НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_Кафедра обчислювальної техніки\_\_\_\_\_\_\_\_\_\_\_\_

(повна назва кафедри, циклової комісії)

**КУРСОВА РОБОТА**

з дисципліни «Паралельні та розподілені обчислення»

(назва дисципліни)

на тему: «Розробка програмного забезпечення для паралельних комп’ютерних систем»

Студента 3 курсу групи ІО-24

напряму підготовки 050102 «Комп’ютерна інженерія»

\_\_\_\_\_\_\_\_\_\_\_\_Репети Я. Г.\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище та ініціали)

Керівник доцент Корочкін О.В.

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_

Оцінка: ECTS \_\_\_\_\_\_\_\_\_\_\_

Члени комісії \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали

Київ- 2015 рік

Національний технічний університет України

“Київський політехнічний інститут”

Факультет (інститут) інформатики та обчислювальної техніки

( повна назва )

Кафедра обчислювальної техніки

( повна назва )

Освітньо-кваліфікаційний рівень бакалавр

Напрям підготовки 6.050102 «Комп’ютерна інженерія»

*(шифр і назва)*

***З А В Д А Н Н Я***

НА КУРСОВУ РОБОТУ СТУДЕНТУ

Репеті Ярославу Григоровичу\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(прізвище, ім’я, по батькові)*

1. Тема роботи «Розробка програмного забезпечення для паралельних

комп’ютерних систем»

керівник роботи Корочкін Олександр Володимирович к.т.н.**,** доцент

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Строк подання студентом роботи 18 травня 2014 р.

3. Вхідні дані до роботи

- порівняння засобів роботи із процесами в мовах Ада і Java

- математична задача

- структури ПКС ОП та ПКС ЛП

- мови і бібліотеки програмування:

- засоби організації взаємодії процесів:

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

- огляд засобів роботи з процесами в мові Ада

- розробка і тестування програми ПРГ1 для ПКС ОП

- розробка і тестування програми ПРГ2 для ПКС ЛП

5. Перелік графічного матеріалу

- структурна схема ПКС ОП

- структурна схема ПКС ЛП

- схеми алгоритмів процесів і головної програми для ПРГ1

- схеми алгоритмів процесів і головної програми для ПРГ2.

7. Дата видачі завдання 10.02.15

***КАЛЕНДАРНИЙ ПЛАН***

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строк виконання етапів КР |
| 1 | Виконання Розділу 1 | 23.02.2015 |
| 2 | Виконання Розділу 2 | 23.03.2015 |
| 3 | Виконання Розділу 3 | 23.04.2015 |
| 4 | Тестування програм ПРГ1 та ПРГ2 | 10.05.2015 |
| 7 | Оформлення КР | 17.05.2015 |
| 8 | Захист КР | 18.05.2015 |

**Студент \_\_\_\_\_\_\_\_\_ \_\_\_\_**Репета Я. Г.\_\_\_\_\_\_\_\_

( підпис ) (прізвище та ініціали)

**Керівник роботи \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_**Корочкін О. В.\_\_\_\_\_\_

( підпис ) (прізвище та ініціали)

**ЗМІСТ**

[ВСТУП 3](#_Toc417593373)

[РОЗДІЛ 1. ПОРІВНЯННЯ ЗАСОБІВ РОБОТИ ІЗ ПРОЦЕСАМИ В МОВАХ АДА І JAVA 3](#_Toc417593374)

[1.1 Процеси в мові Ада 3](#_Toc417593375)

[1.2 Процеси в мові Java 3](#_Toc417593376)

[1.3 Методи роботи із потоками 3](#_Toc417593377)

[1.3.1 Пріоритет процесу 3](#_Toc417593378)

[1.3.2 Затримка виконання процесу 3](#_Toc417593379)

[1.3.3 Примусове завершення процесу 3](#_Toc417593380)

[1.4 Змінні для синхронізації 3](#_Toc417593381)

[1.4.1 Змінні, що розділяються в Ада 3](#_Toc417593382)

[1.4.2 Змінні, що розділяються в Java 3](#_Toc417593383)

[1.5 Монітори 3](#_Toc417593384)

[1.5.1 Монітори в Ада 3](#_Toc417593385)

[1.5.2 Монітори в Java 3](#_Toc417593386)

[1.6 Висновки до розділу 1 3](#_Toc417593387)

[РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС СП 3](#_Toc417593388)

[2.1. Розробка паралельного математичного алгоритму 3](#_Toc417593391)

[2.2. Розробка алгоритмів процесів 3](#_Toc417593392)

[2.3. Розробка схеми взаємодії процесів 3](#_Toc417593393)

[2.4. Розробка програми ПРГ1 3](#_Toc417593394)

[2.5. Тестування програми ПРГ 1 3](#_Toc417593395)

[2.6. Висновки до Розділу 2 3](#_Toc417593396)

[РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП 3](#_Toc417593397)

[3.1. Розробка паралельного математичного алгоритму 3](#_Toc417593400)

[3.2. Розробка алгоритмів процесів 3](#_Toc417593401)

[3.3. Розробка схеми взаємодії процесів 3](#_Toc417593402)

[3.4. Розробка програми ПРГ2 3](#_Toc417593403)

[3.5. Тестування програми ПРГ 2 3](#_Toc417593404)

[3.6. Висновки до Розділу 3 3](#_Toc417593405)

ОСНОВНІ РЕЗУЛЬТАТИ І ВИСНОВКИ…………………………………

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 3](#_Toc417593406)

# ВСТУП

# ИНТЕРВАЛ ВЕЗДЕ 1,5!!!!! ШРИФТ Times New Roman!!!!

У даній роботі розглянуто порівняння створення та використання процесів в мовах програмування Ада та Java. Також розглянуто властивості процесів у цих мовах та різні засоби для побудови паралельних програм.

Зокрема, розглянуто засоби створення процесів в мові Ада, їхні властивості та структура. Структура потоків в мові Java, два можливих способи їх створення.

Розглянуто можливості роботи, які властиві потокам в Java та процесам в Ада, такі як: пріоритетність потоку та методи за допомогою яких її можна отримати та встановити, можливість тимчасового призупинення виконання, можливість виходу із потоку до його логічного завершення.

Розглянуто атомарні змінні, що вони виконують та способи їх об’явлення та методи роботи із ними у мовах Java та Ада. А також монітори, способи їх створення та їхнє призначення у обраних мовах.

Робиться порівняння коефіцієнтів прискорення та ефективності використання в залежності від розмірностей даних та від кількості наявних робочих ядер комп’ютера.

В першому роздіді……………..

В другому розділі…………….

В третьому розділі………………..

# РОЗДІЛ 1. ПОРІВНЯННЯ ЗАСОБІВ РОБОТИ ІЗ ПРОЦЕСАМИ В МОВАХ АДА І JAVA

У даному розділі розглянуто приклади створення нових процесів у мовах Ада та Java. Показано та пояснено основні їхні складові.

### Процеси в мові Ада

В мові програмування Ада процеси створюються за допомогою задач [1]. Конструкція задач не є унікальною, тому що вона має опції, які також властиві для пакетів, процедур та структур даних:

* Як і структура даних задача має тип, тому є можливість існування її, як змінної цього типу.
* Задача має розділ опису та виконувану частину, що робить її подібною до процедури, однак вона не може бути викликана, як процедура. Взамін цьому, вона починає виконання автоматично, як частина структури, в якій описана.
* Подібно пакету, задача має тіло та специфікацію, але вона не може бути самостійною компільованою частинкою, яка може бути поміщена в окремий файл із сирцевим кодом. Взамін цьому, вона повинна бути поміщена в іншу структуру.

Специфікація задачі починається словами зарезервованими словами “*task type”,* які визначаються, що об’явлено тип задачі. Значення об’єкта типу задачі представляє собою задачу. Також, існують задачі, специфікації яких не містять зарезервовано слова “*type*”. Подібна задача представляє собою одиночну задачу, причому опис такої задачі рівносильний опису анонімного типу задачі із одночасним описом об’єкту цього типу.

Нижче наведено приклад простої програми:

**with** Ada.Text\_IO;

**procedure** Tasks **is**

**task**  Task1;

**task** **body** Task1 **is**

**begin**

Ada.Text\_IO.Put\_Line(“Task1”);

**end** Task1;

**task** **type** Task2;

**task body** Task2 **is**

**begin**

Ada.Text\_IO.Put\_Line(“Task2”);

**end** Task2;

**begin**

**null**;

**end**  Tasks;

У даній програмі створюється дві задачі, причому *Task1* має анонімний тип, який був раніше описаний. На відміну від процедур, які потрібно викликати, щоб вони виконалися, задачі виконуються автоматично, причому починають своє виконання одразу після операції *begin,* яка відноситься до процедури в якій знаходиться задача. Частина програмного коду, яка підлягає до виконання задачею знаходиться між операторами  *end* і *begin,* які відносяться до тіла цієї задачі.

### Процеси в мові Java

В мові програмування Java процес можна створити за допомогою потоків [2 ]. Створити потік можна реалізуючи об’єкт класу *Thread*. Існує два способи виконання цієї операції:

* За допомогою реалізації інтерфейсу *Runnable*
* За допомогою розширення класу *Thread*

У першому випадку потрібно об’явити клас, який реалізує інтерфейс *Runnable*. Даний інтерфейс абстрагує одиницю виконуваного коду. Можливе створення потоку із будь-якого об’єкту, який реалізує інтерфейс *Runnable*. Для того, щоб реалізувати даний інтерфейс клас повинен об’явити єдиний метод – *run()*. Всередині цього методу повинен міститися код, який складає новий потік. Єдине, що його відрізняє від звичайного методу є те, що він встановлює точку входу іншого паралельного потоку у програмі. Коли потік завершується метод *run()* повертає управління.

Після того як буде об’явлений клас, що реалізує інтерфейс *Runnable* потрібно створити об’єкт типу *Thread* із цього класу. Клас *Thread* має кілька конструкторів. В поточному випадку використовується:

*Thread(Runnable об’єкт\_потоку, String ім’я\_потоку),*

де *об’єкт\_потоку* – це екземпляр класу, який реалізує інтерфейс *Runnable.* Він визначає, де починається виконання потоку. Ім’я нового потоку передається за допомогою параметра *ім’я\_потоку.*

Після того як новий потік буде створено, він не буде запускатися до тих пір, поки не буде викликано метод  *start(),* об’явлений в класі *Thread.* Можна сказати, що метод *start(),* виконує виклик методу *run().*

Нижче приведений приклад створення потоку за допомогою інтерфейсу *Runnable*:

**class** Thread1 **implements** Runnable {

Thread t;

Thread1() {

t = **new** Thread(**this**, “Потік”);

t.start();

}

**public** **void**  run() {

System.out.println(“Виконується створений потік”);

}

}

**class** Main {

**public static void** main(String args[]) {

**new** Thread1();

}

}

При виборі іншого способу створення потоку потрібно об’явити клас, який буде розширювати клас *Thread,* а потім створити екземпляр цього класу. Цей клас обов’язково повинен перевизначити метод *run()*, який є точкою входу для нового потоку. Він також повинен визвати метод *run()* для запуску виконання нового потоку.

Нижче приведено приклад створення потоку за допомогою класу *Thread:*

**class** Thread2 **extends** Thread {

Thread2() {

super( “Потік”);

start();

}

**public** **void**  run() {

System.out.println(“Виконується створений потік”);

}

}

**class** Main {

**public static void** main(String args[]) {

**new** Thread2();

}

}

## Методи роботи із потоками

### Пріоритет процесу

У мовах Java та Ада кожен процес має свій пріоритет. Завдяки цьому в деяких ситуаціях при зверненні кількох процесів до одного ресурсу буде формуватися черга, у якій пріоритет буде грати першочергове значення.

У мові Ада задача може володіти своїм пріоритетом, який задається за допомогою директиви компілятора *Priority:*

**pragma**  priority ( exp );

де значення *exp* використовується безпосередньо для вказування пріоритету. Це значення може приймати лише цілі значення від одиниці до семи. Приклад використання директиви показаний нижче:

**task** Task1 **is**

**pragma** Priority (5);

. . .

**end** Task1;

Також пріоритет задачі може бути встановлений динамічно, тобто в процесі виконання програми. Це представляється можливим зробити за допомогою стандартного пакету *Ada.Dynamic\_Priorities,* який є доволі простим і має наступний вигляд:

**with** System;

**with** Ada.Task\_Identification;

**package** Ada.Dynamic\_Priorities **is**

**procedure** Set\_Priority

(Priority : **in** System.Any\_Priority;

T : **in** Ada.Task\_Identification.Task\_ID :=

Ada.Task\_Identification.Current\_Task);

**function** Get\_Priority

(T : Ada.Task\_Identification.Task\_ID :=

Ada.Task\_Identification.Current\_Task)

**return** System.Any\_Priority;

**end** Ada.Dynamic\_Priorities;

В Java деякі конструктори класу *Thread* включають в себе можливість встановлення пріоритету при створені потоку. Також у цьому класі визначені два методи для роботи із пріоритетом. Нижче вони наведені (для встановлення та отримання значення пріоритету відповідно):

**final void** setPriority (**int** пріоритет)

**final int** getPriority ()

Пріорітети в Java можуть мати значення від одного до десяти.

### Затримка виконання процесу

Для різних цілей інколи при роботі процесу потрібно його затримати на деякий час. У мовах Ада та Java присутні засоби для виконання цієї операції.

У мові програмування Ада розрізняють два види інструкцій затримки виконання [3]:

* відносна затримка виконання
* абсолютна затримка виконання

Загальний вид інструкції відносної затримки виконання наступний:

**delay**  час\_затримки;

Де *час\_затримки* визначає кількість секунд на яку потрібно призупинити виконання задачі починаючи із поточного моменту. Наприклад, призупинення на п’ять секунд виглядатиме:

**delay**  5.0;

Загальний вид інструкції абсолютної затримки такий:

**delay until**  момент\_часу;

Де *момент\_часу* визначає точний момент часу до якого потрібно зробити призупинення виконання задачі. Наприклад, призупинення до першого березня 2015 року матиме вигляд:

**delay until**  Time\_Of (2015, 3, 1, 0.0);

У мові програмування Java для таких цілей існує метод *sleep(),* який визначений у класі *Thread*. Він має дві форми:

**static void** sleep(**long** миллисекунды) **throws** InterruptedException

**static void** sleep (**long** миллисекунды, **long** наносекунды) **throws**

InterruptedException

За допомогою першої можна вказати затримку часу в мілісекундах, а наступна форма більша розширена і дозволяє вказати час затримки з точністю до наносекунд.

### Примусове завершення процесу

При роботі з процесами інколи виникає потреба достроково виходу із нього. У мовах Ада та Java для цього існують досить прості способи.

У мові Ада виконати дострокове завершення об’єкту задачі можна за допомогою інструкції зупинення, яка може мати наступний вигляд:

**abort** Task;

де *Task* є ім’ям будь-якого об’єкту задачі. За допомогою цієї інструкції задача припиняє виконання будь-яких інструкцій у тілі задачі, за виключенням тих, які викликають операції, відкладені до примусового завершення *(abort deffered operations).*

У мові програмування Java клас *Thread* визначає метод *stop(),* який зупиняє потік. Його сигнатура виглядає так:

**final void** stop();

Зупинений потік не може бути відтворений.

## Змінні для синхронізації

### Змінні, що розділяються в Ада

У мові Ада для вказівки змінних, що поділяються використовують наступні директиви компілятора:

**pragma** Atomic ( *Local\_Name* ) ;

**pragma** Atomic\_Components ( *Local\_Array\_Name* ) ;

**pragma** Volatile ( *Local\_Name* ) ;

**pragma** Volatile\_Components ( *Local\_Array\_Name* ) ;

Тут *Local\_Name* вказує локальне ім’я об’єкту або опис типу, а *Local\_Array\_Name* ­вказує локальне ім’я об’єкту-масиву або опис типу-масиву.

Директиви компілятора *Atomic* і *Atomic\_Components* забезпечують неперервність операцій читання/запису для всіх вказаних в них об’єктах. Такі об’єкти називають атомарними, а операції над ними виконуються лише послідовно.

Директиви компілятора *Volatile* і *Volatile\_Components* забезпечує виконання операцій читання/запису для всіх вказаних в них об’єктах безпосередньо в пам’яті.

Приклади використання цих директив компілятора можуть мати наступний вид:

Element : Positive;

**pragma** Atomic ( Element ) ;

**pragma** Volatile ( Element ) ;

Array **is array** ( 1 . . Element) **of Integer** ;

**pragma** Atomic\_Components (Array) ;

**pragma** Volatile\_Components ( Array ) ;

Змінні, що поділяються і перечисленні для них директиви компілятора можуть бути використані для організації:

* взаємодії задач
* взаємодії Ада-програми із іншими процесами
* управління пристроями із Ада-програми

### Змінні, що розділяються в Java

У мові програмування Java є аналогічні засоби для доступу до змінних. В Java існує ключове слово *volatile*. Модифікатор *volatile* повідомляє компілятору, що позначена ним змінна може бути несподівано змінена іншими частинами вашої програми. Одна з таких ситуацій виникає в багатопоточних програмах. В багато поточних програмах іноді два або більше потоків мають спільний доступ до однієї і тієї ж змінної. З міркувань ефективності, кожен потік може зберігати свою власну закриту копію цієї змінної. Реальна копія змінної оновлюється в різні моменти, наприклад при вході в метод *synchronized*. Хоча такий підхід працює нормально, все ж іноді він недостатньо ефективний. У деяких випадках все, що дійсно відбувається - це те, що майстер-копія змінної завжди відображає її поточний стан. Щоб забезпечити це, просто оголосіть змінну як *volatile*, що повідомить компілятору про необхідність завжди використовувати майстер-копію цієї змінної або ж, як мінімум, завжди тримати закриті її копії синхронізованими з майстер-копією і навпаки. Крім того, доступ до майстер-копії змінної повинен здійснюватися в тому ж порядку, як він виконувався до закритої копії. Загалом реалізація є подібною до реалізації в мові Ада. Для прикладу:

**volatile int** x;

В Java також присутні атомарні змінні. У пакет *java.util.concurrent.atomic* входять 9 видів атомарних змінних: *AtomicInteger*, *AtomicLong*, *AtomicReference*, *AtomicBoolean* і так далі. Застосувати їх можна створивши екземпляр відповідного класу. Для прикладу:

AtomicLong Number = **new** AtomicLong(0);

## Монітори МАЛО!!!!!!!

### Монітори в Ада

Концепцію монітора реалізовано за допомогою захищених модулів. Захищені модулі включають в себе властивості монітора, тому що включають в себе захищені операції (захищені функції; захищені процедури; захищені входи), тому реалізувати монітор можна просто реалізувавши захищений модуль. ……………………

### Монітори в Java

В мові Java не має спеціальних засобів для реалізації моніторів, таких як захищені модулі в мові Ада. Для їх реалізації необхідно створити клас із методами та використовувати модифікатори *synchronized* (що дозволить виконувати методи лише одному потокові) та private (для інкапсуляції спільного ресурсу). Монітор в Java можна реалізувати наступним чином:

**class** Monitor {  
 **private** double resource;

**synchronized** double read() {

**return** resource;

}

**synchronized void** write(double x) {

resource = x;

}

}

# Висновки до розділу 1

1. У мові Ада паралельна ділянка створюється за допомогою типу задачі – “Task type”. Всі задачі мають свій пріоритет, який можна змінювати. Можна зробити затримку задачі на певний час або до певного моменту. Існує дострокове завершення процесу. Для вирішення завдання взаємного виключення використовуються Atomic та Volatile змінні. Концепцію монітора реалізовано за допомогою захищеного модуля.
2. У мові Java паралельні ділянки називають потоками і створюють їх за допомогою наслідування класу Thread або реалізації інтерфейсу Runnable. Потоки в Java мають пріоритетність виконання. Затримки в Java мають високу точність. Можливо перервати виконання потоку.
3. Ада та Java мають засоби для роботи із процесами. Обидві мови підтримуються пріоритетність процесів. В Ада затримка задачі може бути реалізована до певної дати, а в Java затримка проходить на певний час із точністю до наносекунд. Обидві мови мають засіб для передчасного завершення процесу та підтримують змінні, що поділяються. Також обидві мови не мають власних типів данних для реалізації монітора, однак цю концепцію легко вирішують за допомогою інших засобів.
4. Рішення завдання взаємного виключення…..
5. Рішення завдання синхронізації
6. Порівняння…..

**РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС СП**

В розділі розглядається розробка програми ПРТ1 для ПКС СП

Математична задача:



Мова програмування: C#

Засоби взаємодії процесів: семафори, події, механізм lock

Структура ПКС СП представлена на рис 2.1:



ОП

1

P

. . .

Рис. 2.1 Структура ПКС СП

**2.1. Розробка паралельного математичного алгоритму**

Згідно з технічним завданням необхідно розробити паралельний алгоритм. Він складається із одного етапу:



Пояснення до використовуваних констант:

*  – розмірність векторів і матриць;
*  – кількість ядер;
* .



Оцінка прискорення і ефективність розробленого алгоритму, спираючись на теорему Мунро-Петерсона:



де – кількість бінарних операцій;



– час розв’язання задачі на Р вузлах.



1. Операція множення двох матриць розмірністю потребує операцій множення і операцій додавання для формування одного елемента результуючої матриці. Отже, , . Але в завданні є два послідовних множення матриці, отже .



1. Операція множення матриці на число виконується за один такт .



1. Операція додавання матриць виконується за один такт .



Отже, сумарний час виконання операції:



Якщо , то



**2.2. Розробка алгоритмів процесів**

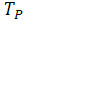
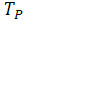
**Задача :**



1. Введення *.*



1. Чекати закінчення введення в .



1. Сигнал про закінчення вводу.



1. Копії



1. Обчислення



1. Чекати закінчення обчислення в .



1. Виведення *A*.

**Задача :**



1. Введення *.*



1. Сигнал про закінчення вводу.



1. Чекати закінчення введення в .



1. Копії



1. Обчислення



1. Сигнал про закінчення обчислення .



**Задача :**



1. Чекати закінчення введення в .



1. Копії



1. Обчислення



1. Сигнал про закінчення обчислення .



?

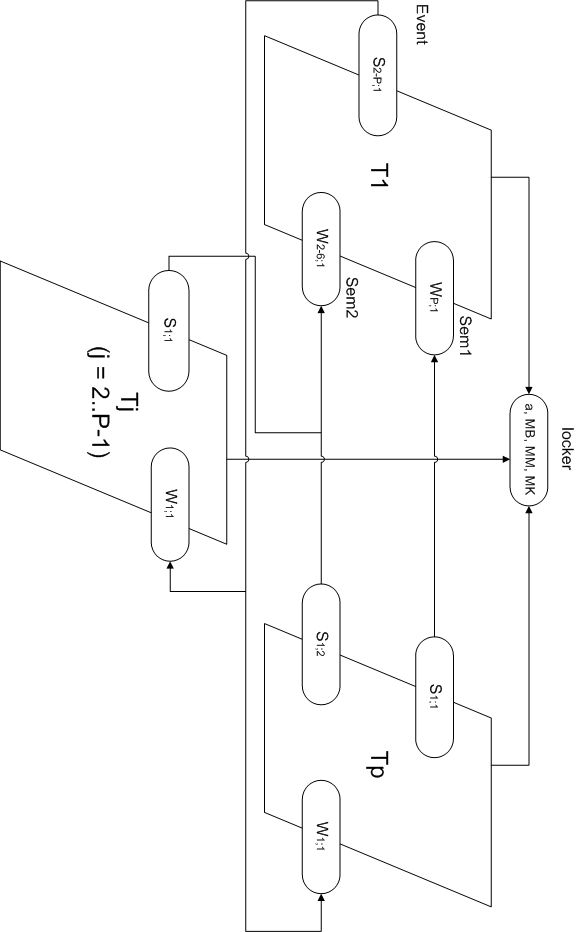


Рис. 2.2 Схема взаємодії процесів ПОДПИСЬ – ПОД РИСУНКОМ!!!!!

**2.3. Розробка схеми взаємодії процесів**

На основі алгоритму процесів розроблена структурна схема взаємодії процесів, що реалізовані за допомогою механізмів семафорів, моніторів та подій у мові програмування (рис. 2.2).

Об’єкт *locker* призначений для вирішення задачі взаємного виключення відносно ресурсів , за допомогою механізму *lock,* який є скороченою формою використання монітору у мові програмування



Семафор *Sem1* призначений для повідомлення потоком першого потоку про закінчення вводу.



Подія *Event* призначена для повідомлення всіх потоків першим потоком, що закінчилося введення даних.

Семафор *Sem2* призначений для повідомлення всіма потоками першого потоку про завершення обчислень.

**2.4. Розробка програми ПРГ1**

Програма ПРГ1 згідно технічного завдання розроблена на мові програмування *C#*. Для взаємодії процесів використовуються концепції моніторів, семафорів та подій. Програмний код наведений у додатку А.

Програма складається з єдиного класу *Program*, яка складається з:

* Визначення констант *N, P, H*.
* Об'явлення даних та їх типів, які потрібні для вирішення задачі.
* Об’явлення події *Event*, семафорів *Sem1* та *Sem2* та об’єкту *locker.*
* Методів *inputMatrix, copyMatrix, outputMatrix* призначених для організації вводу/виводу та копіювання матриць.
* Методу *Func*, який виконує кожний потік.
* Головного методу *Main*, який займається створенням та запуском потоків, присвоєнням їм унікально ідентифікатора та замірами часу.

Алгоритм роботи задач наведено у додатку B.

**2.5. Тестування програми ПРГ 1**

Метою проведення тестування є оцінка коефіцієнтів прискорення і коефіцієнтів ефективності для розроблених програм при їх виконанні на реальній паралельній обчислювальній системі. Для визначення вищевказаних коефіцієнтів проведений ряд експериментів із різними розмінностями операндів (N = 1000, 1500, 2000) і різною кількістю працюючих ядер ( P = 1, 2, 3, 4).

Для виміру часу використовується стандартний клас Stopwatch мови програмування, що за допомогою методів *Start()* та *Stop()* запускає та зупиняє таймер. Після чого за допомогою методу *ElapsedMilliseconds()* повертає замірений час у мілісекундах.

Для оцінки ефективності програми використовуються коефіцієнти прискорення та ефективності.

Коефіцієнт прискорення  показує скорочення часу виконання паралельної програми в паралельній системі з ** процесорами ** в порівнянні з часом виконання послідовної програми в однопроцесорній системі :



Коефіцієнт ефективності  застосування комп’ютерної системи показує ступінь використання ** процесорів системи:



Результати тестування і проведених досліджень ефективності розробленої програми наведено в табл. 2.1-2.3.

Таблиця 2.1. Час виконання програми ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 1000 | 53710 | 27670 | 19046 | 15101 |
| 1500 | 205374 | 103291 | 73611 | 54383 |
| 2000 | 493477 | 254831 | 172544 | 139348 |

На основі даних із табл. 2.1 виконано розрахунок значень коефіцієнтів прискорення, які наведені в табл. 2.2.

Використовуючи табл. 2.1‑2.3 побудовано графіки зміни часу виконання обчислень та коефіцієнтів прискорення і ефективності в залежності від *N* і  (рис. 2.4 – 2.6).

Таблиця 2.2. Коефіцієнти прискорення для програми ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 1000 | 1 | 1,94 | 2.82 | 3.56 |
| 1500 | 1 | 1,99 | 2.79 | 3.68 |
| 2000 | 1 | 1,94 | 2.86 | 3.54 |

Коефіцієнти ефективності табл. 2.3 обчислено за даними табл. 2.2.

Таблиця 2.3. Коефіцієнти ефективності для програми ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | P | | | |
| 1 | 2 | 3 | 4 |
| 1000 | 100 | 97.06 | 94.13 | 88.92 |
| 1500 | 100 | 99.42 | 93.21 | 92.41 |
| 2000 | 100 | 96.82 | 89.67 | 88.53 |

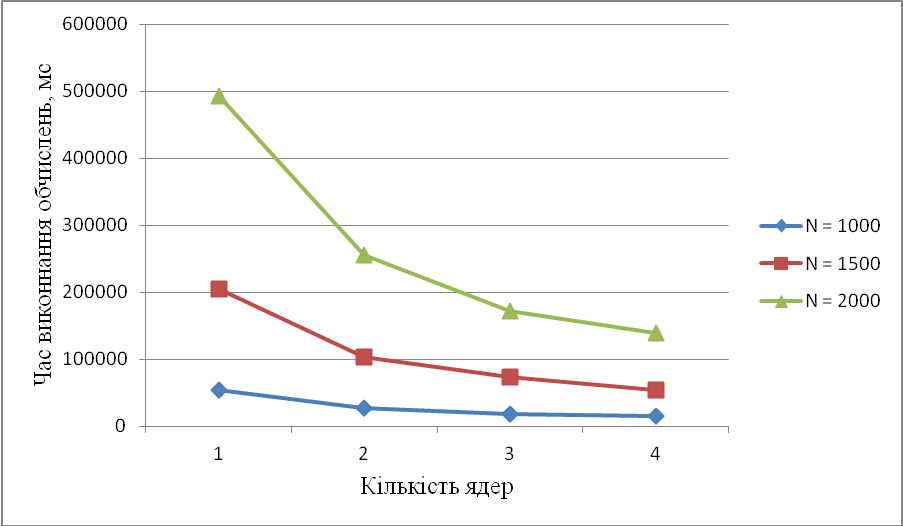


Рис. 2.4. Графік зміни часу виконання обчислень програми ПРГ1 в залежності від кількості ядер

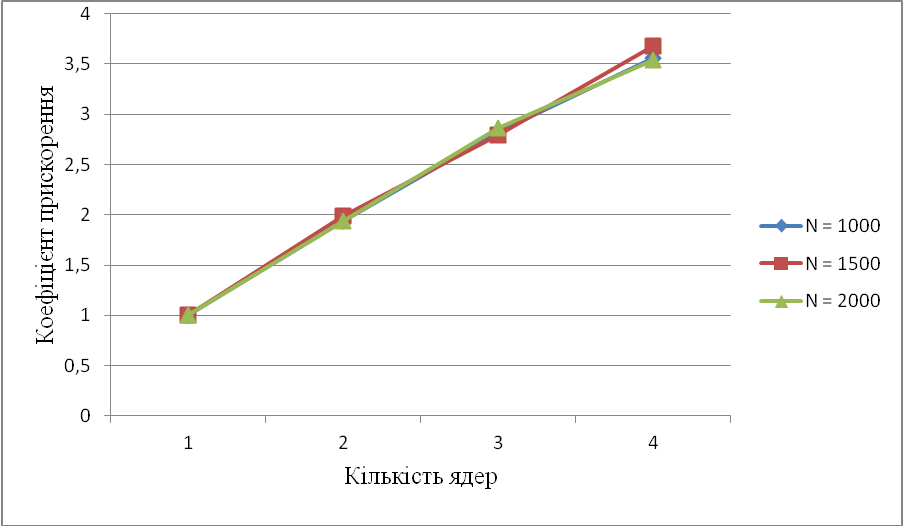


Рис. 2.5. Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер

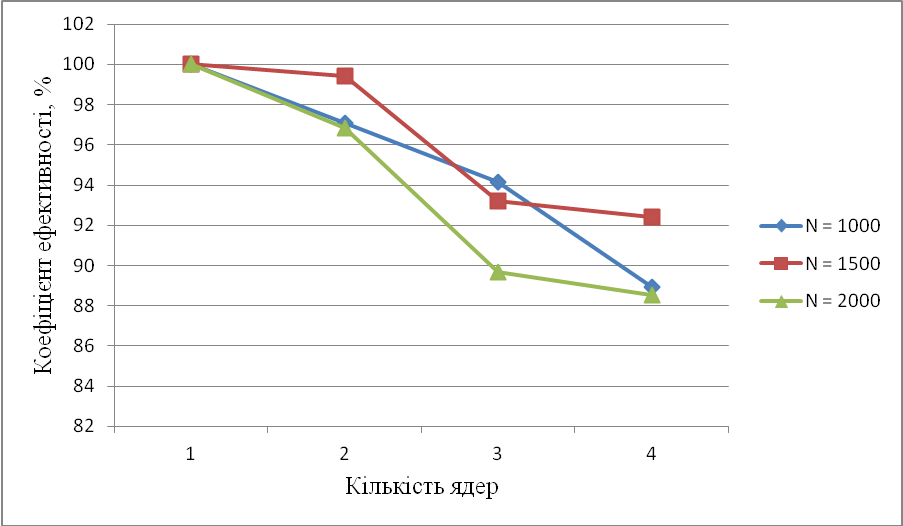


Рис. 2.6. Графік зміни коефіцієнту ефективності програми ПРГ1 в залежності від кількості ядер

**2.6. Висновки до розділу 2**

Виконано розробку програми ПРГ1 для ПКС із СП. При розробці використовувалась мова програмування *C#.* Для взаємодії процесів використовувались механізми моніторів, семафорів та подій.

На основі даних тестування програми ПРГ1 заповнено таблиці 2.1-2.3 та побудовано відповідні графіки на яких показано наступні результати:

* Використання багатоядерної ПКС забезпечує скорочення часу обчислення заданої математичної задачі.
* Максимальне значення  = 3.68 забезпечує ПКС з Р = 4 та N=1500.
* Мінімальне значення  = 1,94 забезпечує ПКС з Р = 2 та N = 2000.
* По графікам видно, що коефіцієнт прискорення збільшується, але пропорція його збільшення падає із збільшенням використовуваних ядер.
* Коефіцієнти ефективності коливаються в межах від 88-100%, отже алгоритм є ефективним.
* Максимальне значення = 99.42 для ПКС з Р = 2 та N = 1500;
* Мінімальне значення = 88.53 для ПКС з Р = 4 та N = 2000.

Оцінивши значення  та  можна зробити висновок про доцільність використання ресурсів ПКС для розв’язання задач такого типу.

**РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП**

В розділі розглядається розробка програми ПРГ2 для ПКС ЛП

Математична задача:



Мова програмування: C++ (MPI)

Засоби взаємодії процесів: механізм посилання повідомлень

Структура ПКС ЛП представлена на рис 3.1:



Рис. 3.1 Структура ПКС ЛП

**3.1. Розробка паралельного математичного алгоритму**

Згідно з технічним завданням необхідно розробити паралельний алгоритм. Він складається із одного етапу:



Пояснення до використовуваних констант:

*  – розмірність векторів і матриць;
*  – кількість ядер;
* .



**3.2. Розробка алгоритмів процесів**

**Задача:**

1. Прийняти з



1. Передати в



1. Прийняти з



1. Передати в



1. Передати в



1. Передати в



1. Обчислення



1. Прийняти з



1. Прийняти з



1. Прийняти з



1. Передати в



**Задача :**



1. Прийняти з



1. Передати в



1. Прийняти з



1. Передати в



1. Передати в



1. Передати в



1. Обчислення



1. Прийняти з



1. Прийняти з



1. Прийняти з



1. Передати в



**Задача :**



1. Введення



1. Передати в



1. Прийняти з



1. Передати в



1. Передати в



1. Обчислення



1. Прийняти з



1. Прийняти з



1. Прийняти з



1. Вивід



**Задача :**



1. Введення



1. Передати в



1. Прийняти з



1. Передати в



1. Передати в



1. Обчислення



1. Прийняти з



1. Прийняти з



1. Передати в



**Задача :**



1. Прийняти з



1. Обчислення



1. Передати в



**3.3. Розробка схеми взаємодії процесів**

На основі алгоритму процесів розроблена структурна схема взаємодії процесів. Вона реалізована за допомогою механізму відправки повідомлень у бібліотеці (рис. 3.2).



Всі процеси можна робити на п’ять груп:

* Процес у якому виконується ввід. Один в групі, оскільки по ліву сторону від цього процесу немає інших процесів



* Процес у якому виконується ввід. Єдиний у якого по праву сторону немає інших процесів



* - ліва група процесів. Якщо є процес, який знаходиться по середині, то він відносить до цієї групи. Спочатку оброблює ресурс введений в , а потім із .



* - права група процесів. Спочатку оброблює ресурс введений в , а потім із .



* - другорядні процеси, оскільки доступ до них має лише один інший процес, вони собою представляють тупик. Лише приймають ресурс, роблять обчислення та повертають результат.



Без поділу процесів на ліву та праву групу розповсюдження було б лінійним, тобто на кожному кроці задіються лише два процеси для взаємодії відправки повідомлення і спочатку ресурс введений в буде доставлено до і тільки потім ресурс введений в почне розповсюджуватися або ж події будуть відбуватися навпаки.



?



Рис. 3.2 Схема взаємодії процесів

На схемі об’єкти та відповідають методам бібліотеки – та відповідно.



Позначки на схемі:



**3.4. Розробка програми ПРГ2**

Програма ПРГ2 згідно технічного завдання розроблена за допомогою бібліотеки . Для взаємодії процесів використовується концепція передачі повідомлень. Програмний код наведений у додатку С.



Програма складається з єдиного класу *Program*, яка складається з:

* Визначення констант *N, P, H*.
* Методів *inputMatrix, copyMatrix, outputMatrix* призначених для організації вводу/виводу матриць.
* Методу *calc*, який обчислює функцію.
* Головного методу *Main*, у якому написаний виконуваний код потоків.

Алгоритм роботи задач наведено у додатку E.

**3.5. Тестування програми ПРГ 2**

Метою проведення тестування є оцінка коефіцієнтів прискорення і коефіцієнтів ефективності для розроблених програм при їх виконанні на реальній паралельній обчислювальній системі. Для визначення вищевказаних коефіцієнтів проведений ряд експериментів із різними розмінностями операндів (N = 1000, 1500, 2000) і різною кількістю працюючих ядер ( P = 1, 2, 3, 4).

Для виміру часу використовується стандартний метод класу Java.lang мови програмування Java – System.currentTimeMilliis(), який повертає час, який пройшов із 1970 року в мілісекундах. Взявши різницю двох викликів – на початку і вкінці – отримаємо час роботи програми.

Для оцінки ефективності програми використовуються коефіцієнти прискорення та ефективності.

Коефіцієнт прискорення  показує скорочення часу виконання паралельної програми в паралельній системі з ** процесорами ** в порівнянні з часом виконання послідовної програми в однопроцесорній системі : 

Коефіцієнт ефективності  застосування комп’ютерної системи показує ступінь використання ** процесорів системи:



Таблиця 3.1. Час виконання програми ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 1000 |  |  |  |  |
| 1500 |  |  |  |  |
| 2000 |  |  |  |  |

Результати тестування і проведених досліджень ефективності розробленої програми наведено в табл. 3.1-3.3.

На основі даних із табл. 3.1 виконано розрахунок значень коефіцієнтів прискорення, які наведені в табл. 3.2.

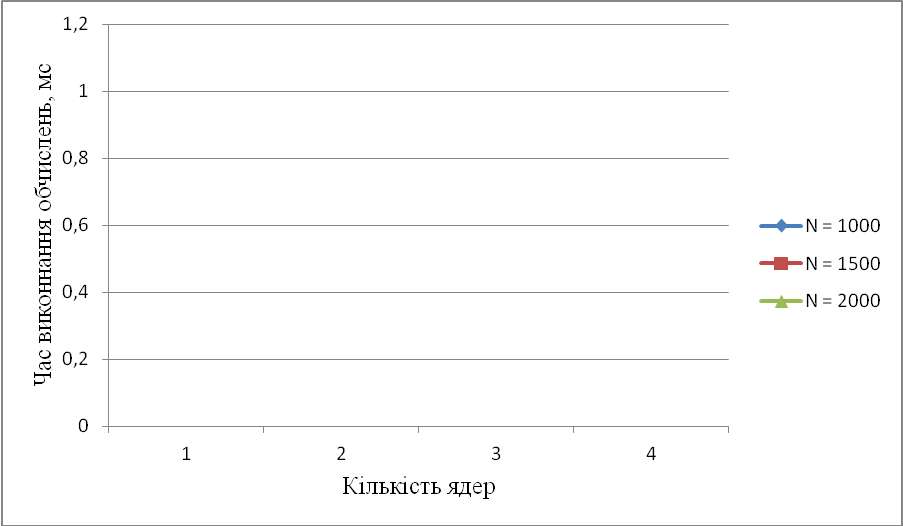


Рис. 3.4. Графік зміни часу виконання обчислень програми ПРГ2 в залежності від кількості ядер

Таблиця 3.2. Коефіцієнти прискорення для програми ПРГ2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 1000 |  |  |  |  |
| 1500 |  |  |  |  |
| 2000 |  |  |  |  |

Таблиця 3.3. Коефіцієнти ефективності для програми ПРГ2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | P | | | |
| 1 | 2 | 3 | 4 |
| 1000 |  |  |  |  |
| 1500 |  |  |  |  |
| 2000 |  |  |  |  |

Використовуючи табл. 3.1‑3.3 побудовано графіки зміни часу виконання обчислень та коефіцієнтів прискорення і ефективності в залежності від *N* і  (рис. 3.4 – 3.6).

Коефіцієнти ефективності табл. 3.3 обчислено за даними табл. 3.2.

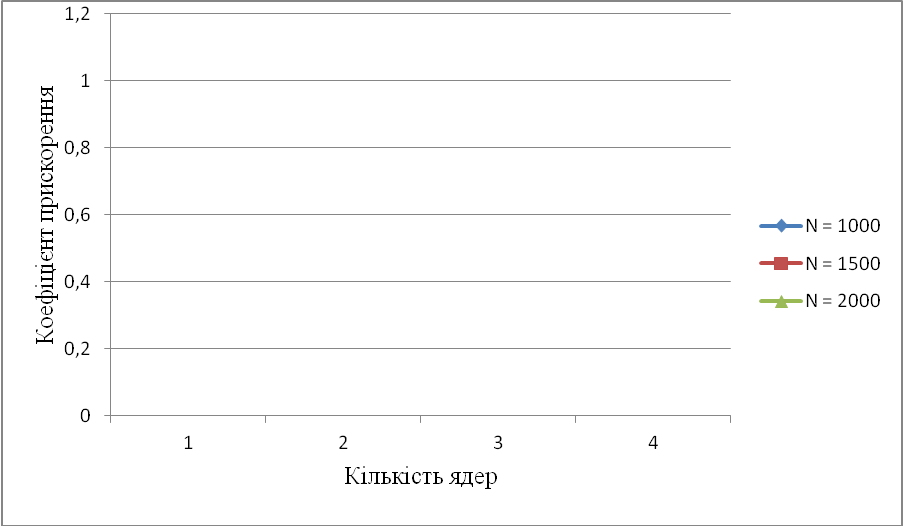


Рис. 3.5. Графік зміни коефіцієнту прискорення програми ПРГ2 в залежності від кількості ядер

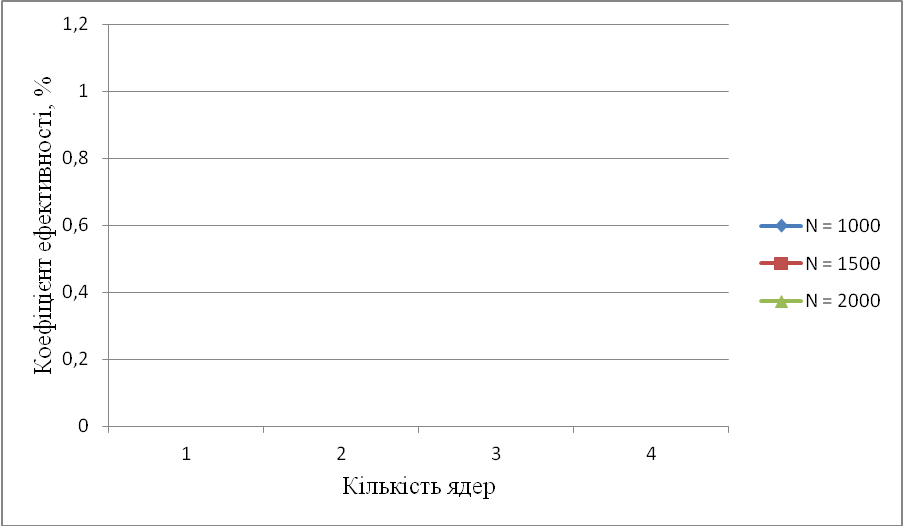


Рис. 3.6. Графік зміни коефіцієнту ефективності програми ПРГ2 в залежності від кількості ядер

**3.6. Висновки до Розділу 3**

Виконано розробку програми ПРГ2 для ПКС із ЛП. При розробці використовувалась мова програмування *Java* – бібліотека *MPI.* Для взаємодії процесів використовувались механізми посилання повідомлень.

На основі даних тестування програми ПРГ2 заповнено таблиці 3.1-3.3 та побудовано відповідні графіки на яких показано наступні результати:

* Використання багатоядерної ПКС забезпечує скорочення часу обчислення заданої математичної задачі.
* Максимальне значення  = 0.00 забезпечує ПКС з Р = 0 та N=0000.
* Мінімальне значення  = 0.00 забезпечує ПКС з Р = 0 та N = 0000.
* Коефіцієнти ефективності коливаються в межах від 00-000%
* Максимальне значення = 00.00 для ПКС з Р = 0 та N = 0000;
* Мінімальне значення = 00.00 для ПКС з Р = 0 та N = 0000.

Оцінивши значення  та  можна зробити висновок про доцільність використання ресурсів ПКС для розв’язання задач такого типу.

ЗАГАЛЬНІ ВИСНОВКИ,,,,,

1

2

3

4

5

6

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Жуков І. А., Корочкін О. В. Паралельні та розподілені обчислення. Навч. посіб. – 2-ге вид. виправл. і доп. К.: «Корнійчук», 2014. – 284с.
2. Шилдт, Герберт. Java. Полное руководство, 8-е изд.: Пер. с англ. — М.: ООО “И.Д. Вильямс”,2012. — 1104 с .
3. А.Е. Гавва - Адское программирование Май 2004. - 433 с.

МАЛО!!!!! СТАНДАРТ НЕ ВЫДЕРЖАН!!!!!

**ДОДАТОК А**

/\*\*

\* Author : Repeta Yaroslav

\* Group : IO – 24

\* Faculty : FICT

\* Date : 19.04.2015

\*/

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

using System.Diagnostics;

namespace CourseWork1

{

class Program

{

const int N = 2000;

const int P = 2;

const int H = N / P;

static int a;

static int[,] MA = new int[N, N];

static int[,] MB = new int[N, N];

static int[,] MC = new int[N, N];

static int[,] MM = new int[N, N];

static int[,] MK = new int[N, N];

static int[,] MT = new int[N, N];

static EventWaitHandle E1 = new ManualResetEvent(false);

static Semaphore S1 = new Semaphore(0, 1);

static Semaphore S2 = new Semaphore(0, P - 1);

static object locker = new object();

/\*

\* Метод для вводу матриці

\*\*/

static int[,] inputMatrix(int[,] XX)

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

XX[i, j] = 1;

}

}

return XX;

}

/\*

\* Метод для копіювання матриці

\*\*/

static int[,] copyMatrix(int[,] XX, int[,] YY)

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

YY[i, j] = XX[i, j];

}

}

return YY;

}

/\*

\* Метод для виводу матриці

\*\*/

static void outputMatrix(int[,] XX)

{

if (N < 20)

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

Console.Write(XX[i, j] + " ");

}

Console.WriteLine();

}

}

/\*

\* Виконавчий метод потоків

\*\*/

static void Func()

{

int thread = Convert.ToInt32(Thread.CurrentThread.Name);

Console.WriteLine("T" + (thread + 1) + " started.");

switch (thread)

{

case 0:

inputMatrix(MB);

inputMatrix(MT);

break;

case P-1:

inputMatrix(MC);

inputMatrix(MM);

inputMatrix(MK);

a = 1;

break;

default:

break;

}

switch (thread)

{

case 0:

S1.WaitOne();

break;

case P-1:

S1.Release();

break;

default:

break;

}

if (thread == 0)

E1.Set();

else

E1.WaitOne();

int ai;

int[,] MBi = new int[N, N];

int[,] MMi = new int[N, N];

int[,] MKi = new int[N, N];

lock (locker)

{

ai = a;

copyMatrix(MB, MBi);

copyMatrix(MM, MMi);

copyMatrix(MK, MKi);

}

int buff;

for (int i = 0; i < N; i++)

{

for (int j = thread \* H; j < (thread + 1) \* H; j++)

{

buff = 0;

for (int k = 0; k < N; k++)

{

MA[i, j] += MBi[i, k] \* MC[k, j];

buff += MKi[i, k] \* MT[k, j];

}

for (int k = 0; k < N; k++)

{

MA[k, j] += (buff \* MMi[k, j] \* ai);

}

}

}

if (thread == 0)

for (int i = 0; i < P - 1; i++)

S2.WaitOne();

else

S2.Release();

if (thread == 0)

outputMatrix(MA);

Console.WriteLine("T" + (thread + 1) + " finished.");

}

/\*

\* Голвний метод для створення потоків

\*\*/

static void Main(string[] args)

{

Console.Read();

Console.WriteLine("Main thread started.");

Stopwatch timer = new Stopwatch();

timer.Start();

Thread[] threads = new Thread[P];

for (int i = 0; i < P; i++)

{

threads[i] = new Thread(Func);

threads[i].Name = "" + i;

threads[i].Start();

}

threads[0].Join();

timer.Stop();

long time = timer.ElapsedMilliseconds;

Console.WriteLine("Time of work: " + time);

Console.WriteLine("Main thread finished");

}

}

}

**СХЕМЫ СДЕЛАНЫ С НАРУШЕНИЕМ СТАНДРТА!!!!**

