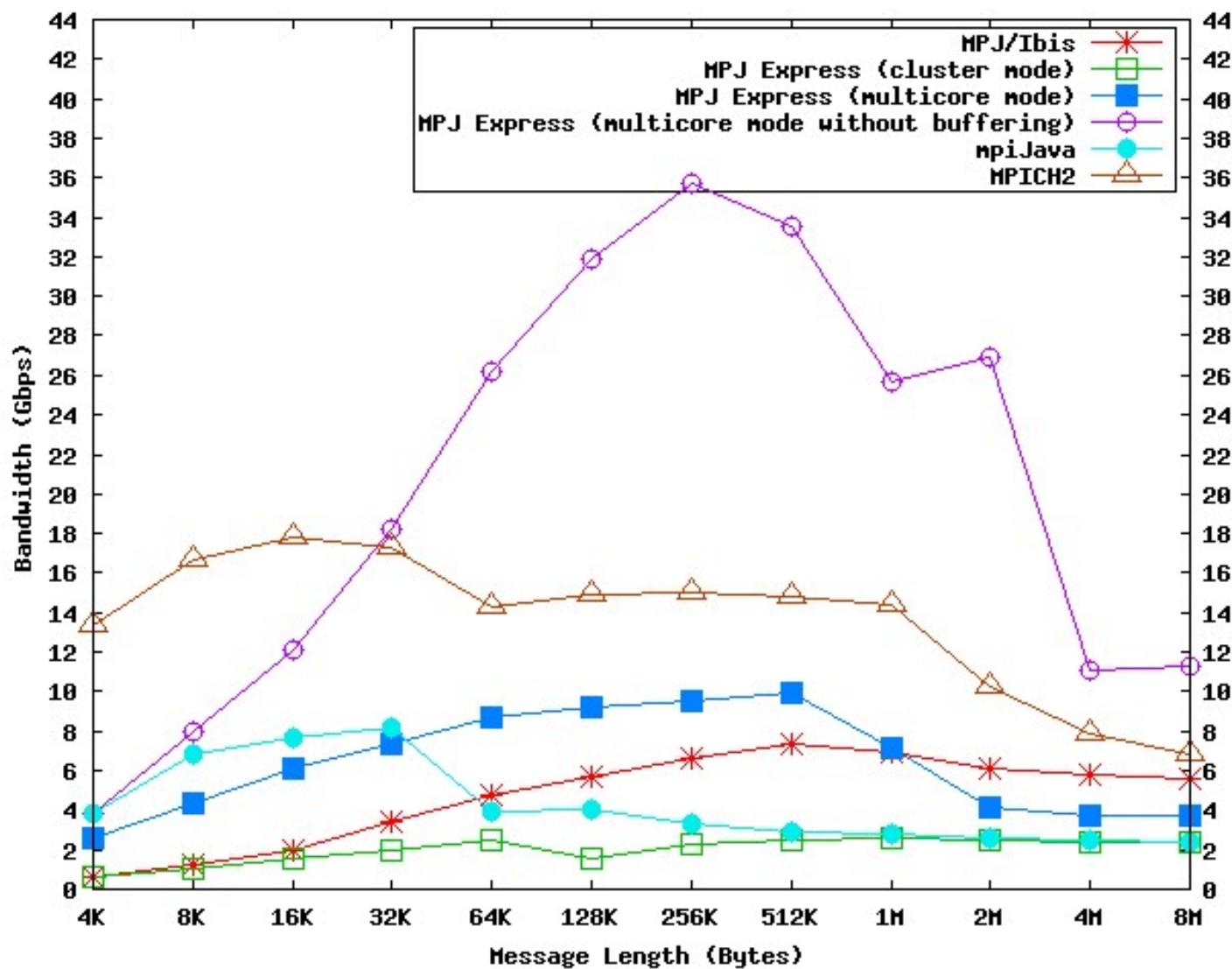
Throughput Comparison



Latency Comparison on a multicore machine



Throughput Comparison on a multicore machine



Лекція 6 (2 семестр)

Грід-системи

Грід-система та технологія Grid

Грід-системою є вільне об'єднання обчислювальних та, можливо, інформаційних ресурсів користувачів.

Grid є технологією забезпечення гнучкого, безпечного і скоординованого загального доступу до ресурсів

Технологія Grid не є технологією паралельних обчислень, в її завдання входить лише координація використання ресурсів

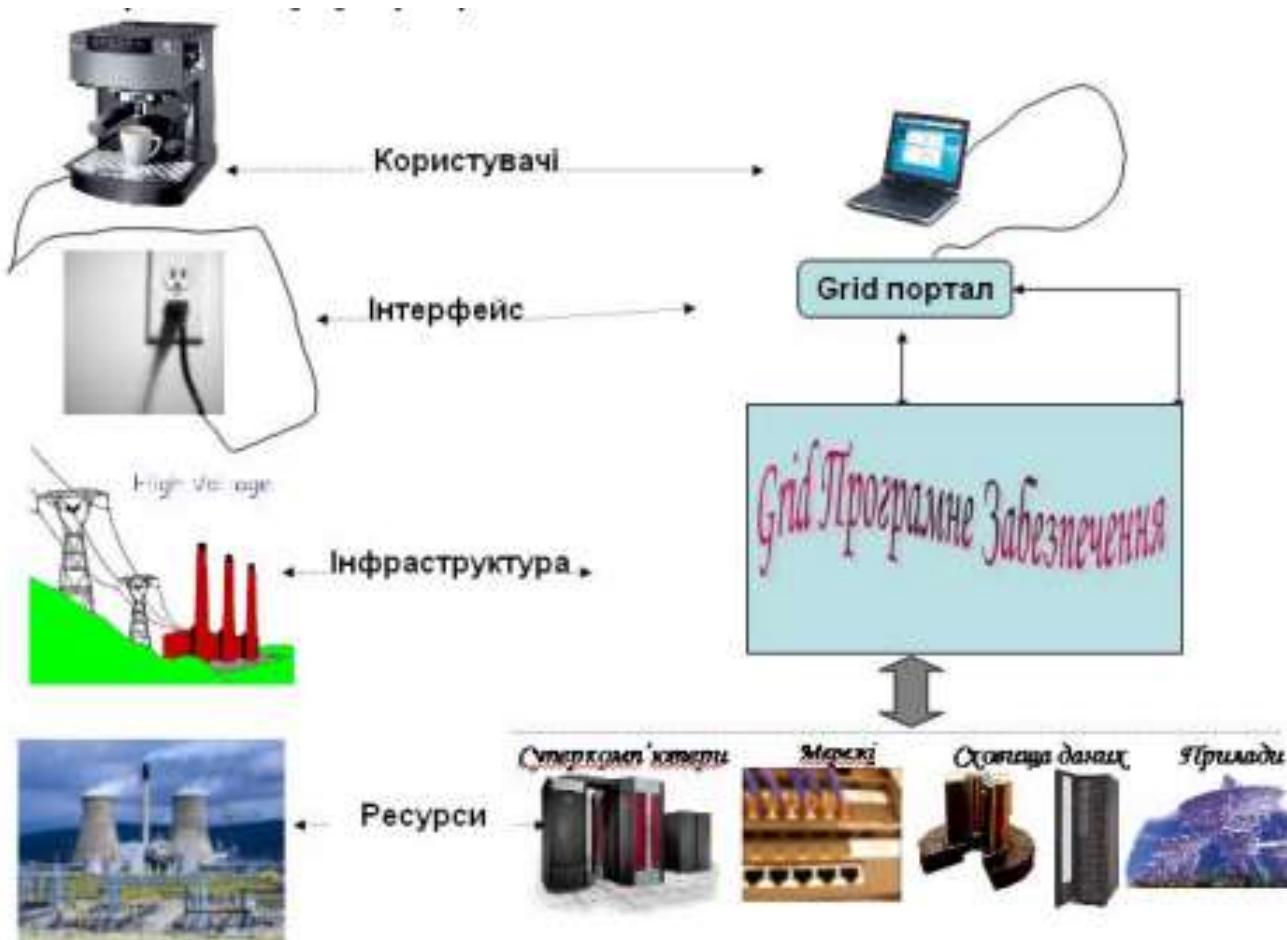
Типи грід-систем

Розрізняють:

- обчислюальні грід-системи (Computing Grid),
- інформаційні грід-системи (Data Grid)
- мішані грід-системи

Аналогія з електромережею

(за публікацію Петренко А.І. «Національна Grid - інфраструктура для забезпечення наукових досліджень і освіти»)



Програмне забезпечення для грід-систем

- Globus Toolkit
- Sun Grid Engine
 - Sun Grid Engine, Enterprise Edition (комерційне)

Програмне забезпечення грід-системи

Платформа – взаємоузгоджений набір засобів, здатних дати комплексне рішення задачі обслуговування Grid - інфраструктури виробничого призначення

- ARC(NorduGrid),
- gLite (www.glite.org),
- gUSE (grid User Support Environment), gUSE / WSPGRADE
- Alien,
- LCG,
- DataGrid (www.datagrid.org),
- Unicore (www.unicore.org),

В основі всіх - Globus Toolkit (www.globus.org)

Сервіси платформи грід-системи

- Сервіс аутентифікації і авторизації
- Сервіс брокера.
- Сервіс моніторингу і діагностики.
- Сервіс реплікації даних і програм.
- Сервіс пошуку програмного забезпечення.
- Сервіс каталогів даних і додатків
- Сервіс менеджменту ресурсів і завдань
- Сервіс зборки і вирішення задачі
- Сервіс обліку і оплати отриманих послуг

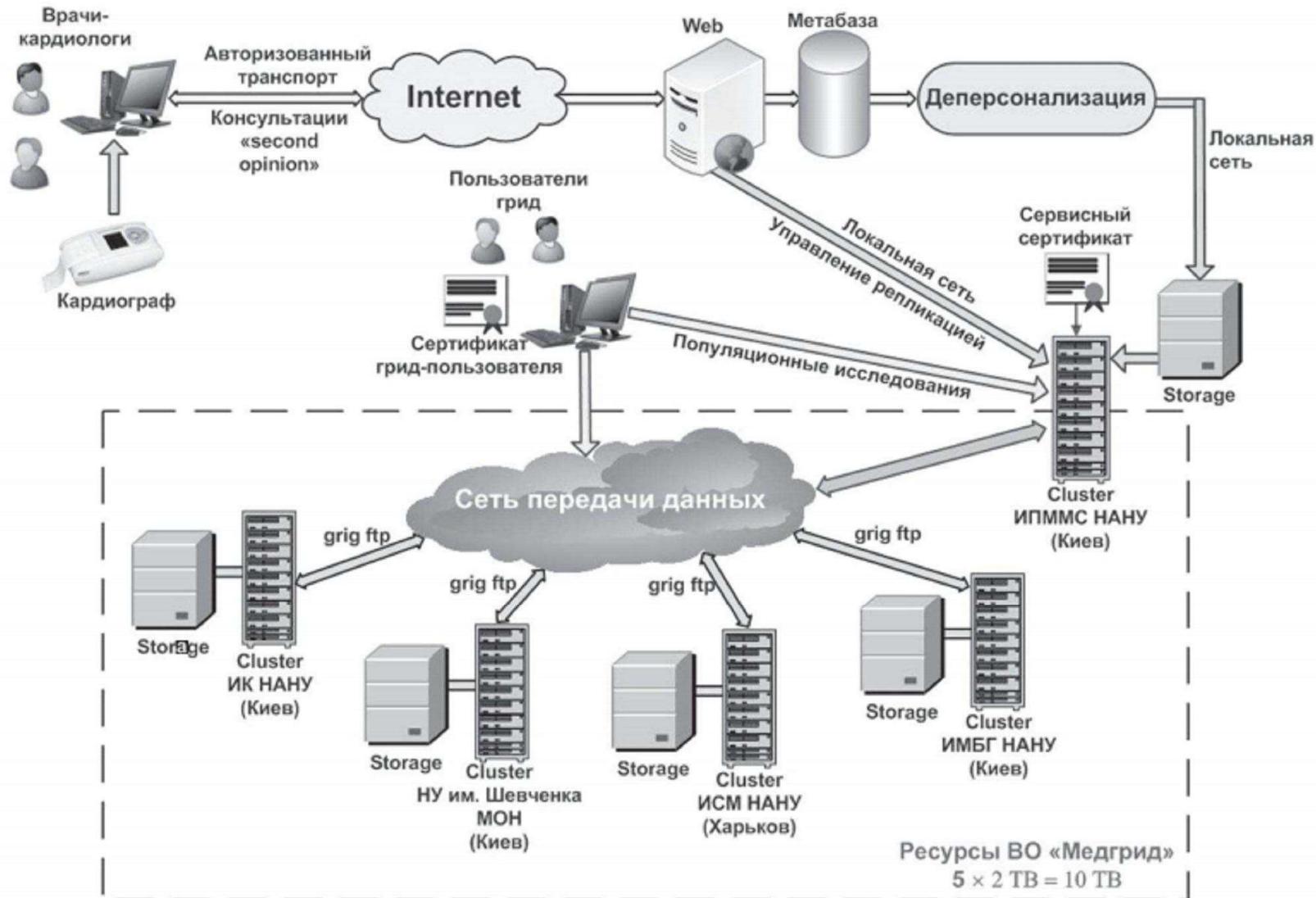
Структура сервісів ПЗ грід-системи



Найбільші грід-системи

- Європейський грід EGEE(Enabling Grids for E-science) функціонують 240 вузлів в 45 країнах (див. рис. 9.1), в яких задіяні 41,000 процесорів і 5 Пб (Petabytes) пам'яті. Мережа обслуговує більше 10,000 споживачів і 150 віртуальних організацій з продуктивністю більше 100,000 обчислювальних завдань за день

«Медгрид»



Системи моделювання грід-систем

- MicroGrid,
- OptorSim,
- GridSim,
- SimGrid

Архітектура грід-системи

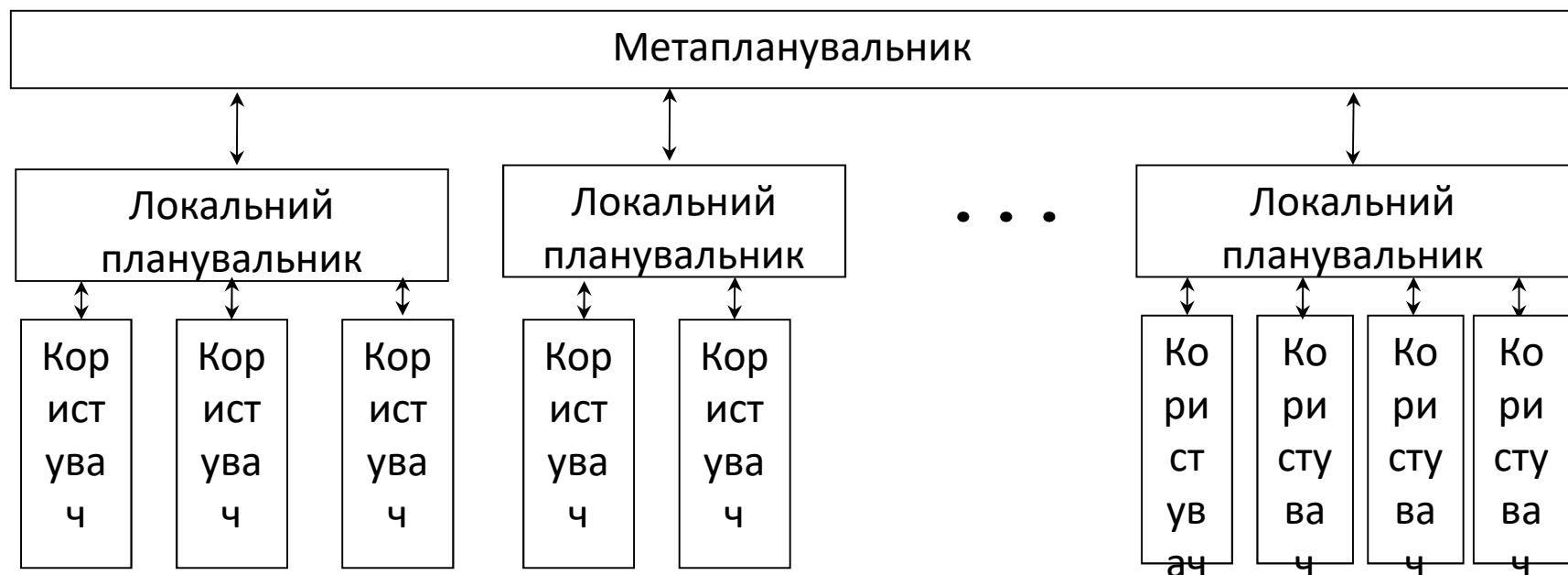
Складові архітектури:

- ❑ Метапланувальник
- ❑ розподілені обчислювальні ресурси (спільний віртуальний ресурс користувача)

Види архітектур:

- ❑ Однорівнева: планувальник – комп’ютери користувачів
- ❑ Дворівнева: метапланувальник – локальні планувальники - комп’ютери користувачів

Архітектура дворівневої грід-системи



Метапланувальник

- Метапланувальник здійснює розподіл завдань між обчислювальними кластерами
- У розподілених обчислювальних вузлах та сховищах даних встановлені грід-сервіси програмної інфраструктури, що надають інформацію про поточний стан ресурсу планувальнику виконання завдань
- Метапланувальник взаємодіє не з апаратними ресурсами, а з грід-сервісами, що представляють ці ресурси
- Локальні планувальники забезпечують управління виконанням завдань локальним обчислювальним ресурсом і взаємодіють з метапланувальником

Управління грід-ресурсами

Задача управління грід-ресурсами полягає в тому, щоб на основі інформації про поточний стан розподілених обчислювальних ресурсів та сховищ даних надавати дозвіл на використання віртуального обчислювального ресурсу користувачам грід-системи в залежності від їх потреби та у відповідності до їх вкладу у віртуальний обчислювальний ресурс (або пріоритету використання ресурсу), а також надавати несуперечливий доступ до інформаційного ресурсу.

Політики розподілу ресурсів

Sun Grid Engine, Enterprise Edition (SGEEE) надає можливість вибору однієї з чотирьох політик розподілу ресурсів між незалежно працюючими користувачами:

- 1) політика розділених ресурсів,
- 2) функціональна політика,
- 3) політика „роботи до строку”,
- 4) політика явного виділення ресурсів

В усіх запропонованих політиках користувач отримує дозвіл на використання ресурсу в залежності від наданої йому за домовленістю квоти на використання ресурсу, що визначається часткою віртуального ресурсу, на яку претендує користувач, або пріоритетом користувача на використання ресурсу

Розподіл ресурсів планувальником SGEEE

Розподіл ресурсів здійснюється планувальником на визначений інтервал часу.

Планувальник розподіляє ресурси, виходячи з співвідношення білетів, „закуплених” на даний інтервал часу.

Користувачі „купують квитки”, тобто посилають планувальніку замовлення ресурсу, для того, щоб отримати дозвіл на використання ресурсу у наступний інтервал часу. Планувальник, виходячи з одержаних замовлень на використання ресурсів, здійснює розподіл ресурсів.

Для кожного користувача підраховується усереднене фактичне використання обчислювального ресурсу і якщо воно перебільшує (або зменшує) надану квоту, то спрацьовує механізм компенсації (перене-(-недо-) споживання обчислювального ресурсу. Тобто, якщо користувач певний час не використовував ресурс, то його компенсаційний коефіцієнт збільшився і він має право отримати більшу частину ресурсу.

Якщо завдання, які захопили ресурс (раніше інших завдань), використовують ресурс більше наданої квоти (більше, ніж надає квиток), то користувач лишається права отримати певну кількість квитків.

Частка доступного віртуального ОР користувача у випадку однорівневої грід-системи

$$\forall i \quad x_i = \begin{cases} \frac{p_i}{\sum_{i \in A} p_i}, & i \in A, \\ 0, & i \notin A. \end{cases}$$

де A - підмножина активних користувачів грід-системи, p - частка доступного обчислювального ресурсу користувача.

Частка доступного віртуального ОР користувача у випадку дворівневої грід-системи

$$\forall i \in U, j \in K \quad x_i = \begin{cases} \frac{q_i}{\sum_{i \in A \cap U_j} q_i} \cdot \frac{r_j}{\sum_{j \in K_A} r_j}, & (i \in A) \wedge (i \in U_j), \\ 0, & (i \notin A) \vee (i \notin U_j) \end{cases}$$

де K – підмножина активних вузлів(кластерів),
 A - підмножина активних користувачів вузла грід-системи,
 q – частка доступного віртуального обчислювального ресурсу користувача вузла,
 r – частка доступного віртуального обчислювального ресурсу вузла .

На початку кожного такту планування завдання, обсяг вимоги ОР яких не перевищує обсяг доступного віртуального ресурсу користувача, отримують дозвіл на захоплення віртуального ОР і захоплюють ресурс на час, рівний тривалості виконання завдання.

Модель розподілу ресурсів

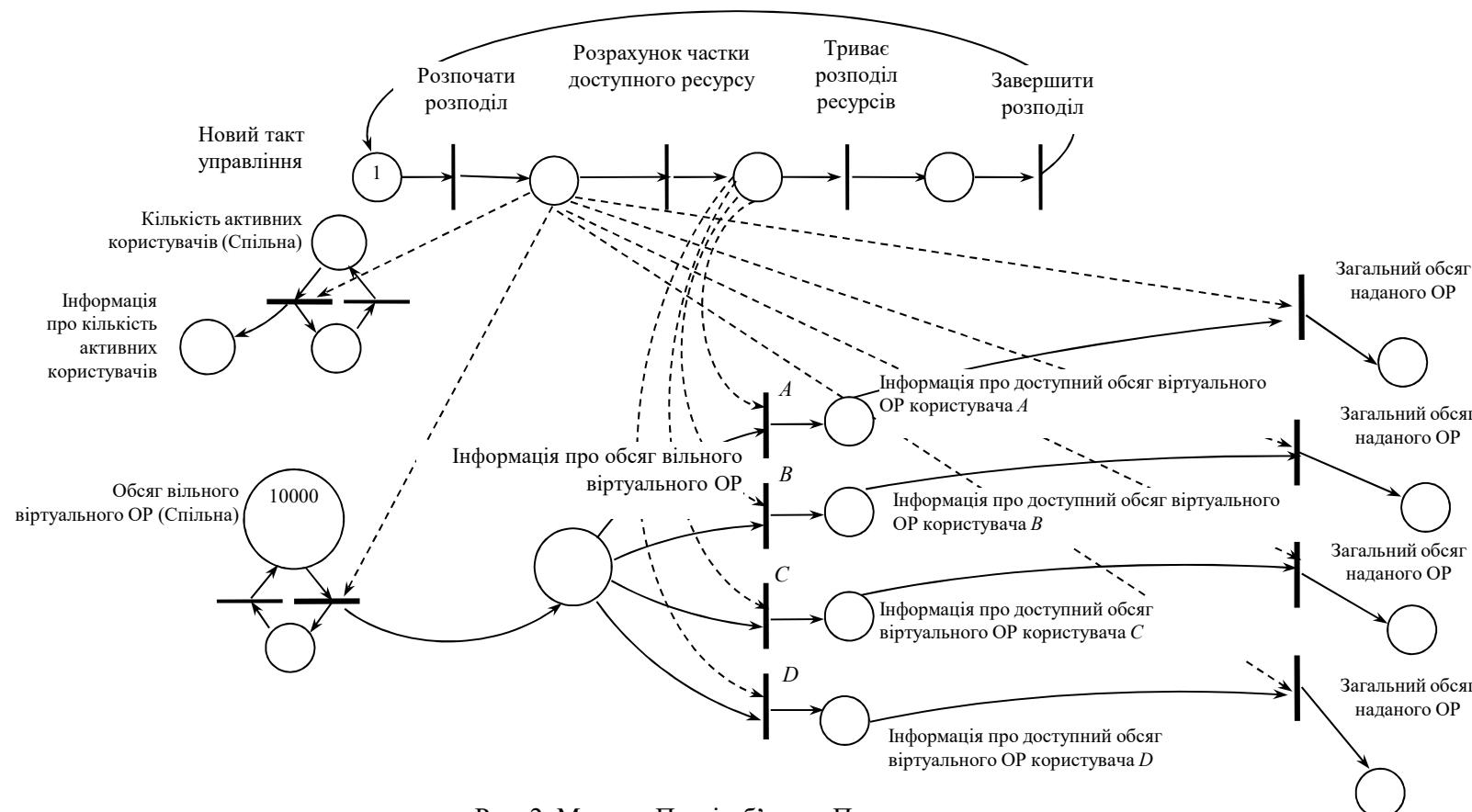
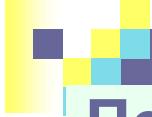
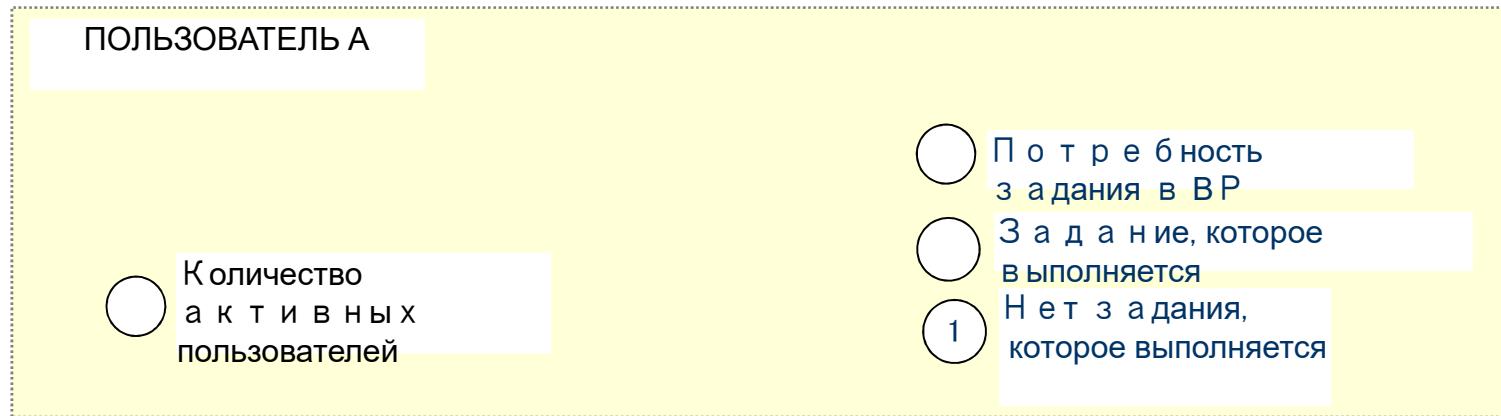
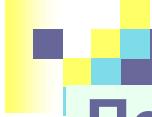


Рис. 2. Мережа Петрі-об'єкта «Планувальник»

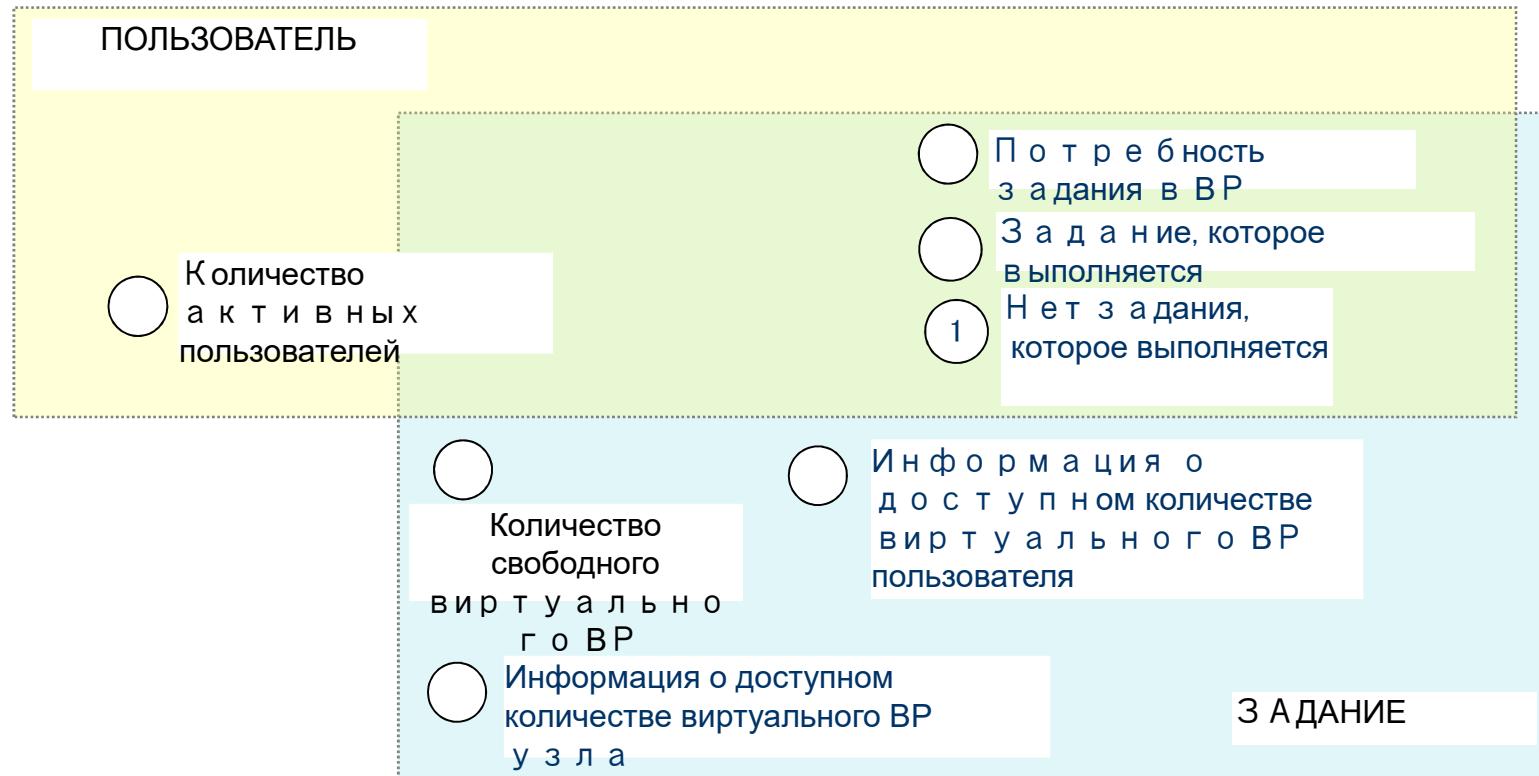


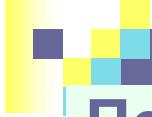
Петри-объектная модель системы управления распределенными вычислительными ресурсами



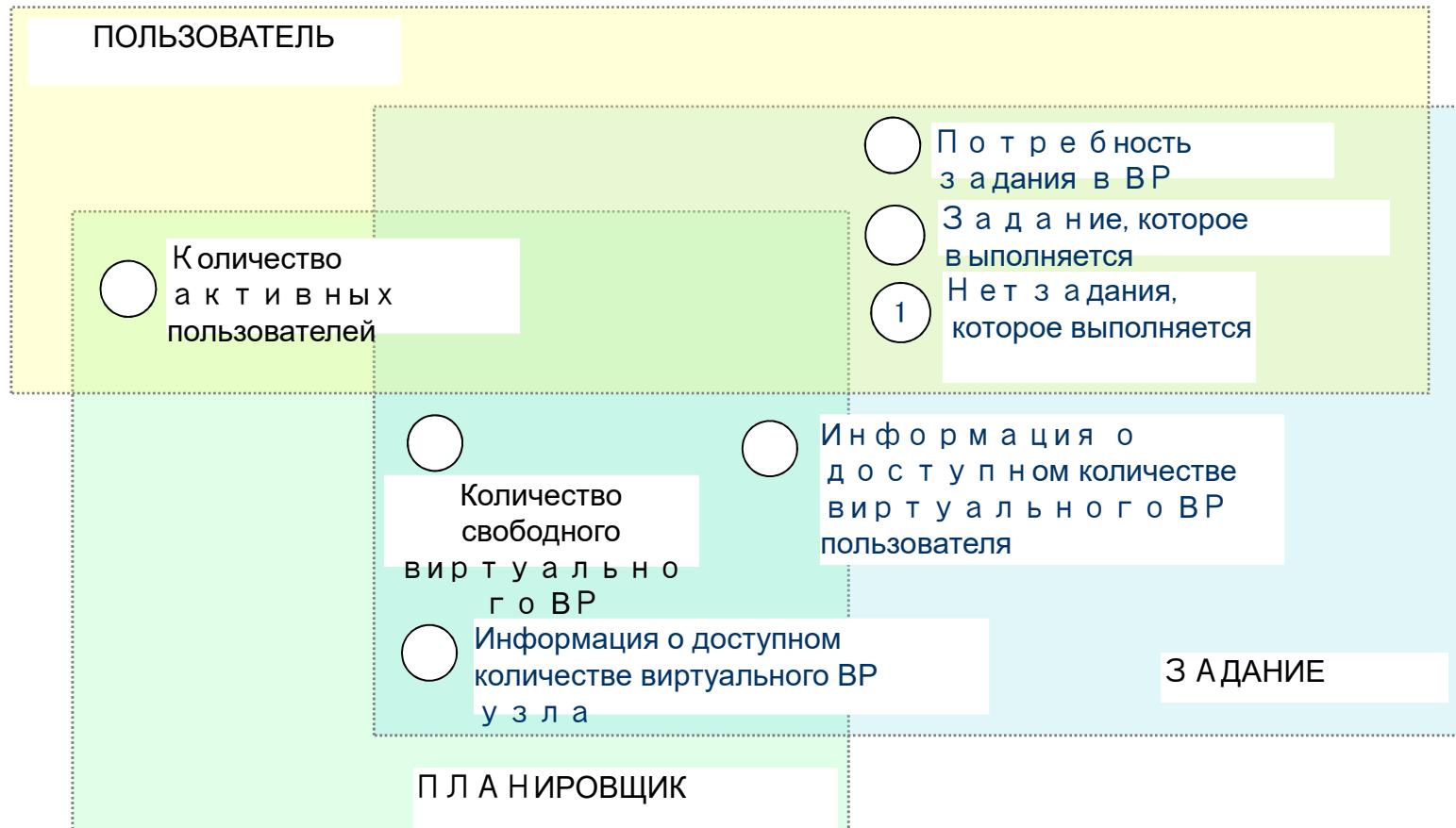


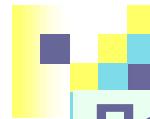
Петри-объектная модель системы управления распределенными вычислительными ресурсами



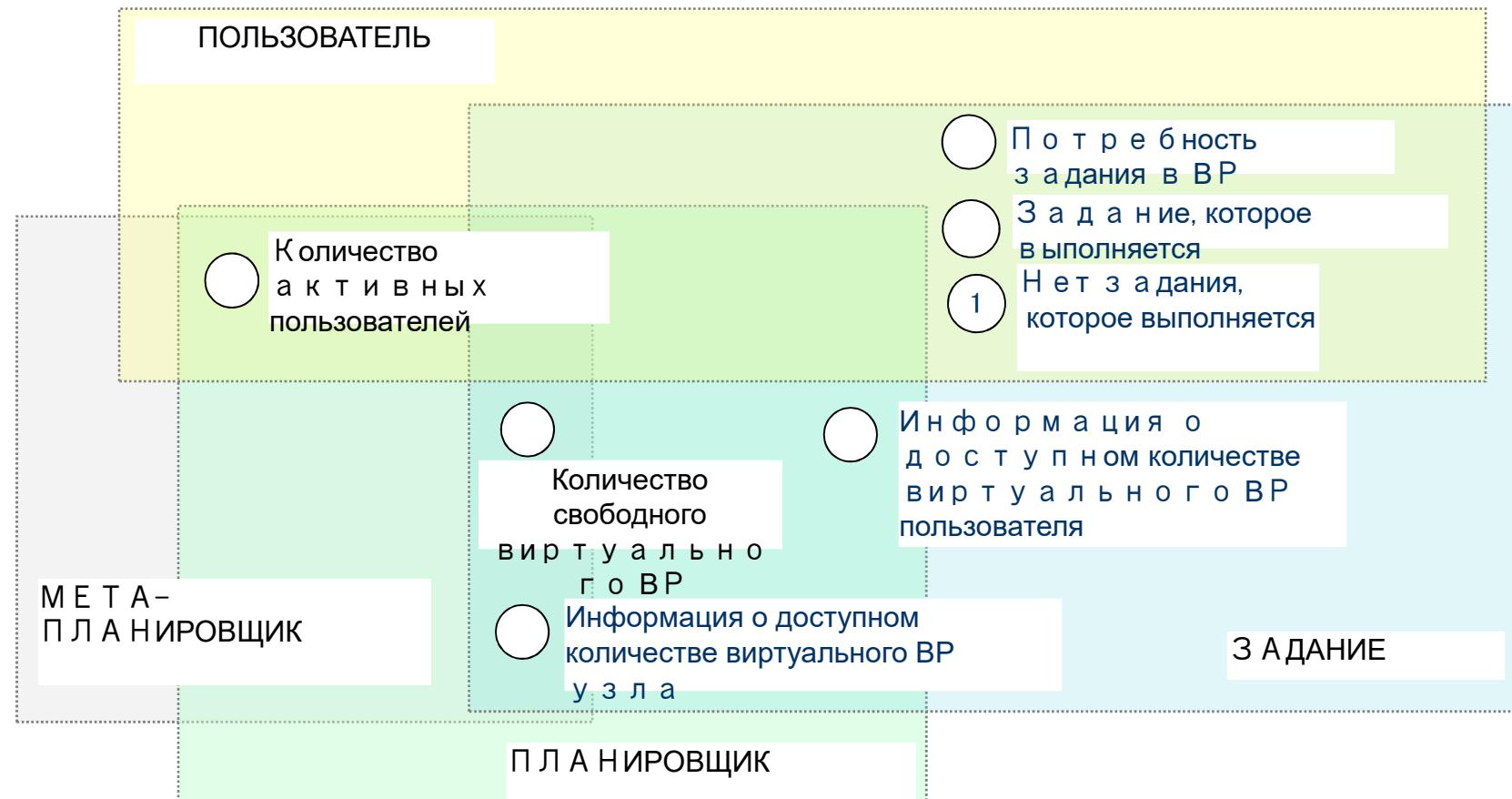


Петри-объектная модель системы управления распределенными вычислительными ресурсами





Петри-объектная модель системы управления распределенными вычислительными ресурсами



Java-реализация модели системы управления распределенными вычислительными ресурсами

The screenshot shows the NetBeans IDE 7.0.1 interface with the following components:

- Title Bar:** GridResources - NetBeans IDE 7.0.1
- Menu Bar:** Файл Правка Вид Переход Исходный файл Реорганизация код Выполнить Отладка Профилировать Группа Сервис Окно Справка Поиск (Ctrl+I)
- Toolbar:** Includes icons for file operations like New, Open, Save, and Run.
- Project Explorer:** Shows the project structure under GridResources, including packages PetriObj and gridresources, and files like FunRand.java, NetLibrary.java, PetriNet.java, PetriObjModel.java, PetriP.java, PetriSim.java, PetriT.java, TieIn.java, TieOut.java, Main.java, PlanSim.java, TaskSim.java, and UserSim.java.
- Code Editor:** Displays Java code for PetriObjModel.java. The code handles task prioritization, printing simulation marks, and calculating summa based on user and task lists.
- Output Window:** Shows the command-line output for the run configuration "GridResources (run)". The output lists results for various indices i, each consisting of two values separated by a space.
- Bottom Status Bar:** Includes icons for file operations and status information like 148 | 15 | INS.

```
        ListObj.add(T);
        for(TaskSim T:ListTaskC)
        { T.setPriority(2);
            ListObj.add(T);
        }
        for(TaskSim T:ListTaskD)
        { T.setPriority(2);
            ListObj.add(T);
        }

        ListObj.add(Plan);
        Plan.setPriority(1);

System.out.println("Початок моделювання: ");
for(PetriSim Sim>ListObj)
    Sim.printMark();
PetriObjModel GridModel = new PetriObjModel(ListObj);
GridModel.setIsProtokol(false); //НЕ друкувати протокол подій

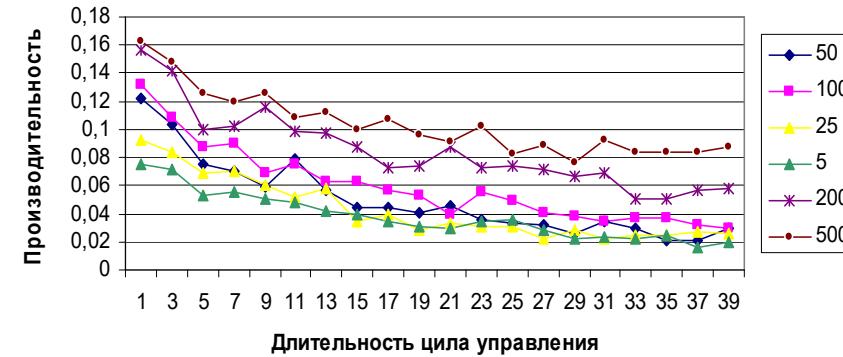
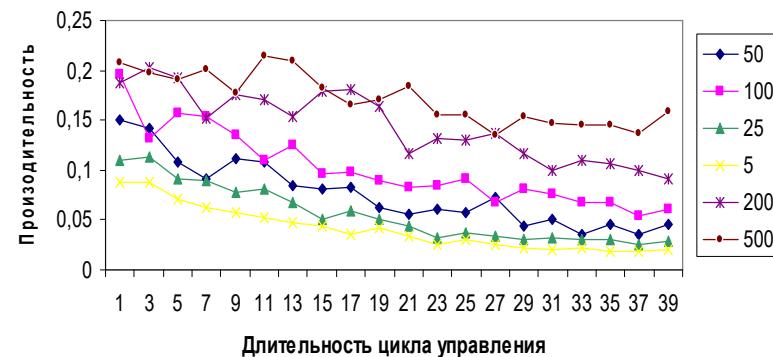
GridModel.Go(timeModeling);
for(UserSim U:ListUser)
    for(TaskSim T:U.getTaskList())
        summa =summa+ T.getNumRefusals();
result[i] = result[i]+summa;
```

i	result[i]	value
8	result[8]	=160.25 185
9	result[9]	=134.25 230
10	result[10]	=101.75 280
11	result[11]	=132.0 335
12	result[12]	=105.5 395
13	result[13]	=147.75 460
14	result[14]	=131.5 530
15	result[15]	=146.5 605
16	result[16]	=97.0 685
17	result[17]	=139.0 770

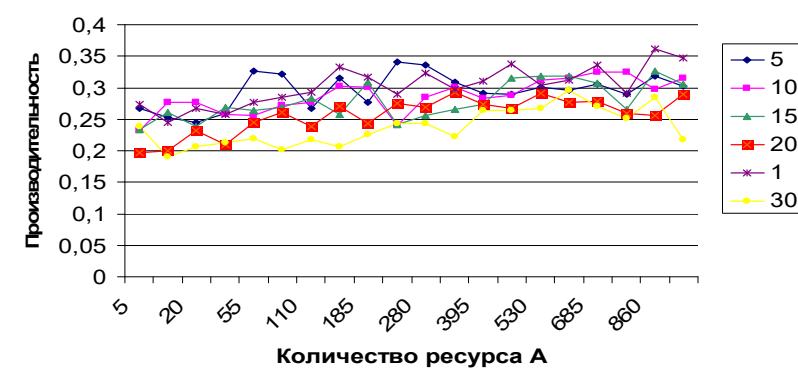
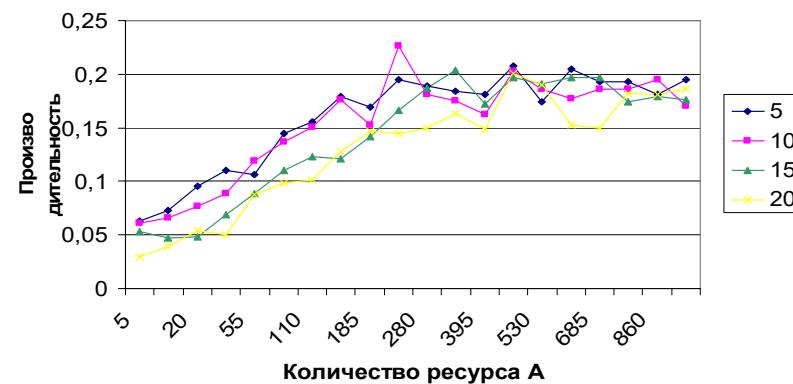


Результати моделювання

Производительность при изменении длительности цикла управления для различных значений количества ресурса пользователя A: а) статическое управление, б) динамическое управление

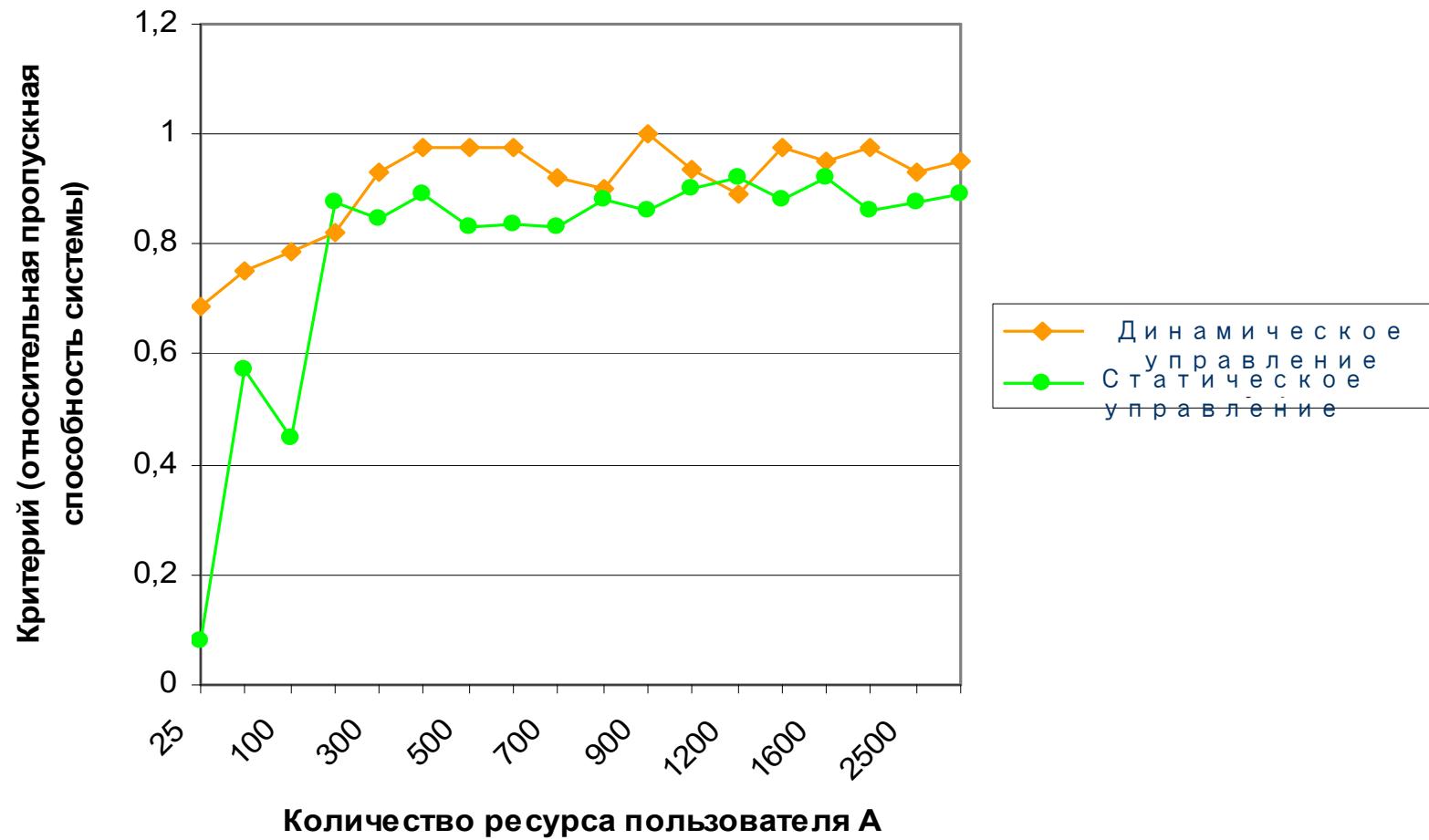


Производительность при изменении количества ресурса пользователя A для различных значений цикла управления : а) количество ресурса других пользователей 5MIPS, б) количество ресурса других пользователей 100 MIPS





Результаты исследования влияния типа управления на эффективность функционирования системы



Лекція

Технології розподіленого
програмування java

- Java Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

Клієнт-серверна модель

Модель клієнт / сервер є однією з форм розподілених обчислень, в якій одна програма (клієнт) обмінюється даними з іншою програмою (сервером) з метою обміну інформацією. У цій моделі, як клієнт, так і сервер зазвичай «говорять» на одній мові - протоколі, що є зрозумілим і клієнту, і серверу - тому вони мають можливість спілкуватися.

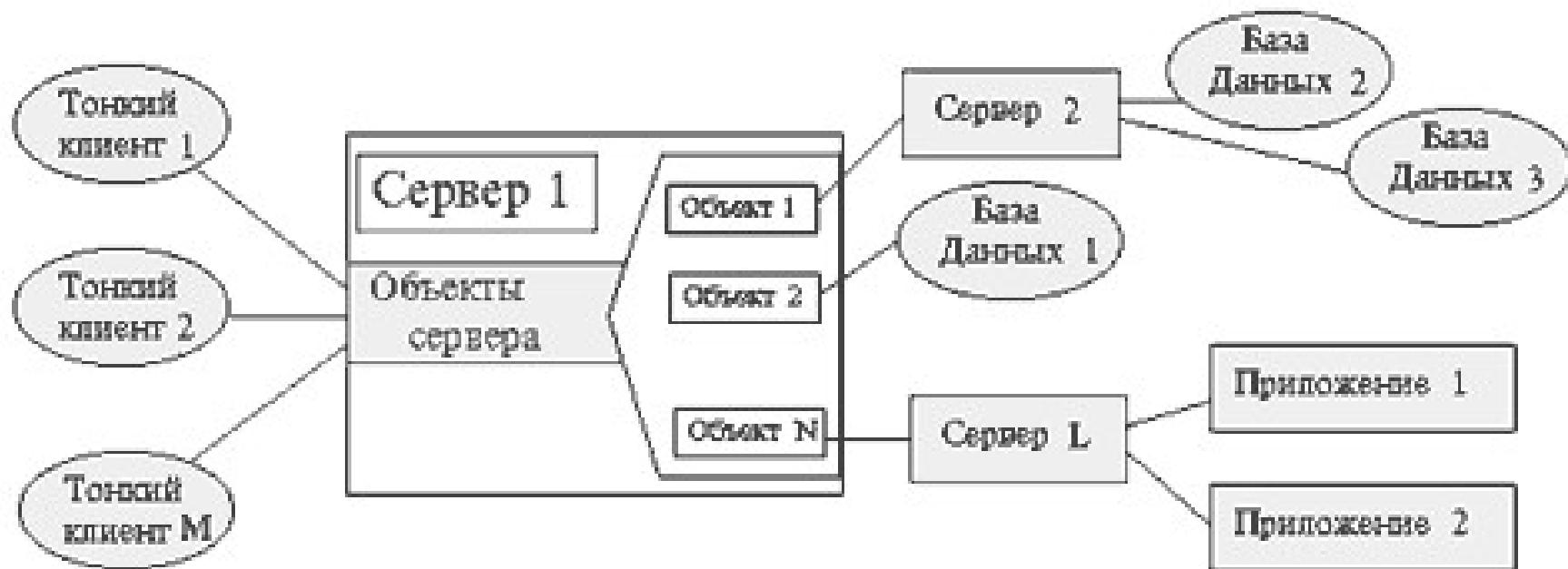
Модель клієнт / сервер може бути реалізована в різний спосіб, але зазвичай вона реалізується за допомогою сокетів низького рівня. Використання сокетів для розробки систем клієнт / сервер означає, що ми повинні розробити протокол, який представляє собою набір команд, узгоджених між клієнтом і сервером, за допомогою якого вони зможуть спілкуватися. Наприклад, протокол HTTP, який забезпечує метод GET, що має бути реалізований всіма веб-серверами і використовуватись веб-клієнтами (браузери) для відновлення документів.

Модель розподілених об'єктів

Розподілений об'єкт-система являє собою колекцію об'єктів, які ізолюють запитувачів сервісів (клієнтів) від постачальників сервісів (серверів) за допомогою добре визначеного зовнішнього інтерфейсу. Іншими словами, клієнти ізольовані від реалізації сервісів через представлення даних і виконуваний код. Це одна з основних відмінностей, що відрізняють розподілену об'єктно-орієнтовану модель від чистої моделі клієнт / сервер.

У розподіленій об'єктно-орієнтованій моделі, клієнт посилає повідомлення об'єкту, який в свою чергу інтерпретує повідомлення, щоб вирішити, який сервіс виконувати. Вибір сервісу, або методу, може бути виконаний або об'єктом, або брокером. Технології Java Remote Method Invocation (RMI) і Common Object Request Broker Architecture (CORBA) є прикладами реалізації цієї моделі.

Модель розподілених об'єктів



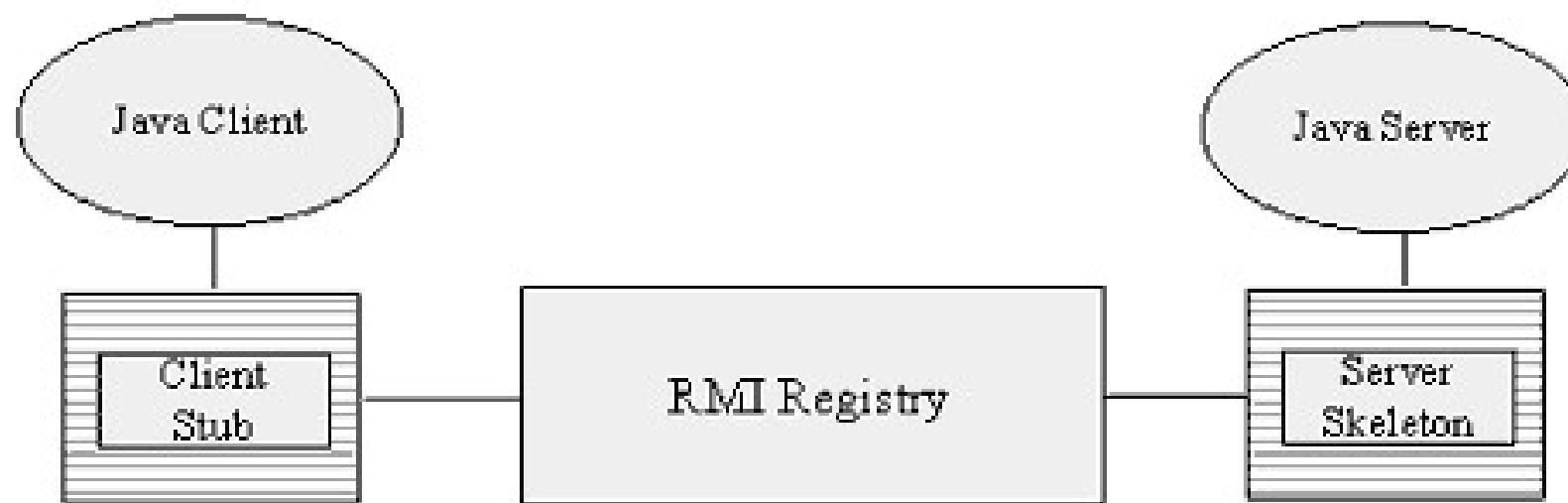
Remote Method Invocation (RMI)

RMI є розподілена об'єктна система, що надає можливість легко розробляти розподілені Java-додатки.

Розробка розподілених додатків в RMI простіша, ніж розробка з сокетами, оскільки немає необхідності в розробці протоколу.

В RMI, розробник має ілюзію виклику локального методу з локального файлу класу, в той момент як насправді аргументи відправляються до віддаленого ресурсу та інтерпретуються, а результати відправляються назад замовнику.

Remote Method Invocation (RMI)



Розробка розподілених додатків з використанням RMI

складається з таких кроків:

- визначити віддалений інтерфейс
- реалізувати віддалений інтерфейс
- розробити сервер
- розробити клієнт
- створити client stub і server skeleton,
запустити RMI реєстрацію серверу і клієнта

Приклад

Крок 1. Визначення remote інтерфейсу

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface FileInterface extends Remote {
    public byte[] downloadFile(String fileName)
        throws RemoteException;
}
```

Приклад

Крок 2. Реалізація remote інтерфейсу

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class FileImpl extends UnicastRemoteObject implements FileInterface {
    private String name;
    public FileImpl(String s) throws RemoteException{
        super();
        name = s;
    }
    public byte[] downloadFile(String fileName) {
        try {
            File file = new File(fileName);
            byte buffer[] = new byte[(int)file.length()];
            BufferedInputStream stream = new BufferedInputStream(
                new FileInputStream(fileName));
            stream.read(buffer,0,buffer.length());
            stream.close();
            return (buffer);
        } catch(Exception e) {
            System.out.println("FileImpl: "+e.getMessage());
            e.printStackTrace();
            return (null);
        }
    }
}
```

Приклад

Крок 3. Розробка серверу

```
import java.io.*;
import java.rmi.*;

public class FileServer {
    public static void main(String argv[]) {
        if(System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            FileInterface fi = new FileImpl("FileServer");
            Naming.rebind("//127.0.0.1/FileServer", fi); // порт за замовчуванням 1099
            // Naming.rebind("//127.0.0.1:4500/FileServer", fi) //порт 4500

        } catch(Exception e) {
            System.out.println("FileServer: "+e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Приклад

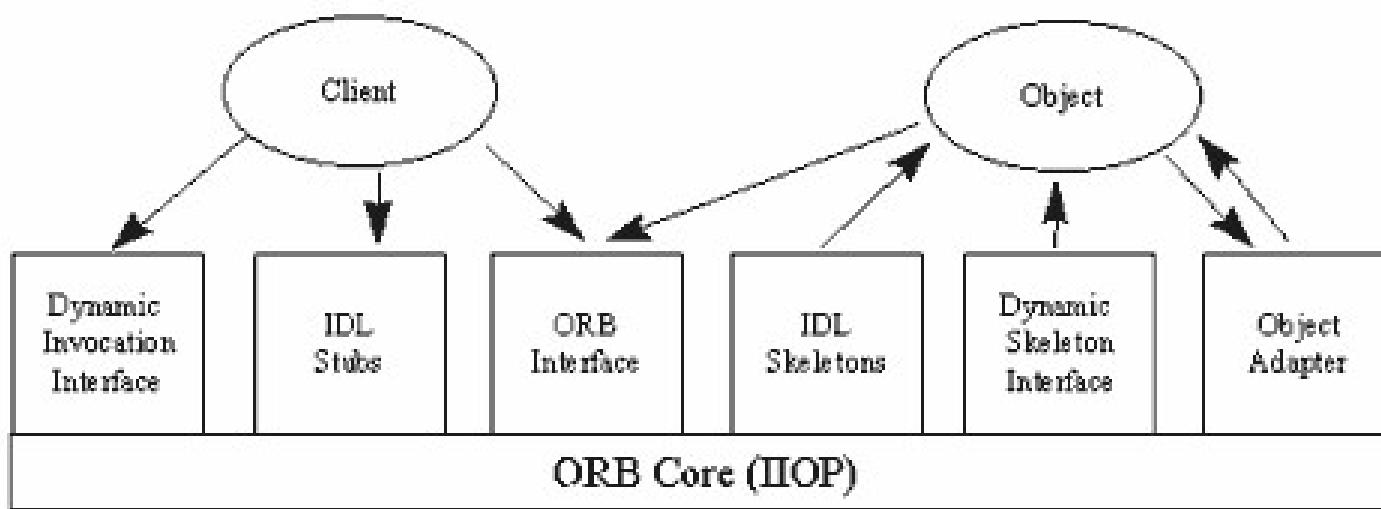
Крок 4. Розробка клієнту

```
import java.io.*;
import java.rmi.*;
public class FileClient{
    public static void main(String argv[]) {
/* the client accepts two arguments at the command line: the name of the file to be downloaded and the address of
the machine from which the file is to be downloaded, which is the machine that is running the file server */
        if(argv.length != 2) {
            System.out.println("Usage: java FileClient fileName machineName");
            System.exit(0);
        }
        try {
            String name = "//" + argv[1] + "/FileServer";
            FileInterface fi = (FileInterface) Naming.lookup(name);
            byte[] filedata = fi.downloadFile(argv[0]);
            File file = new File(argv[0]);
            BufferedOutputStream stream = new BufferedOutputStream(
                new FileOutputStream(file.getName()) );
            stream.write(filedata,0,filedata.length);
            stream.flush();
            stream.close();
        } catch(Exception e) {
            System.err.println("FileServer exception: "+ e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Common Object Request Broker Architecture (or CORBA)

- CORBA – стандарт архітектури, розроблений Object Management Group (OMG) для розвитку розподіленого об'єктного програмування
- ORB (or Object Request Broker), VisiBroker, ORBIX – реалізації CORBA
- JavaIDL – Java-реалізація CORBA

Common Object Request Broker Architecture (or CORBA)



Dynamic Invocation Interface (DII): надає можливість клієнту знаходити сервера та викликати їх методи під час роботи системи

IDL Stubs: визначає, яким чином клієнт виконує виклик сервера

ORB Interface: як для клиєнта, так і для сервера сервіси

IDL Skeleton: забезпечує статичні інтерфейси для об'єктів певного типу

Dynamic Skeleton Interface: спільні інтерфейси для об'єктів, незалежно від їх типу, які не були визначені в IDL Skeleton

Object Adapter: здійснює комунікаційну взаємодію між об'єктом та ORB

Розробка CORBA-застосунку

Складається з таких кроків:

- Визначити інтерфейс в IDL
- Скласти карту інтерфейсу IDL в Java
(виконується автоматично)
- Реалізувати інтерфейс
- Розробити сервер
- Розробити клієнт
- Запустити службу імен, сервер, і клієнт

Визначення інтерфейсу

```
interface FileInterface {  
    typedef sequence<octet> Data;  
    Data downloadFile(in string fileName);  
};
```

Реалізація інтерфейсу

```
import java.io.*;  
  
public class FileServant extends _FileInterfaceImplBase {  
    public byte[] downloadFile(String fileName) {  
        File file = new File(fileName);  
        byte buffer[] = new byte[(int)file.length()];  
        try {  
            BufferedInputStream stream = new BufferedInputStream(  
                new FileInputStream(fileName));  
            stream.read(buffer,0,buffer.length);  
            stream.close();  
  
        } catch(Exception e) {  
            System.out.println("FileServant Error: "+e.getMessage());  
            e.printStackTrace();  
        }  
        return(buffer);  
    }  
}
```

Розробка серверу

```
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class FileServer {
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null); // create and initialize the ORB
            FileServant fileRef = new FileServant(); //create the servant and register it with the ORB
            orb.connect(fileRef);
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService"); // get the root naming context

            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc =
                new NameComponent("FileTransfer", " ");
                // Bind the object reference in naming context
            NameComponent path[] = {nc};
            ncRef.rebind(path, fileRef);
            System.out.println("Server started....");

            java.lang.Object sync = new java.lang.Object(); // Wait for invocations from clients
            synchronized(sync) {
                sync.wait();
            }
        } catch(Exception e) {
            System.err.println("ERROR: " + e.getMessage());
            e.printStackTrace(System.out);
        }
    }
}
```

Розробка клієнта

```
import java.io.*;
import java.util.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
public class FileClient {
    public static void main(String argv[]) {
        try {
            ORB orb = ORB.init(argv, null); // create and initialize the ORB
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService"); // get the root naming context
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("FileTransfer", " ");
            NameComponent path[] = {nc}; // Resolve the object reference in naming
            FileInterfaceOperations fileRef =
                FileInterfaceHelper.narrow(ncRef.resolve(path));
            if(argv.length < 1) {
                System.out.println("Usage: java FileClient filename");
            }
            File file = new File(argv[0]); // save the file
            byte data[] = fileRef.downloadFile(argv[0]);
            BufferedOutputStream output = new
                BufferedOutputStream(new FileOutputStream(argv[0]))
            output.write(data, 0, data.length);
            output.flush();
            output.close();
        } catch(Exception e) {
            System.out.println("FileClient Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Порівняння CORBA з RMI

- CORBA інтерфейси визначені в IDL, а інтерфейси RMI визначені в Java. RMI-IIOP дозволяє записувати всі інтерфейси в Java (див. RMI-IIOP)
- CORBA підтримує `in` та `out` параметри, в той час як RMI не так як локальні об'єкти передаються копії і віддалені об'єкти передаються по посиланню
- CORBA задумувалась та розроблялась як мовно незалежна. Це означає, що деякі об'єкти можуть бути написані на Java, наприклад, а інші можуть бути написані на C++, та все ж всі вони можуть взаємодіяти. Тому CORBA є ідеальним механізмом для наведення мостів між різними мовами програмування. Водночас, RMI була розроблена для однієї мови, де всі об'єкти написані на Java. Однак слід зазначити, що з RMI-IIOP можна досягти сумісності
- об'єкти CORBA не підтримують garbage collection. CORBA є незалежним від мови, а деякі мови (C++, наприклад) не підтримують збір сміття. Це можна вважати недоліком, оскільки як тільки об'єкт CORBA створюється, він продовжує існувати до тих пір, поки від нього не позбутися, а вирішити, коли позбутися від об'єкта, не є тривіальним завданням. Водночас, об'єкти RMI підтримують garbage collection.

Більш детальна інформація:

<http://www.oracle.com/technetwork/articles/javase/rmi-corba-136641.html>

[RMI](#)

[CORBA Specification \(OMG\)](#)

[JavaIDL](#)

[Distributed Programming with Java book \(Chapter 11: Overview of CORBA\)](#)

[CORBA Server and Servlet Client](#)

[RMI-IIOP](#)

Java Networking

By The Java Tutorials

<https://docs.oracle.com/javase/tutorial/networking/overview/index.html>

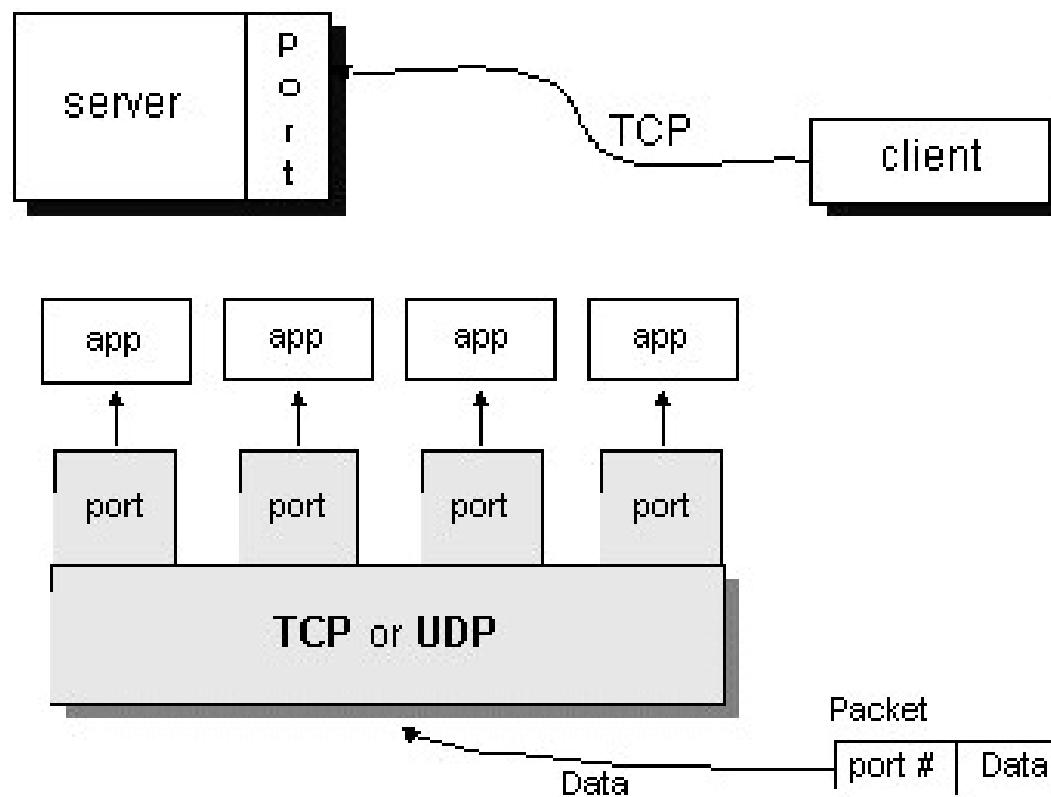
- URL зв'язок (високорівневий)
 - Слугує для зчитування даних з Інтернет-ресурсу
 - Не гарантує надходження даних до клієнта у тому порядку, в якому вони передавались з сервера
- Зв'язок через сокети (низькорівневий)
 - Слугує для програмування клієнт-серверних застосувань
 - Зв'язує дві програми (клієнта і сервера), які виконуються на розподілених ресурсах комп'ютерної мережі, для двостороннього обміну даними

Networking Basics

- **java.net package** provides classes system-independent network communication. However, to decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.
- *TCP (Transmission Control Protocol)* is a connection-based protocol that provides a reliable flow of data between two computers. TCP provides a point-to-point channel for applications that require reliable communications.
- *UDP (User Datagram Protocol)* is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival. UDP is not connection-based like TCP.
- Through the classes in `java.net`, Java programs can use TCP or UDP to communicate over the Internet. The `URL`, `URLConnection`, `Socket`, and `ServerSocket` classes all use TCP to communicate over the network. The `DatagramPacket`, `DatagramSocket`, and `MulticastSocket` classes are for use with UDP.

Port

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.



Working with URLs

- URL (Uniform Resource Locator) is a reference (an address) to a resource on the Internet. Java programs that interact with the Internet also may use URLs to find the resources on the Internet they wish to access. Java programs can use a class called [URL](#) in the `java.net` package to represent a URL address.
- A URL has two main components:

Protocol identifier: For the URL `http://example.com`, the protocol identifier is `http`.

Resource name: For the URL `http://example.com`, the resource name is `example.com`.

Resource name

The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the following components:

- **Host Name**
 - The name of the machine on which the resource lives.
- **Filename**
 - The pathname to the file on the machine.
- **Port Number**
 - The port number to which to connect (typically optional).
- **Reference**
 - A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

For many protocols, the host name and the filename are required, while the port number and reference are optional. For example, the resource name for an HTTP URL must specify a server on the network (Host Name) and the path to the document on that machine (Filename); it also can specify a port number and a reference.

Creating a URL

```
URL myURL = new URL("http://example.com/");
```

```
URL myURL = new URL("http://example.com/pages/");  
URL page1URL = new URL(myURL, "page1.html");  
URL page2URL = new URL(myURL, "page2.html");
```

```
new URL("http", "example.com", "/pages/page1.html");
```

```
new URL("http", "example.com", 80, "pages/page1.html");
```

```
URI uri =  
new URI("http", "example.com", "/hello world/", "");  
URL url = uri.toURL();
```

!!! URLs are "write-once" objects. Once you've created a URL object, you cannot change any of its attributes (protocol, host name, filename, or port number).

URL objects. Methods

```
import java.net.*;
import java.io.*;
public class ParseURL {
    public static void main(String[] args) throws Exception {

        URL url = new
URL("http://example.com:80/docs/books/tutorial"
        +
"/index.html?name=networking#DOWNLOADING");

        System.out.println("protocol = " + url.getProtocol());
        System.out.println("authority = " + url.getAuthority());
        System.out.println("host = " + url.getHost());
        System.out.println("port = " + url.getPort());
        System.out.println("path = " + url.getPath());
        System.out.println("query = " + url.getQuery());
        System.out.println("filename = " + url.getFile());
        System.out.println("ref = " + url.getRef());
    }
}
```

```
protocol = http
authority = example.com:80
host = example.com
port = 80
path = /docs/books/tutorial/index.html
query = name=networking
filename = /docs/books/tutorial/index.html?name=networking
ref = DOWNLOADING
```

Reading Directly from a URL

```
import java.io.*;

public class URLReader {
    public static void main(String[] args) throws Exception {

        URL oracle = new URL("http://www.oracle.com/");
        BufferedReader in =
            new BufferedReader(new InputStreamReader(oracle.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

Connecting to a URL

```
try {
    URL myURL = new URL("http://example.com/");
    URLConnection myURLConnection = myURL.openConnection();
    myURLConnection.connect();
}
catch (MalformedURLException e) {
    // new URL() failed
    // ...
}
catch (IOException e) {
    // openConnection() failed
    // ...
}
```

Reading from a URLConnection

```
import java.net.*;
import java.io.*;

public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL url = new URL("http://www.oracle.com/");
        URLConnection yc = url.openConnection();
        BufferedReader in =
            new BufferedReader(new InputStreamReader(yc.getInputStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

Writing to a URLConnection

```
import java.io.*;
import java.net.*;
public class Reverse {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: java Reverse "+
                "http://<location of your servlet/script>+"+ " string_to_reverse");
            System.exit(1);
        }
        String stringToReverse = URLEncoder.encode(args[1], "UTF-8");

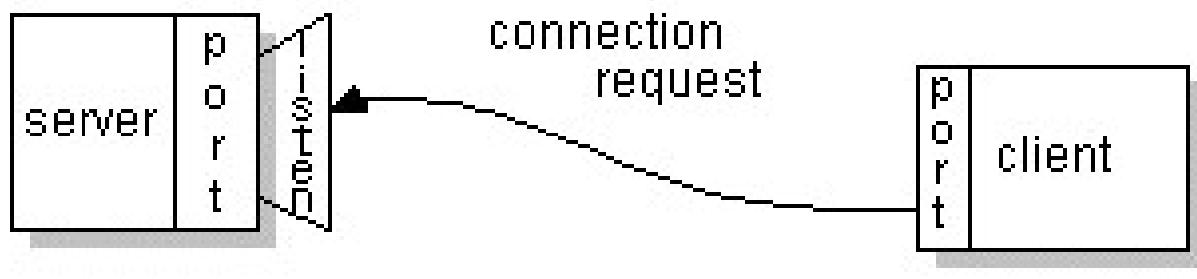
        URL url = new URL(args[0]);
       URLConnection connection = url.openConnection();
        connection.setDoOutput(true);
        OutputStreamWriter out =
            new OutputStreamWriter(connection.getOutputStream());
        out.write("string=" + stringToReverse);
        out.close();

        BufferedReader in =
            new BufferedReader(new InputStreamReader(connection.getInputStream()));

        String decodedString;
        while ((decodedString = in.readLine()) != null) {
            System.out.println(decodedString);
        }
        in.close();
    }
}
```

Socket

- A socket is one end-point of a two-way communication link between two programs running on the network.
- Socket classes are used to represent the connection between a client program and a server program.
- The `java.net` package provides two classes - `Socket` and `ServerSocket` - that implement the client side of the connection and the server side of the connection, respectively.



The client and server can now communicate by writing to or reading from their sockets.

Class Socket

- implements one side of a two-way connection between your Java program and another program on the network
- sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program
- instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion

ServerSocket class

- implements a socket that servers can use to listen for and accept connections to clients

Reading from and Writing to a Socket

The example program implements a client, [EchoClient](#), that connects to an echo server. The echo server receives data from its client and echoes it back. The example [EchoServer](#) implements an echo server. (Alternatively, the client can connect to any host that supports the Echo Protocol.)

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);
try ( // try-with-resources statement
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out = new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader ( new InputStreamReader(echoSocket.getInputStream()) );
    BufferedReader stdIn =
        new BufferedReader ( new InputStreamReader(System.in) )
)
```

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

```
java EchoServer 7
java EchoClient echoserver.example.com 7
```

Example. EchoClient

```
import java.io.*;
import java.net.*;
public class EchoClient {
    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.err.println("Usage: java EchoClient <host name> <port number>");
            System.exit(1);
        }
        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);
        try {
            Socket echoSocket = new Socket(hostName, portNumber);
            PrintWriter out = new PrintWriter(echoSocket.getOutputStream(), true);
            BufferedReader in =
                new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
            BufferedReader stdIn =
                new BufferedReader(new InputStreamReader(System.in))
        } {
            String userInput;
            while ((userInput = stdIn.readLine()) != null) {
                out.println(userInput);
                System.out.println("echo: " + in.readLine());
            }
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                hostName);
            System.exit(1);
        }
    }
}
```

Writing the Server Side of a Socket

The example program implements a client, [EchoClient](#), that connects to an echo server. The echo server receives data from its client and echoes it back. The example [EchoServer](#) implements an echo server. (Alternatively, the client can connect to any host that supports the Echo Protocol.)

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);
int portNumber = Integer.parseInt(args[0]);
try (
    ServerSocket serverSocket = new ServerSocket(Integer.parseInt(args[0]));
    Socket clientSocket = serverSocket.accept();
    PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream())));
)
```

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

Example. EchoServer

```
import java.net.*;
import java.io.*;
public class EchoServer {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }
        int portNumber = Integer.parseInt(args[0]);

        try {
            ServerSocket serverSocket = new ServerSocket(Integer.parseInt(args[0]));
            Socket clientSocket = serverSocket.accept();
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in =
                new BufferedReader( new InputStreamReader(clientSocket.getInputStream()) );
        } {
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                               + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

Running program

```
java EchoServer 7  
java EchoClient echoserver.example.com 7
```

Knock Knock Example

- <https://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html>

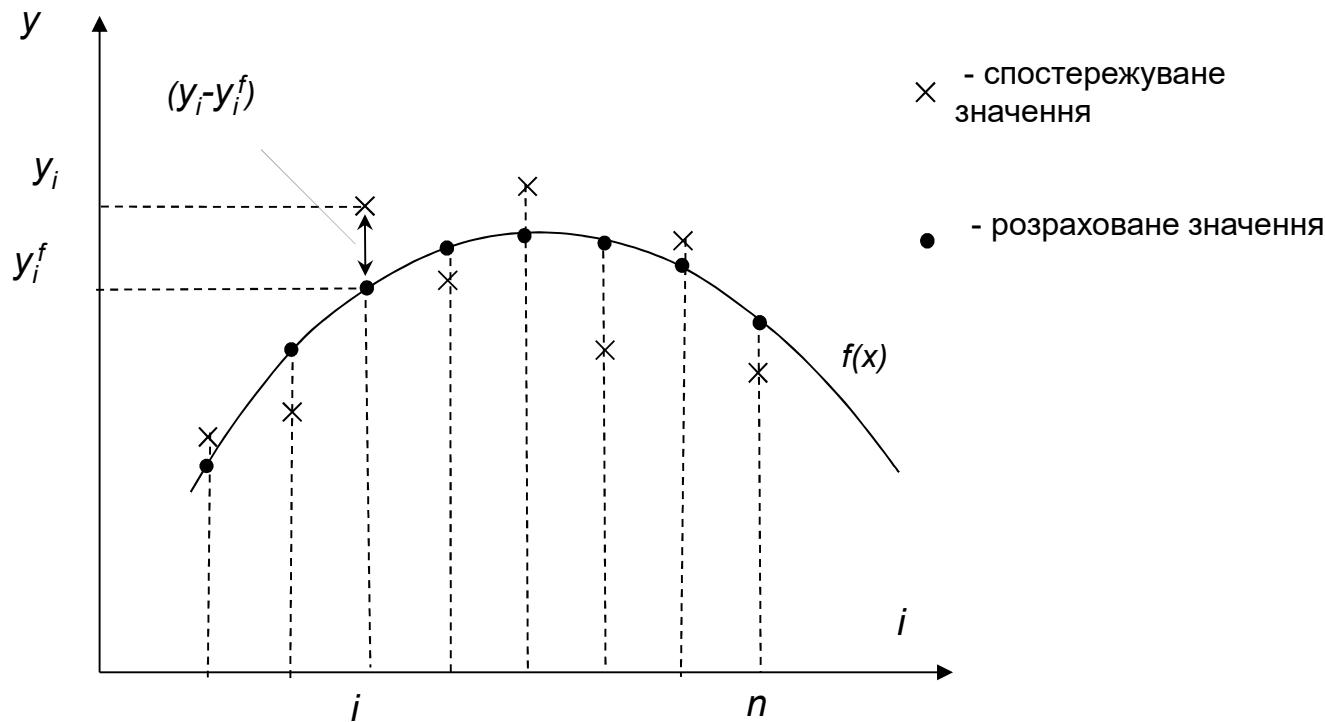
Алгоритми самоорганізації моделей

I.В.Стеценко

Моделі прогнозу

- Прогноз є короткостроковим, якщо час прогнозу складає не більше 10% від часу спостереження
- Довгостроковий прогноз складається на термін від 10% до 100% від часу спостереження

Апроксимація функціональної залежності



Послідовність дій дослідника:

- 1) сформувати масив спостережуваних значень;
- 2) сформувати припущення про вид математичної функції $f(x, b)$, де b – параметри функціональної залежності;
- 3) відшукати значення параметрів функціональної залежності b за критерієм найменших квадратів;
- 4) оцінити якість знайденої функціональної залежності методами багатофакторного кореляційно-регресійного аналізу.

Формування масиву спостережуваних значень

!!!

Задача апроксимації функції багатьох змінних має розв'язок тільки за умови, що змінні x_1, x_2, \dots, x_m не залежать одна від одної

!!!

Потрібно використовувати нормалізовані значення в масиві спостережуваних значень

Формування гіпотези про вид функціональної залежності

!!!

Функціональна залежність має бути лінійною відносно параметрів b_i

$$f(x, b) = b_0 + b_1 f_1(x) + b_2 f_2(x) + \dots + b_k f_k(x),$$

де $x = (x_1, x_2, \dots, x_m)$ – вектор змінних

Оцінка значень параметрів функціональної залежності

Параметри функціональної залежності знаходять з умови забезпечення найменшого значення критерію найменших квадратів:

$$F(b) = \sum_{i=0}^n (f(x_i, b) - y_i)^2 = \sum_{i=0}^n (b_0 + b_1 f_1(x_i) + \dots + b_k f_k(x_i) - y_i)^2 \rightarrow \min$$

Система нормальних рівнянь Гауса

$$\begin{cases} \frac{\partial F}{\partial b_0} = 0 \\ \frac{\partial F}{\partial b_1} = 0 \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial F}{\partial b_k} = 0 \end{cases} \Rightarrow \begin{cases} b_0 n + b_1 \sum_{i=0}^n f_1(x_i) \dots + b_k \sum_{i=0}^n f_k(x_i) = \sum_{i=0}^n y_i \\ b_0 \sum_{i=0}^n f_1(x_i) + b_1 \sum_{i=0}^n f_1(x_i)^2 \dots + b_k \sum_{i=0}^n f_k(x_i) f_1(x_i) = \sum_{i=0}^n y_i f_1(x_i) \\ \cdot \\ \cdot \\ \cdot \\ b_0 \sum_{i=0}^n f_k(x_i) + b_1 \sum_{i=0}^n f_1(x_i) f_k(x_i) \dots + b_k \sum_{i=0}^n f_k(x_i)^2 = \sum_{i=0}^n y_i f_k(x_i) \end{cases}$$

Система умовних рівнянь

$$\begin{cases} y_1 = b_0 + b_1 f_1(x_1) + \dots + b_k f_k(x_1) \\ y_2 = b_0 + b_1 f_1(x_2) + \dots + b_k f_k(x_2) \\ \vdots \\ y_n = b_0 + b_1 f_1(x_n) + \dots + b_k f_k(x_n) \end{cases} \Leftrightarrow X \cdot b = y$$
$$X = \begin{pmatrix} 1 & f_1(x_1) & \dots & f_k(x_1) \\ 1 & f_1(x_2) & & f_k(x_2) \\ \dots & & & \\ 1 & f_1(x_n) & & f_k(x_n) \end{pmatrix}$$
$$b = \begin{pmatrix} b_0 \\ b_1 \\ \dots \\ b_k \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}$$

Система нормальних рівнянь у матричному вигляді

$$\begin{aligned} X \cdot b = y &\Rightarrow X^T \cdot X \cdot b = X^T y \Rightarrow \\ &\Rightarrow b = (X^T \cdot X)^{-1} \cdot X^T y \quad !!! \end{aligned}$$

$$b = \begin{pmatrix} b_0 \\ b_1 \\ \dots \\ b_k \end{pmatrix} \quad X = \begin{pmatrix} 1 & f_1(x_1) & \dots & f_k(x_1) \\ 1 & f_1(x_2) & & f_k(x_2) \\ \dots & & & \\ 1 & f_1(x_n) & & f_k(x_n) \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}$$

Кореляційно-регресійний аналіз функціональної залежності

Індекс кореляції

$$R = \sqrt{\frac{\sigma_{факт}^2}{\sigma_{заг}^2}}$$

$$\sigma_{факт}^2 = \frac{\sum (f(x_i, b) - \bar{y})^2}{n-1} \quad \text{факторна дисперсія}$$

$$\sigma_{заг}^2 = \frac{\sum (y_i - \bar{y})^2}{n-1} \quad \text{загальна дисперсія}$$

!!! Індекс детермінації R^2 характеризує, яка частина загальної варіації результативної ознаки у пояснюється фактором x

$$R^2 > 0,5 \Rightarrow R > 0,7$$

Проблеми застосування методу апроксимації

- Відповіальність за обраний вид функціональної залежності повністю несе дослідник.
- При кількості параметрів більше 5, результат, як правило, не задовільний.
- Немає впевненості, що при іншій множині спостережуваних значень отримаємо ідентичну модель.

!!! Ідея самоорганізації моделей

[Івахненко А.Г., 1985]

1. Формувати множину моделей-претендентів, з якої методом перебору (повного чи неповного) знаходити модель оптимальної складності.
2. Поділити спостережувані значення на дві частини – навчальну та перевірочну послідовності даних. Навчальну використовувати для визначення параметрів моделі-претендента, перевірочну – для оцінки її якості.

Критерій, використовуваний для визначення параметрів моделі-претендента, називається **внутрішнім**.

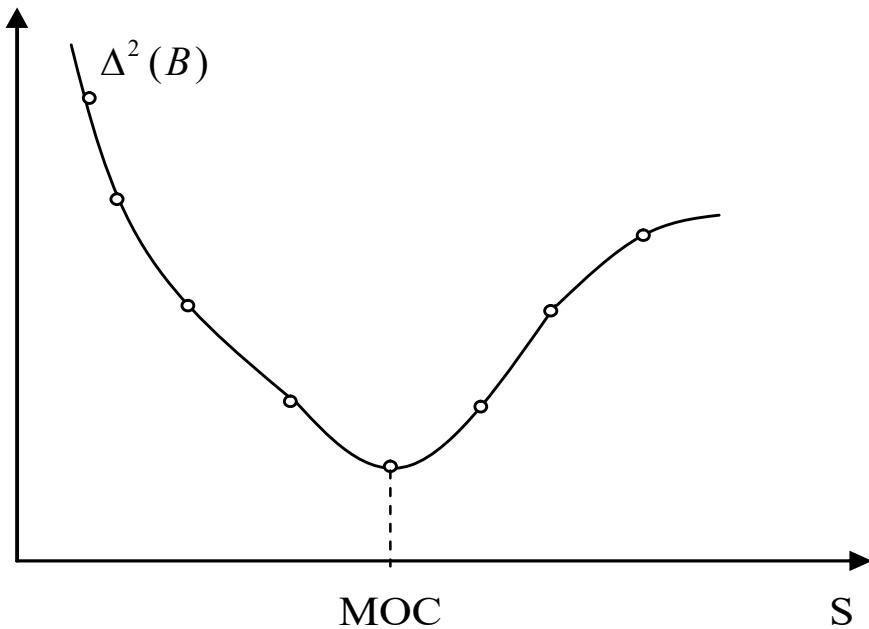
Критерій, використовуваний для оцінки якості моделі-претендента, називається **зовнішнім**.

Принцип самоорганізації моделей

При поступовому збільшенні складності моделі значення зовнішніх критеріїв спочатку падає, досягає мінімуму, а потім або залишається постійним, або починає збільшуватися.

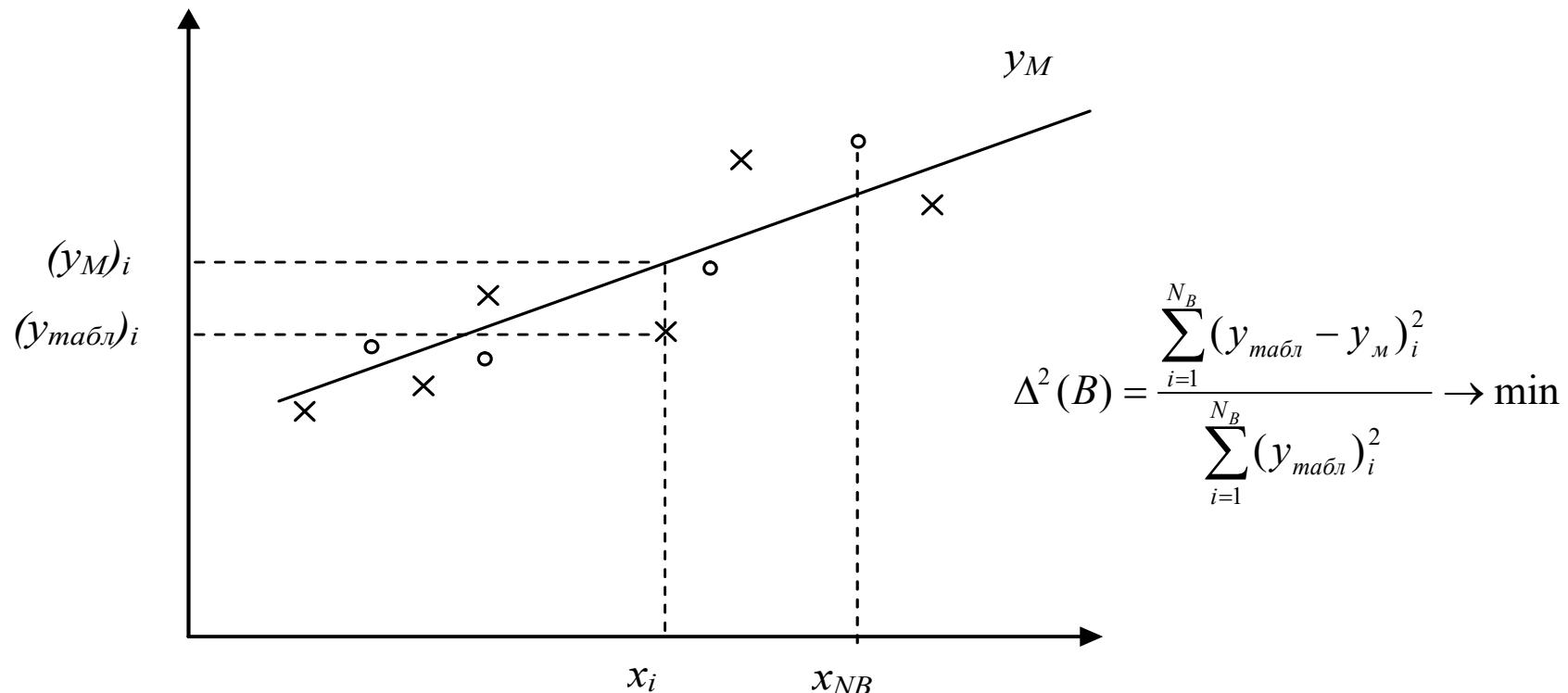
Мінімуму зовнішнього критерія відповідає модель оптимальної складності.

Залежність зовнішнього критерію від складності моделі



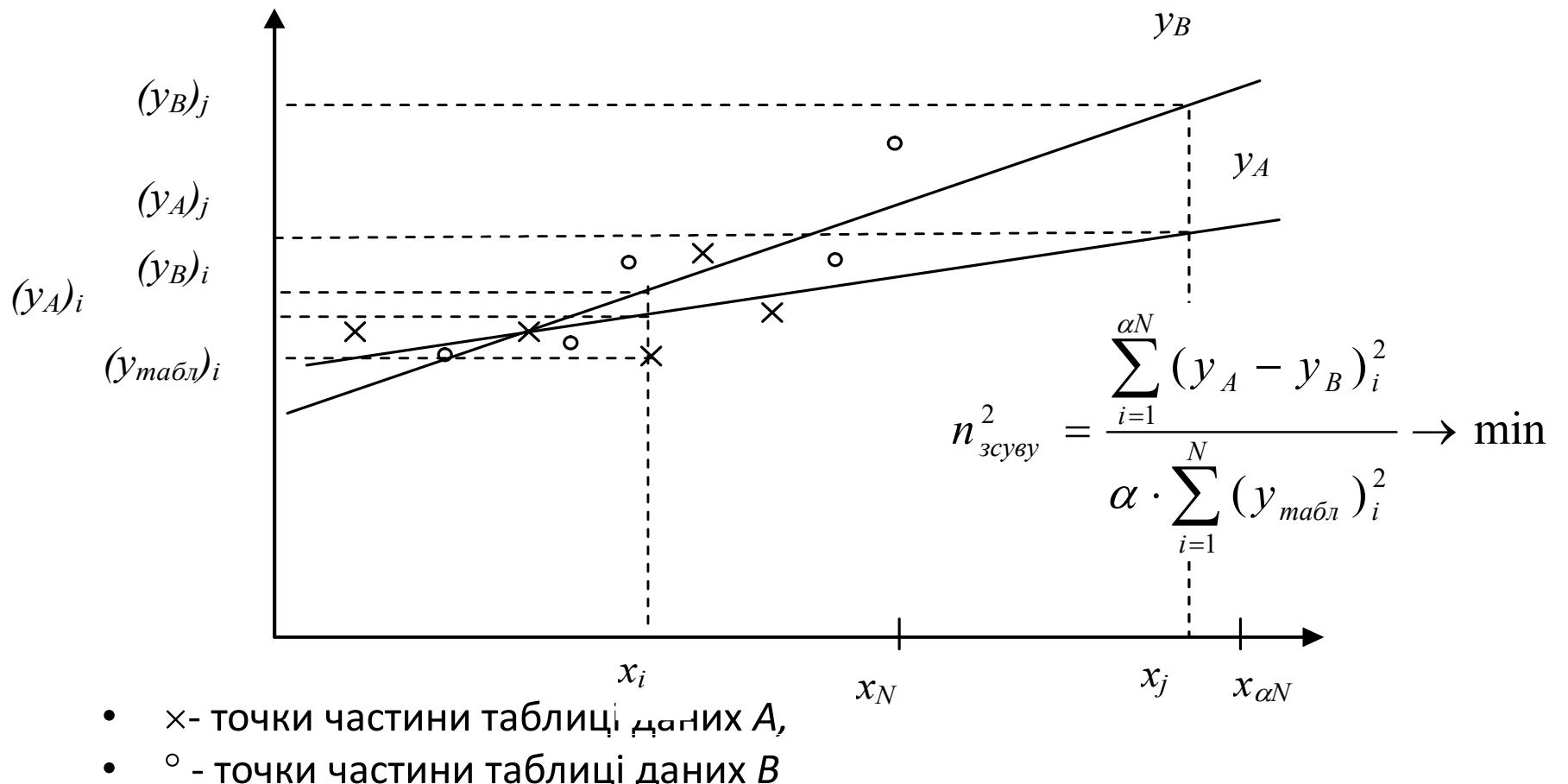
$$\Delta^2(B) = \frac{\sum_{i=1}^{N_B} (y_{\text{мабл}} - y_m)_i^2}{\sum_{i=1}^{N_B} (y_{\text{мабл}})_i^2} \rightarrow \min$$

Розрахунок критерію регулярності



- \times - точки частини таблиці даних A ,
- \circ - точки частини таблиці даних B

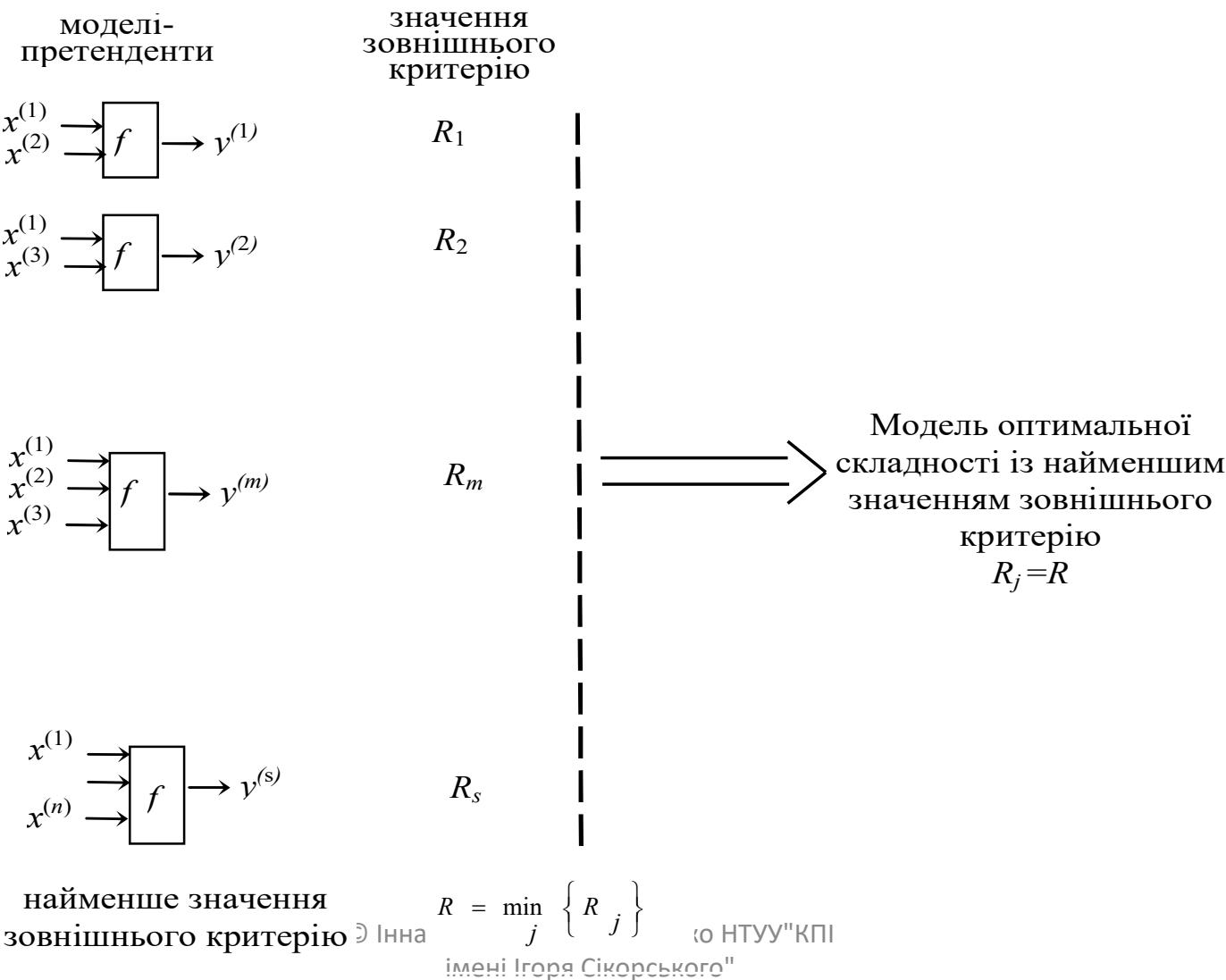
Розрахунок критерію мінімуму зсуву



Алгоритми самоорганізації

- Алгоритми самоорганізації розрізняють:
- за ансамблем зовнішніх критеріїв, використовуваних для пошуку моделі оптимальної складності;
- за набором базисних функцій, використовуваних для побудови моделей-претендентів;
- за способом перебору моделей-претендентів: однорядні (комбінаторні), багаторядні (порогові, селекційні)

Однорядний алгоритм самоорганізації моделей



Вид математичної моделі

Математична модель має бути лінійною по параметрах b :

$$y = b_0 + b_1 f_1(x) + b_2 f_2(x) + b_3 f_3(x) + b_4 f_4(x) + b_5 f_5(x)$$

$f_i(x)$ називають опорними функціями

Вид опорних функцій є припущенням дослідника про вид шуканої моделі.

Вибір опорних функцій

!!! Від правильного вибору опорних функцій в найбільшій мірі залежить успіх алгоритму самоорганізації.

Потрібно пам'ятати, що

- поліноміальні моделі є найбільш універсальним представленням функціональної залежності, але не завжди вірним;
- дробно-раціональна функціональна залежність не може бути представлена лінійною залежністю або поліноміальною залежністю;
- функціональна залежність, що має періодичний характер, представляється тригонометричними функціями.

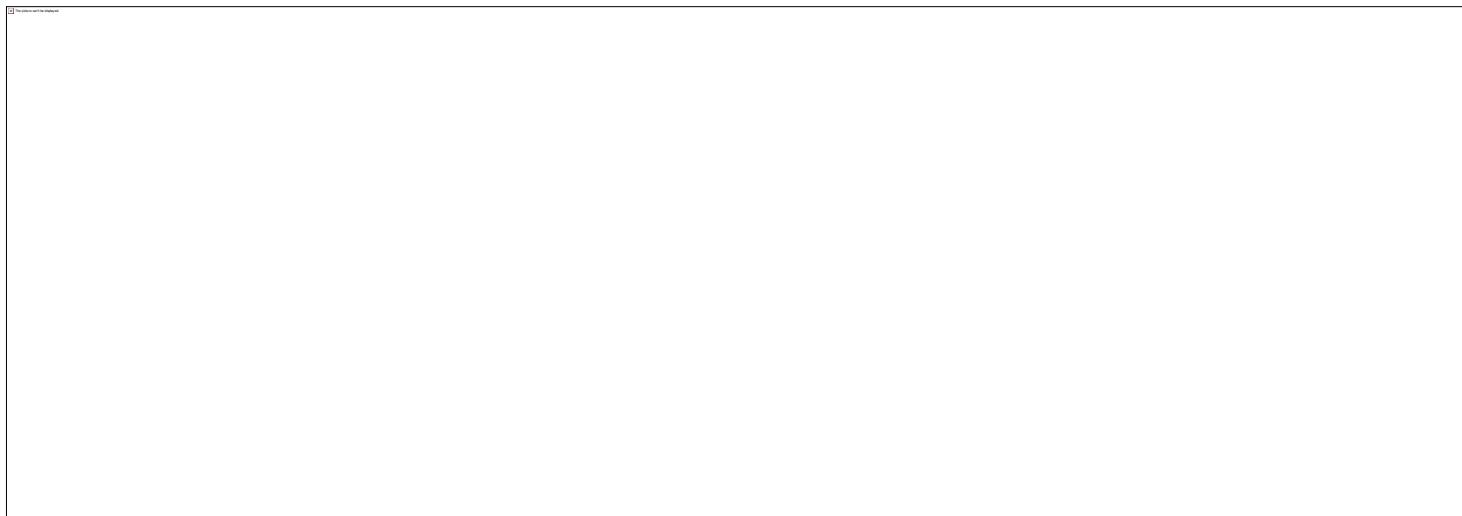
!!! Опорні функції повинні бути лінійно незалежними одна від одної

Формування множини моделей-претендентів

Якщо найскладніша модель має вигляд:



То множина моделей-претендентів складається з $2^3 - 1$ моделей:

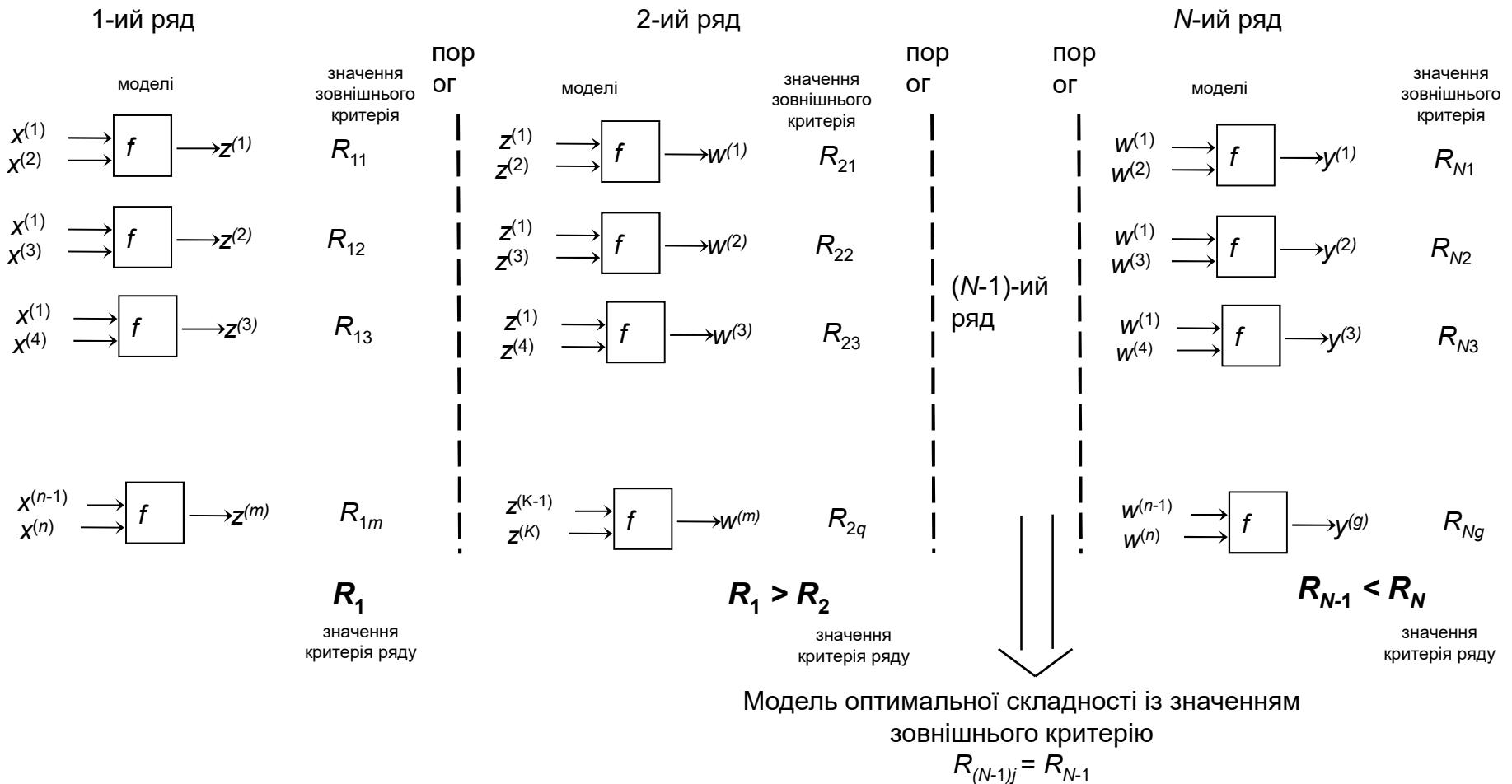


Узагальнена схема однорядного алгоритму



імені Ігоря Сікорського"

Багаторядний алгоритм самоорганізації моделей



Формування результату пошуку моделі оптимальної складності

$$\begin{aligned}y^{(k)} &= b_{03} + b_{13}w^{(i)} + b_{23}w^{(j)} = \\&= b_{03} + b_{13}(b_{02} + b_{12}z^{(m)} + b_{22}z^{(n)}) + b_{23}(b_{02} + b_{12}z^{(g)} + b_{22}z^{(h)}) = \\&= b_{03} + b_{13}(b_{02} + b_{12}(b_{01} + b_{11}x^{(s)} + b_{21}x^{(p)}) + b_{22}(b_{01} + b_{11}x^{(q)} + b_{21}x^{(r)})) + \\&+ b_{23}(b_{02} + b_{12}(b_{01} + b_{11}x^{(u)} + b_{21}x^{(l)}) + b_{22}(b_{01} + b_{11}x^{(v)} + b_{21}x^{(f)}))\end{aligned}$$

Узагальнена схема багаторядного алгоритму



Реалізація однорядного алгоритму

The screenshot shows a Java IDE interface with several windows:

- Проекты**: Shows the project structure with packages `MGUA`, `Matrix`, and `mgua`.
- Источник**: Displays the `MGUA.java` file content. The code imports `javax.swing.JFrame`, `matrix.Matrix`, `matrix.MatrixMathematics`, and `matrix.NoSquareException`. It contains a class `MGUA` with a `main` method that reads data from a file named "VDV2.txt" and prints "Data was red".
- Вывод**: Shows the output of the run command. It includes statistical values for coefficient m_B and a message congratulating the user on having a quality model.

```
import javax.swing.JFrame;
import matrix.Matrix;
import matrix.MatrixMathematics;
import matrix.NoSquareException;

/**
 * @author Inna
 */
public class MGUA {

    /**
     * @param args the command line arguments
     * @throws matrix.NoSquareException
     */
    public static void main(String[] args) throws NoSquareException {
        int n;
        int m;
        String nameOfFile = "VDVResults";
        // double[][] data = getData();

        double[][] data = readData("VDV2.txt", n, m);
        System.out.println("Data was red");
    }

    private static double[][] getData() {
        return null;
    }

    private static void drawDiagram(double[] xCoord, double[] yCoord) {
        // Implementation
    }

    private static void printData(double[][] data) {
        // Implementation
    }
}
```

run:
Data was red
Models with least squares criterion, regularization criterion and unbaisedness criterion
values of coefficient m_B :
7.762818228654461E-4
0.09012373015430875
0.2153208416081817
0.3672954364718133
0.23933896723088255

We have such results:
 $f(x) = 7.762818228654461E-4 + 0.09012373015430875 x_2 + 0.2153208416081817 x_4 + 0.3672954364718133 x_6 + 0.23933896723088255 x_8$
Criterion = 0.0016605295371870245
Congratulations! You have a quality model
values Ymod :
0.059243272832862
0.12042357871093776
0.1565866377037672
-0.15031823659632335
0.1339104912494158
7.762818228654461E-4
0.10909182999333908
0.0841692466087946
0.0873958729153839
0.08094474293609324
7.762818228654461E-4

СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 30 минуты 11 секунды)

Приклад прогнозу середньодобової температури за даними метеостанції Національного природничого парку «Вижницький»

01.01.2002	-8.5	-6	-11
02.01.2002	-8.5	-5	-12
03.01.2002	-8.5	-5	-12
04.01.2002	-8.5	-4	-13
05.01.2002	-10	-5	-15
06.01.2002	-10	-5	-15
07.01.2002	-7.5	-3	-12
08.01.2002	-7.5	-4	-11
09.01.2002	-7.5	-5	-10
10.01.2002	-8	-6	-10
11.01.2002	-7.5	-4	-11
12.01.2002	-7	-4	-10
13.01.2002	-6	-3	-9
14.01.2002	-5.5	-3	-8
15.01.2002	-5	-3	-7
16.01.2002	-5	-3	-7
17.01.2002	-3.5	-2	-5
18.01.2002	-2.5	-1	-4
19.01.2002	-2.5	-1	-4
20.01.2002	-1.5	0	-3
21.01.2002	4.5	8	1
22.01.2002	4.5	8	1
23.01.2002	5.5	9	2
24.01.2002	7.5	12	3
25.01.2002	7.5	12	3
26.01.2002	10.5	17	4
27.01.2002	9	14	4
28.01.2002	8.5	15	2
29.01.2002	8	15	1
30.01.2002	8	14	2
31.01.2002	9	15	3
01.02.2002	9	15	3
02.02.2002	10.5	17	4
03.02.2002	10	16	4
04.02.2002	9	15	3
05.02.2002	9	15	3
06.02.2002	8	14	2
07.02.2002	8	14	2
08.02.2002	7.5	14	1

Середньодобова,
максимальна та мінімальна
температура за рік

Формульовання гіпотези

Опорні функції: тригонометричні з періодом рік, сезон, місяць, “лунний” місяць та поліноміальні.

Зовнішній критерій: сума критеріїв регулярності та мінімуму зсуву

Модель для даних 2003 року

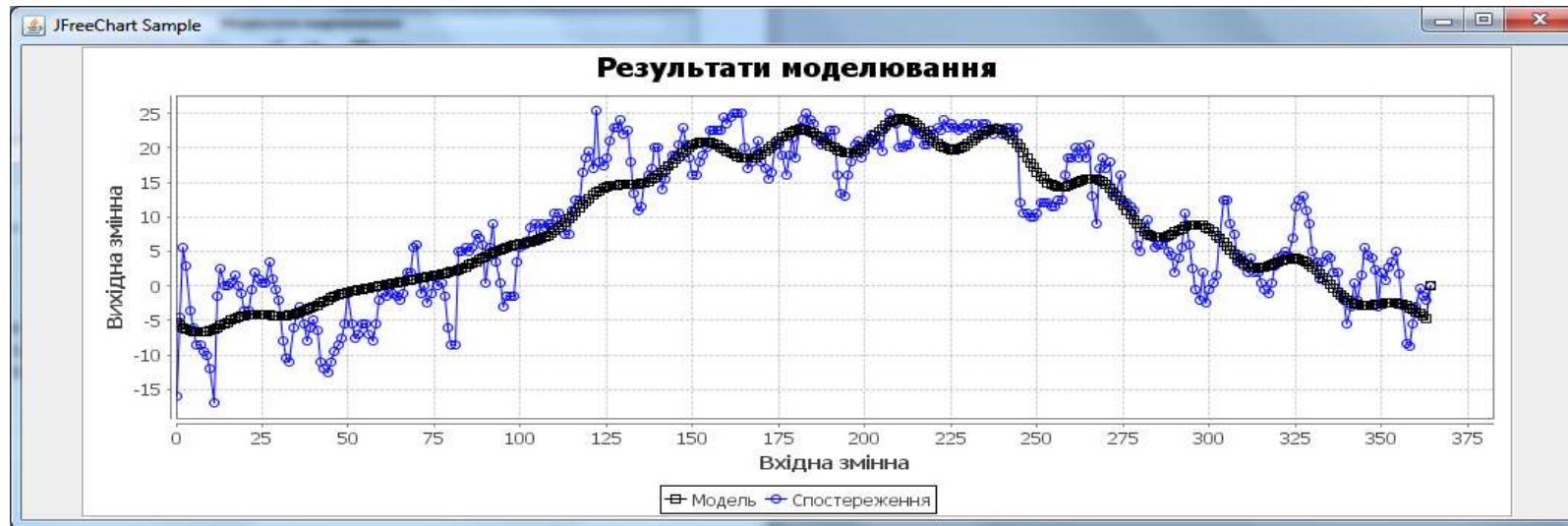


We have such results for 2003:

$$\begin{aligned} f(t) = & 15,4547 - 18,2040 \cos(2\pi t/365) - 2,3276 \sin(2\pi t/365) + \\ & + 0,8041 \cos(\pi t^2/24/365) - 0,7721 \sin(\pi t^2/24/365) - \\ & - 1,3610 \cos(\pi t^8/365) - 0,1083 \sin(\pi t^8/365) - \\ & - 1,0569 \cos(\pi t/14) - 0,2658 \sin(\pi t/14) - \\ & - 47,8375*t/365.0 + 52,2220*t^2/365.0^2 \end{aligned}$$

$$\text{Criterion} = 0.0451749581116046$$

Модель для даних 2003 року

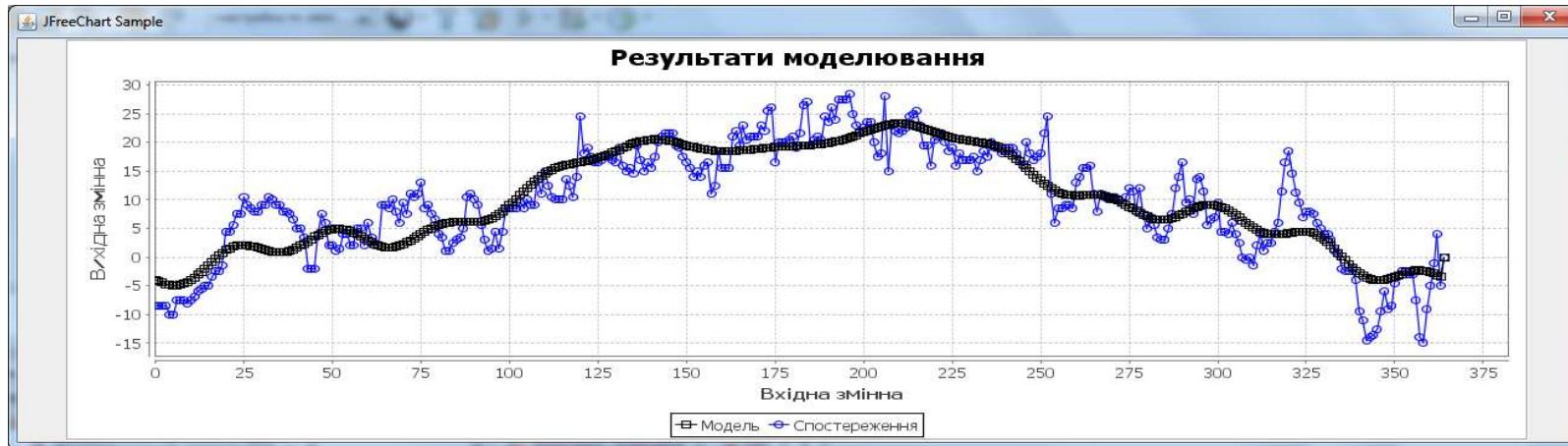


We have such results if only trigonometric functions:

$$\begin{aligned} f(t) = & 8,9232 - 12,9529 \cos(2\pi * t/365) - 3,7227 \sin(2\pi * t/365) + \\ & + 0,8011 \cos(\pi * t*24/365) - 0,8789 \sin(\pi * t*24/365) - \\ & - 1,0704 \cos(\pi * t*8/365) - 0,4550 \sin(\pi * t*8/365) - \\ & - 1,0377 \cos(\pi * t/14) - 0,3642 \sin(\pi * t/14) \end{aligned}$$

$$\text{Criterion} = 0.04907202492564103$$

Дослідження даних 2002 та 2012 року



2002 year

We have such results:

$$f(t) = 9,9124 - 11,6301 \cos(2\pi t/365) - 0,8718 \sin(2\pi t/365) - \\ - 1,1021 \sin(\pi t^{24}/365) - \\ - 2,2003 \cos(\pi t^8/365) + 1,3976 \sin(\pi t^8/365) - \\ - 0,6777 \sin(\pi t/14)$$

Criterion = 0.0470184442800699

Congratulations! You have a quality model

Дослідження даних 2002 та 2012 року



2012 year

We have such results:

$$\begin{aligned} g(t) = & 10,0313 - 12,9336 \cos(2\pi t/365) - 3,7513 \sin(2\pi t/365) + \\ & + 1,0430 \cos(\pi t * 24/365) - 1,0747 \sin(\pi t * 24/365) + \\ & + 1,3700 \cos(\pi t * 8/365) \end{aligned}$$

$$\text{Criterion} = 0.049472896780342605$$

Congratulations! You have a quality model

ВИСНОВКИ

1. Методи самоорганізації є потужним інструментом побудови математичних моделей складних систем
2. Реалізація універсальними мовами програмування надає можливість використовувати математичні моделі систем у якості компонентів інформаційних систем
3. Є невирішені проблеми застосування методів самоорганізації, які можуть стати об'єктом наукової роботи магістра: вибір опорних функцій, визначення множини зовнішніх критеріїв.