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-34

напряму підготовки 050102 «Комп’ютерна інженерія»

\_\_Кривоносова О. О.

(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- 2016 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

\_Кривоносов Олексій Олексійович\_

*(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 17 травня 2016 р.

3. Вхідні дані до роботи

* Порівняння засобів роботи із процесами в мовах програмування Ada і Java
* математична задача: A = (B\*MO)\*α + C\*(MT\*MR)
* структури паралельної комп’ютерної системи з спільною пам’яттю (ПКС СП) та паралельної комп’ютерної системи з локальною пам’яттю (ПКС ЛП);
* мова програмування: Ada

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

- порівняння засобів роботи із процесами в мовах програмування Ada і Java

- розробка і тестування програми ПРГ1 для ПКС ОП



Рис 1.1. Структура ПКС СП

- розробка і тестування програми ПРГ2 для ПКС ЛП



Рис 1.2. Структура 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.

6. Дата видачі завдання 16.02.2016

***КАЛЕНДАРНИЙ ПЛАН***

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строк виконання етапів КР |
| 1 | Виконання Розділу 1 | 29.02.2016 |
| 2 | Виконання Розділу 2 | 24.03.2016 |
| 3 | Виконання Розділу 3 | 23.04.2016 |
| 4 | Тестування програм ПРГ1 та ПРГ2 | 10.05.2016 |
| 7 | Оформлення КР | 16.05.2016 |
| 8 | Захист КР | 17.05.2016 |

**Студент \_\_\_\_\_\_\_\_\_** Кривоносов О.О.

(підпис) (прізвище та ініціали)

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

(підпис) (прізвище та ініціали)

# **ЗМІСТ**

[ЗМІСТ 5](#_Toc451102173)

[РОЗДІЛ 1. ПОРІВНЯННЯ ЗАСОБІВ РОБОТИ ІЗ ПРОЦЕСАМИ В МОВАХ АДА І JAVA 7](#_Toc451102174)

[1.1. Потоки і процеси 7](#_Toc451102175)

[1.1.1. Загальне визначення потоків і процесів 7](#_Toc451102176)

[1.1.2. Порівняння потоків і процесів 8](#_Toc451102177)

[1.1.3. Відмінності між потоками і процесами 8](#_Toc451102178)

[1.1.4. Процеси в мові Ада 8](#_Toc451102179)

[1.1.5. Процеси в мові Java 9](#_Toc451102180)

[1.2. Методи роботи з процесами 12](#_Toc451102181)

[1.2.1. Пріоритет процесу 12](#_Toc451102182)

[1.2.2. Блокування виконання процесу 13](#_Toc451102183)

[1.2.3. Примусове завершення процесу 14](#_Toc451102184)

[1.3. Змінні для синхронізації 14](#_Toc451102185)

[1.3.1. Змінні, що розділяються в Ада 14](#_Toc451102186)

[1.3.2. Змінні, що розділяються в Java 15](#_Toc451102187)

[1.4. Семафори 16](#_Toc451102188)

[1.4.1. Семафори в Ада 16](#_Toc451102189)

[1.4.2. Семафори в мові Java 18](#_Toc451102190)

[1.5. Монітори 19](#_Toc451102191)

[1.5.1. Монітори в Ада 19](#_Toc451102192)

[1.5.2. Монітори в Java 19](#_Toc451102193)

[1.6. Висновки до розділу 1 20](#_Toc451102194)

[РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС СП 22](#_Toc451102195)

[2.1 Розробка паралельного математичного алгоритму 22](#_Toc451102196)

[2.2 Розробка алгоритмів роботи процесів 23](#_Toc451102197)

[2.3 Розробка структурної схеми взаємодії задач 24](#_Toc451102198)

[2.4 Розробка програми 25](#_Toc451102199)

[2.5 Тестування програми ПРГ1 25](#_Toc451102200)

[2.6 Висновки до розділу 2 28](#_Toc451102201)

[РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП 29](#_Toc451102202)

[3.1 Розробка паралельного математичного алгоритму 29](#_Toc451102203)

[3.2 Розробка алгоритмів процесів 30](#_Toc451102204)

[3.3 Розробка схеми взаємодії процесів 31](#_Toc451102205)

[3.4 Розробка програми ПРГ2 32](#_Toc451102206)

[3.5 Тестування програми ПРГ2 33](#_Toc451102207)

[3.6 Висновки до розділу 3 36](#_Toc451102208)

[ОСНОВНІ РЕЗУЛЬТАТИ І ВИСНОВКИ ПО РОБОТІ 38](#_Toc451102209)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 39](#_Toc451102210)

[ДОДАТКИ 40](#_Toc451102211)

[Додаток А. Структурна схема ПКС СП 41](#_Toc451102212)

[Додаток Б. Схема алгоритму головної програми зі вказанням паралельних ділянок для ПРГ1 43](#_Toc451102214)

[Додаток В. Схема алгоритму процесів для програми ПРГ1 45](#_Toc451102216)

[Додаток Г. Лістинг програми ПРГ1 для ПКС СП 47](#_Toc451102218)

[Додаток Д. Структурна схема ПКС ЛП 51](#_Toc451102219)

[Додаток Е. Схема алгоритму головної програми зі вказанням паралельних ділянок для ПРГ2 53](#_Toc451102220)

[Додаток Ж. Лістинг програми ПРГ2 для ПКС СП 55](#_Toc451102221)

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

## 1.1. Потоки і процеси

### 1.1.1. Загальне визначення потоків і процесів

Роботу будь-якої послідовної програми можна розділити між декількома підпрограмами. Кожній підпрограмі призначається конкретна задача, і всі ці задачі виконуються послідовно, тобто наступна задача не може початися, поки не закінчиться поточна. Ця схема буде доцільно працювати доти, поки не буде досягнуто межі продуктивності і складності. Одним з методів вирішення продуктивності і складності є виконування цих задач паралельно, тобто поділити основну програму на потоки. Потоки дозволяють одній програмі складатися за паралельно виконуваних частин, які мають доступ до одних і тих змінних, констант та адресного простору в цілому.

Не слід ототожнювати потоки і процеси (нижче буде більш детально розглянуто порівняння потоків і процесів), однак, потоки можна розглядати як полегшені процеси, тому що вони дозволяють використати переваги процесів.

Процес – абстрактне поняття, що включає опис певних дій, пов’язаних з використанням програми в комп’ютерній системі. При цьому процес оформлюється таким чином, щоб система керування процесами в операційні системі могла ефективно перерозподіляти ресурси системи (процесори, пам’ять, прилади введення-виведення, файли та інше). Процес характеризується власним набором ресурсів і виділеною для нього ділянкою оперативної пам’яті[2].

Потоки – це виконуванні частини програми, що виборюють ресурси комп’ютерної системи – процесор, прилади введення-виводу та інше. У багатопроцесорній системі потоки можуть виконуватись на різних процесорах одночасно, але в межах одного процесу. Наприклад, процесу Process1, що має три потоки, призначено процесори 1, 2, 3, незважаючи на те, що в системі є ще процесори, що простоюють, потоки процесу Process1, будуть виконуватись тільки на процесорах 1, 2, 3 [7].

### 1.1.2. Порівняння потоків і процесів

Потоки і процеси мають багато спільних рис. Наприклад, ідентифікаційний номер (id), стан, набір регістрів та пріоритет. Як і процеси, потоки мають свої атрибути, що характеризують їх для операційної системи. Також, потік чи процес може змінити свої атрибути та створити свої ресурси, але не має доступу до ресурсів, що належать іншим процесам.

### 1.1.3. Відмінності між потоками і процесами

Основна відмінність потоків від процесів полягає в тому, що потоки не мають власного адресного простору, а процеси мають. Наприклад, якщо процес створює багато потоків, то всі вони знаходитимуться в адресному просторі цього процесу. Це використовується для легкого розділу загальних ресурсів та для забезпечення простої і швидкої взаємодії між потоками.

### 1.1.4. Процеси в мові Ада

В мові програмування Ада процеси створюються за допомогою задач [4]. 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** Example1 **is**

**task**  Task1;

**task**  Task2;

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

**begin**

Ada.Text\_IO.Put\_Line(“Task1”);

**end** Task1;

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

**begin**

Ada.Text\_IO.Put\_Line(“Task2”);

**end** Task2;

**begin**

**null**;

**end**  Example1;

У даній програмі створюється дві задачі, 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.

### 1.1.5. Процеси в мові Java

В мові програмування Java процес можна створити за допомогою потоків [5]. 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*. 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 визначає інструкції для створюваного потоку.

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.*

Створений потоковий об’єкт не запускається сам, як це реалізовано в мові Ада. Його необхідно викликати за допомогою методу  *start(),* який, знайшовши в об’єкті визначений метод *run()* та передає йому керування.

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();

}

}

## 1.2. Методи роботи з процесами

### 1.2.1. 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 (1);

. . .

**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 методи для роботи із пріоритетом. Нижче вони наведені (для встановлення та отримання значення пріоритету відповідно):

**final void** setPriority *(****int*** *priority)*

**final int** getPriority ()

Пріоритети в Java можуть мати значення від одного до десяти.

### 1.2.2. Блокування 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[1]:

• відносна затримка виконання

• абсолютна затримка виконання

Загальний вид інструкції відносної затримки виконання наступний:

**delay**  *delay\_time*;

Де *delay\_time* 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**  *time\_moment*;

Де *time\_moment* визначає точний момент часу до якого потрібно зробити призупинення виконання задачі. Наприклад:

**delay until**  Time\_Of (2016, 3, 1, 0.0);

Це означає, що процес буде заблокований до 1 березня 2016 року.

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

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

**static void** sleep (*long millisecond, long nanoseconds*) **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.

### 1.2.3. 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.

## 1.3. Змінні для синхронізації

### 1.3.1. Змінні, що розділяються в Ада

У мові Ада для вказівки змінних, що поділяються використовують наступні директиви компілятора[8]:

**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

### 1.3.2. Змінні, що розділяються в Java

У мові програмування Java є аналогічні засоби для доступу до змінних. [10] 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);

## 1.4. Семафори

### 1.4.1. Семафори в Ада

В мові програмування Ada механізм семафорів реалізується за допомогою спеціального пакету *Ada.Synchronous\_Task\_Control.* [8] Цей пакет містить набор процедур та тип, з яким вони працюють. Специфікація цього пакету виглядає так:

**package** Ada.Synchronous\_Task\_Control **is**

**type** Suspension\_Object **is** **limited** **private**;  
   **procedure** Set\_True(*S:****in******out*** *Suspension\_Object*);  
   **procedure** Set\_False(*S :****in******out****Suspension\_Object*);  
  **function** Current\_State(*S : Suspension\_Object*) **return** Boolean;  
  **procedure** Suspend\_Until\_True(*S :****in******out****Suspension\_Object*);  
**private**  
   ...  
**end** Ada.Synchronous\_Task\_Control;

*Suspension\_Object* – це тип об’єктів, що власне становлять з себе семафор. Процедури, що входять до складу пакету, використовують ці об’єкти для реалізації механізму семафорів. Цей семафор може приймати одне з двох значень – true або false, тобто значення типу Boolean. Початковим значенням семафору при його створенні є false. З точки зору програмного середовища Suspension\_Object становить з себе приватний обмежений тип.

*Set\_True* – це процедура, що встановлює семафор в значення true. Ця процедура, як і всі інші, є атомарною, тобто одночасне її виконання декількома процесами не допускається. Це дозволяє уникнути помилок, пов’язаних з використанням семафору як спільного ресурсу. Ця процедура є реалізацією операції V(S)

*Set\_False* – це функція, що встановлює семафор в значення false. Вона зазвичай використовується лише для встановлення початкового значення

семафору, та, на відміну від попередньої функції, безпосередньої участі в реалізації механізму семафору не бере.

*Suspend\_Until\_True* – це наступна процедура, що є реалізацією операції P(S). При її виклику певним процесом вона перевіряє, чи становить значення семафору *true*. Якщо так, то виконання процесу продовжиться без затримок. В іншому випадку процес блокується доти, доки значення семафору не дорівнюватиме true. Після виконання цієї перевірки цей метод встановить значення семафору в false, таким чином обмежуючи доступ до критичної ділянки іншим процесам.

*Current\_State* – це остання функція з пакету *Ada.Synchronous\_Task\_Control,* яка слугує для повертання поточного значення семафору. Вона також є допоміжною та не використовується безпосередньо для організації взаємодії процесів.

З цієї специфікації можна зробити висновок, що семафори в *Ada* становлять з себе прості семафори, в яких в якості типових значень 0 та 1 використовуються булеві *true* та *false*. Це означає, що при використанні семафору для вирішення задачі синхронізації неможливо використати один семафор для очікування сигналу від декількох процесів. Це відноситься до недоліків семафорів в цій мові програмування. До переваг цієї реалізації механізму семафорів віднесемо той факт, що використання значення типу *Boolean* є більш економічним з точки зору витраченої пам’яті та швидкості роботи семафору у порівнянні зі звичайними числовими значеннями що може бути критичним при використанні великої кількості семафорів в системі з малим об’ємом оперативної пам’яті. Ще однією перевагою цієї реалізації відзначимо наявність службових функцій, які дозволяють встановити початкове значення семафору та дізнатись його поточне значення, що полегшує роботу з семафором та розширює можливості, що від надає.

### 1.4.2. Семафори в мові Java

Реалізувати об’єкт семафора в Java доволі просто, але не дивлячись на це в JDK вже є пакет, що надає функціональність семафору - *java.util.concurrent.Semaphore*[6].

На відміну від мови програмування Ада, де для управління доступом до ресурсу семафор використовує бінарні логічні змінні, тобто *true* і *false* , у мові програмування Java використовується лічильник, який представляє кількість дозволів. Якщо значення лічильника більше нуля, то потік отримує доступ до ресурсу, при цьому лічильник зменшується на одиницю. Після закінчення роботи з ресурсом потік звільняє семафор, і лічильник збільшується на одиницю. Якщо ж лічильник дорівнює нулю, то потік блокується і чекає, поки не отримає дозвіл від семафора.

Встановити кількість дозволів для доступу до ресурсу можна за допомогою конструкторів класу Semaphore:

**Semaphore**(**int** permits);

**Semaphore**(**int** permits, **boolean** fair);

Параметр *permits* вказує на кількість допустимих дозволів для доступу до ресурсу. Параметр *fair* в другому конструкторі дозволяє встановити порядок отримання доступу. Якщо він дорівнює true, то дозволи надаватимуться потокам, що очікують, в тому порядку, в якому вони просили доступ. Якщо ж він дорівнює false, то дозволи надаватимуться в невизначеному порядку.

Для отримання дозволів у семафора необхідно викликати метод acquire(), що має дві форми:

**void** acquire() **throw** InterruptedException

**void** acquire(***int*** *permits*) **throw** InterruptedException

Для отримання одного дозволу застосовується перший варіант, а для отримання декількох дозволів - другий варіант. Після виклику цього методу поки потік не отримає дозвіл, він блокується. Після закінчення роботи з ресурсом отримане раніше дозвіл треба звільнити за допомогою методу *release ():*

**void** release();

**void** release(*int permits*);

Одним з суттєвих мінусів семафора є те, що він займає так званий процесорний час, а також можливість фактично заблокувати роботу процесу[3].

## 1.5. Монітори

### 1.5.1. Монітори в Ада

Монітори – це інкапсуляція визначеного ресурсу, де всі оператори, що керують ресурсом виконується за принципом взаємного виключення[9]. Монітори створені для вирішення деяких проблем, що виникають при роботі с семафорами.

З точки зору Ада, монітор може розглядатися як пакет, в специфікації якого визначені лише процедури, сам сирцевий код прихований в «корпусі» монітора і має доступ тільки до цих процедур, тобто модуль є захищеним. Однак, на відміну від пакета , монітор виконує контроль над своїми зовнішніми підпрограмами. Зокрема, гарантується взаємовиключне виконання процедур монітора. Це гарантує, що змінні, оголошені в тілі монітора ніколи не можуть бути предметом паралельного одночасного доступу.

### 1.5.2. 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)[10].

Монітор в Java можна реалізувати наступним чином:

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

**synchronized** double read() {

**return** resource;

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

resource = x;

}

}

## 1.6. Висновки до розділу 1

1. Було розглянуто процеси та потоки. Головною відмінністю потоків від процесів є те, що процеси мають власний адресний простір. Але, бувають випадки, коли необхідно мати в одному і тому ж адресному просторі декілька потоків керування, що виконуються квазіпаралельно, наче це, процеси(за винятком адресного простору) - для цього використовуються потоки.
2. Проведено аналіз реалізації процесів в мовах програмування Ada та Java. В мові програмування Ada керування процесами реалізовано за допомогою задач. Кожна задача має специфікацію та тіло. В мові програмування Java процеси реалізовані за допомогою потоків. Існує два способи створення потоків: за допомогою реалізації інтерфейсу *Runnable* та за допомогою розширення класу Thread. Було виявлено, що в Java для запуску потоку, необхідно викликати його метод *run(),* через виклик методу основного потоку *start()*, на відміну від Ada, де задача запускається разом з основною програмою.
3. Були проаналізовані методи роботи з процесами та виявлено, що в обох мовах програмування процес(потік чи задача, в залежності від мови, що розглядається) має власне ім’я та пріоритет, а також методи для встановлення пріоритету. Також, в цих мовах реалізоване блокування процесу на деякий час, а в Ada, навіть є можливість заблокувати процес до конкретної дати. Щодо примусового завершення процесів, то обидві мови надають приблизно однакову функціональність.
4. В ході аналізу можливостей мов програмування вирішення проблеми синхронізації було виявлено, що обидві мови мають атомарні змінні та змінні типу *volatile.* В обох мовах реалізовано семафори: в Ada семафор реалізовано бінарною логічною змінною, а в Java – це лічильник. Для реалізації моніторів в мові Ada присутні захищенні модулі, в Java не мається спеціальних засобів реалізації моніторів, для цього використовуються класи, що мають методи з модифікаторами *synchronized* та *private.*

# РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС СП

В розділі розглянуто розробку та дослідження програми ПРГ1 для ПКС зі спільною пам’яттю.

**Математична задача:**  A = (B\*MO)\*α + C\*(MT\*MR)

**Мова програмування:** Ada

**Засоби взаємодії процесів:** Захищений модуль

**Структура ПКС СП представлено на рис. 2.1**

Спільна пам’ять

•••

•

•

P

MО, MR, α

P-1

2

1

A, B, C, MT

*Рис. 2.1 Структура ПКС зі спільною пам’яттю*

## 2.1 Розробка паралельного математичного алгоритму

Розрахунок даного математичного рівняння можна представити в наступній формі:

AH = (B\*MOH)\*α + C\*(MT\*MRH)

*  – розмірність векторів і матриць;
*  – кількість ядер;
* ;
* AH – H елементів вектора А;
* МОН – Н рядків матриці МО;
* MRH – H рядків матриці MR;

*СР: α, В, C, МТ*

Кожна операція виконується за одиницю часу, а час, за який передаються дані між процесорами системи не враховується.

## 2.2 Розробка алгоритмів роботи процесів

Кроки ТC /КД № процесу

1. Введення B, C, MT tid=1
2. Копіювання Bm = B, Cm = C, MTm = MT КД tid=1
3. Введення МO, МR, α В tid=р
4. Копіювання αm = α. КД tid=p
5. Сигнал завершення вводу S1 tid=1,p
6. Очікування завершення вводу W1 tid=1..p
7. Копіювання Bi = Bm, Ci = Cm, КУ tid=1..р

MTi = MTm, αi = αm. (i= 1..p)

1. Обчислення tid=1..p

AH = (Bi\*MOH)\*αi + Ci\*(MTi\*MRH)

(i = 1.. p)

1. Сигнал про завершення обчислень. S2 tid=2..р
2. Очікування сигналу про завершення W2 tid=1

Обчислень від задач Ti(i=2..p)

1. Виведення результату A tid = 1

***Xm*** *- це змінна Х, що належить до захищеного модулю.*

## 2.3 Розробка структурної схеми взаємодії задач



*Рис. 0.2 Схема взаємодії задач для системи зі спільною пам’яттю*

Для вирішення проблем синхронізації та доступу до спільних ресурсів, вводиться захищений модуль Control. Його процедури, входи та функції:

***InputFinished*** – захищена процедура, сигнал про завершення введення, використовується першим і останнім (за номером) процесами, для оповіщення усіх інших потоків про кінець введення

***WaitForInput*** – захищений вхід, очікування сигналу про закінчення введення

***CalcFinished*** – захищена процедура, сигнал про закінчення обчислень. Використовується усіма потомками (окрім першого)

***WaitForFinishCalc*** – очікування завершення розрахунків усіма процесами (використовується першим потоком, як сигнал до виведення результату обчислень)

***Get\_В*** – функція отримання локальної копії вектору В.

***Get\_С*** – функція отримання локальної копії вектору С.

***Get\_MT*** – функція отримання локальної копії матриці МТ.

***SetB\_C\_MT*** – захищена процедура, для запису у модуль B, C, MT.

***Get\_alfa*** – функція отримання локальної копії константи α.

***Set\_alfa*** – захищена процедура, для запису у модуль константи α

## 2.4 Розробка програми

Для реалізації алгоритму обчислення матричного рівняння на обчислювальній системі зі спільною пам’яттю було використано мову Ada. Ця мова має потужний апарат для забезпечення паралельної обробки.

Для вирішення проблем синхронізації та доступу до спільного ресурсу було створено захищений модуль, що містить захищені входи для синхронізації по вводу та виводу. А також захищені процедури і функції для вирішення проблем взаємного виключення.

Лістинг програми наведено у Додатку Г.

## 2.5 Тестування програми ПРГ1

Для тестування використовувалась паралельна обчислювальна система з наступними апаратними характеристиками: процесор – **Intel Core i5-4300U** 4\*1.9ГГц; оперативна пам’ять - 2x4096 Гб; OC: Microsoft Windows 8.1 x64. В якості середовища розробки і компіляції Ada програми ObjectAda V8.4

Для виміру часу використовується стандартна процедура Clock мови програмування Ада, що повертає поточний час системи. Різниця між часом системи після завершення обчислення і часом при запуску програми дає час виконання обчислення, що вимірюється в секундах.

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 |
| 1200 | 30 | 15 | 12 | 8 |
| 1800 | 59 | 31 | 24 | 16 |
| 2400 | 102 | 52 | 40 | 28 |
| 3000 | 168 | 89 | 68 | 49 |

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

Таблиця 2.2. Коефіцієнти прискорення для програми ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 1200 | 1 | 2 | 2,5 | 3,75 |
| 1800 | 1 | 1,91 | 2,458333 | 3,6875 |
| 2400 | 1 | 1,961538 | 2,55 | 3,642857 |
| 3000 | 1 | 1,88764 | 2,470588 | 3,428571 |

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 |
| 1200 | 100 | 100 | 83,33333 | 93,75 |
| 1800 | 100 | 95,16129 | 81,94444 | 92,1875 |
| 2400 | 100 | 98,07692 | 85 | 91,07143 |
| 3000 | 100 | 94,38202 | 82,35294 | 85,71429 |

Використовуючи табл. 2.1‑2.3 побудовано графіки зміни часу виконання обчислень та коефіцієнтів прискорення і ефективності в залежності від *N* і  (рис. 2.3 – 2.5).

*Рис. 2.3. Графік зміни часу виконання обчислень програми ПРГ1 в залежності від кількості ядер*

*Рис. 2.4. Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер*

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

## 2.6 Висновки до розділу 2

1. Мова Ada надає зручні інструменти організації паралельних обчислень для розробника програмного забезпечення. Досить просто реалізовувати монітор в мові Ada у вигляді захищеного модуля. Для копіювання були використані захищені функції які надають можливість процес копіювання зробити паралельним, адже функції захищеного модуля виконуються паралельно, тому що функції не мають побічних ефектів.
2. 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  = 3,75 забезпечує ПКС з Р = 4 та N=1200.
* Мінімальне значення  = 1,88764 (для багатоядерної системи) забезпечує ПКС з Р = 2 та N = 3000.

# РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП

В розділі розглянуто розробку та дослідження програми ПРГ2 для ПКС зі локальною пам’яттю.

**Математична задача:**  A = (B\*MO)\*α + C\*(MT\*MR)

**Мова програмування:** С++

**Технологія:** МРІ



*Рис. 3.1 Структура ПКС із локальною пам’яттю*

## 3.1 Розробка паралельного математичного алгоритму

Розрахунок даного математичного рівняння можна представити в наступній формі:

AH = (B\*MOH)\*α + C\*( MTн \* MR)

* N– розмірність векторів і матриць;
* P– кількість ядер;
* H = N/P;
* k = √p

## 3.2 Розробка алгоритмів процесів

Розмір посилки визначається за наступними формулами (i, j – номери процесорів, що посилають відповідні посилки, i = 1..q, j = 1..q):

Задача T1.1:

1. Введення: В,С,МТ
2. Передати В, С, МТ(n-h)/2 в T1.j и Ti.1
3. Прийняти α, MOG(i,1),, MRвід Ti.1
4. Прийняти α, MO G(1,j),, MR від T1.j
5. Обчислення AH = (B\*MOH)\*α + C\*(MTH\*MR)
6. Прийняти A h\*P/2 від Т1.j.
7. Прийняти A h\*P/2 від Тi.1.
8. Вивести А

Задача Тi.j, i = 1.., j = 1.., якщо i = j, то i ≠ 1 та i ≠ 0

1. Прийняти MT F(i-1,j), B, C від Тi-1,j.
2. Прийняти MT F(i,j-1), B, C від Тi,j-1.
3. Передати MT F(i,j) B, C в Тi-1.j, Ti.j-1
4. Прийняти α, MO G(i+1,j) MRвід Тi+1.j.
5. Прийняти α, MO G(i,j+1) MRвід Тi.j+1.
6. Передати α, MO G(i,j) MRв Тi-1.j, Ti.j-1.
7. Обчислення AH = (B\*MOH)\*α + C\*(MTH\*MR).
8. Прийняти AF(i+1,j) від Ti+1,j.
9. Прийняти AF(i,j+1) від Ti,j+1.
10. Передати AF(i,j) в Ti-1,j, Ti,j-1.

Задача :

1. Введення α, MO, MR
2. Передати α, MO(n-h)/2, MR
3. Прийняти B,C, MT від .
4. Прийняти B,C, MT від .
5. Обчислення AH = (B\*MOH)\*α + C\*(MTH\*MR)
6. Передати A в ,.

## 3.3 Розробка схеми взаємодії процесів

На основі алгоритму для всіх типів задач розроблено структурні схеми взаємодії задач (рис. 3.2). За допомогою цих схем можна наочно спостерігати як саме відбувається пересилка даних. Таке графічне зображення дозволяє простіше зрозуміти механізм пересилки повідомлень та взаємодії процесів.

Усі задачі логічно розбиваються на декілька типів:

1. Перша задача , в якій вводяться дані (B,C, MT) та виводиться результат (А).
2. Остання задача , в якій вводяться дані (MO, α, MR).
3. Інші задачі Тi,j, які тільки виконують обчислення та пересилку даних.

За такою організацією взаємодії процесів пересилка даних та обчислення розпаралелюється максимально, що забезпечує максимальну швидкість обчислення на даній ПКС з ЛП.



*Рис. 3.2 Схема взаємодії процесів*

## 3.4 Розробка програми ПРГ2

Програма написана на мові C++ з використанням бібліотеки MPI та складається з чотирьох модулів: основного Source.cpp та допоміжних matrix.cpp, vector.cpp і operations.cpp.

Основний модуль Source.cpp містить тільки функцію main, в якій і зосереджена основна логіка програми: пересилання і отримання повідомлень в залежності від номера задачі. Для передачі і приймання використовувались функції MPI\_Send і MPI\_Recv.

*col()* – повертає кількість рядків у матриці;

*row()* – повертає кількість стовпців у матриці;

*get(int i\_size, int j\_size)* – повертає *i,j* елемент матриці;

*set(int i\_size, int j\_size, double value)* – встановлює значення *value* для *i,j* –го елементу матриці;

*set(int i\_size, double value)* – встановлює значення *value* для *i –го* елементу вектору;

*fill(double value)* – заповнює весь масив значенням *value*.

Лістинг програми наведено у Додатку Ж.

## 3.5 Тестування програми ПРГ2

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 = 1200, 1800, 2400, 3000) і різною кількістю працюючих ядер ( P = 1, 2, 3, 4).

Для виміру часу використовується процедура clock() бібліотеки time.h мови програмування С++, що повертає поточний час системи. Різниця між часом системи після завершення обчислення і часом при запуску програми дає час виконання обчислення, що вимірюється в 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. 3.1-3.3.

Таблиця 3.1. Час виконання програми ПРГ2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 1200 | 15 | 8 | 6 | 4 |
| 1800 | 29 | 15 | 11 | 9 |
| 2400 | 50 | 26 | 19 | 15 |
| 3000 | 85 | 44 | 32 | 24 |

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

Таблиця 3.2. Коефіцієнти прискорення для програми ПРГ2

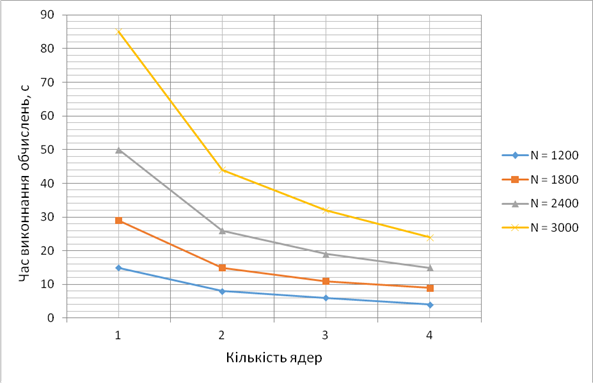
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | P | | | |
| 1 | 2 | 3 | 4 |
| 1200 | 1 | 1,875 | 2,5 | 3,75 |
| 1800 | 1 | 1,933333 | 2,636364 | 3,222222 |
| 2400 | 1 | 1,923077 | 2,631579 | 3,333333 |
| 3000 | 1 | 1,931818 | 2,65625 | 3,541667 |

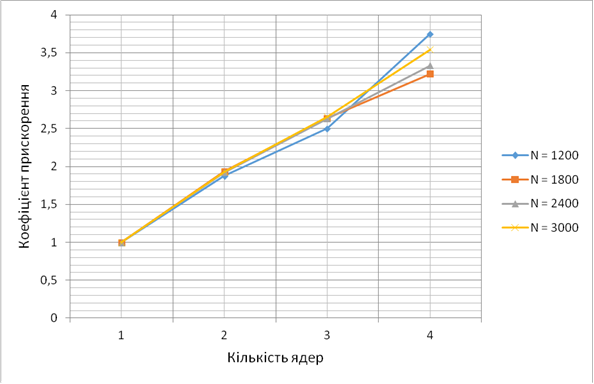
Коефіцієнти ефективності табл. 3.3 обчислено за даними табл. 3.2.

Таблиця 3.3. Коефіцієнти ефективності для програми ПРГ2

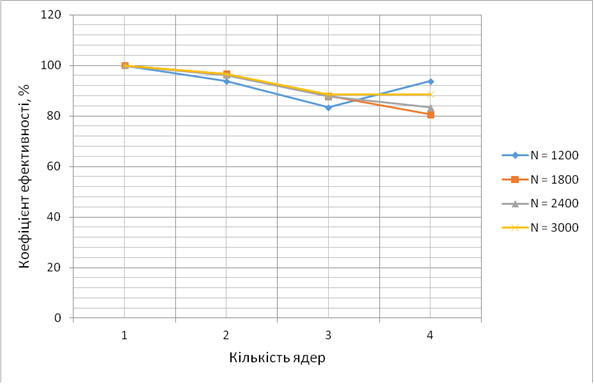
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 1200 | 100 | 93,75 | 83,33333 | 93,75 |
| 1800 | 100 | 96,66667 | 87,87879 | 80,55556 |
| 2400 | 100 | 96,15385 | 87,7193 | 83,33333 |
| 3000 | 100 | 96,59091 | 88,54167 | 88,54167 |

Використовуючи табл. 3.1‑3.3 побудовано графіки зміни часу виконання обчислень та коефіцієнтів прискорення і ефективності в залежності від *N* і  (рис. 3.3 – 3.5).



*Рис. 3.3. Графік зміни часу виконання обчислень програми ПРГ1 в залежності від кількості ядер*

*Рис. 3.4. Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер*

**

*Рис. 3.5. Графік зміни коефіцієнту ефективності програми ПРГ1 в залежності від кількості ядер*

## 3.6 Висновки до розділу 3

Виконано розробку програми ПРГ2 для ПКС із ЛП. При розробці використовувалась мова програмування C++. Для передачі повідомлень між процесами використовувалася бібліотека MPI.

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  = 3,75 забезпечує ПКС з Р = 4 та N=1200.
* Мінімальне значення  = 1,875 забезпечує ПКС з Р = 2 та N=1200.
* По графіку з рис. 3.5 видно, що коефіцієнт прискорення збільшується пропорційно до кількості використовуваних ядер.
* Коефіцієнти ефективності коливаються в межах від 80-100%, це свідчить про ефективність розробленого паралельного алгоритму.
* Максимальне значення = 96,66667% для ПКС з Р = 2 та N = 1500;
* Мінімальне значення = 80,55556% для ПКС з Р = 4 та N = 1500.

Не зважаючи на те, що пересилка даних між процесами – досить затратна операція, реалізація задачі з допомогою бібліотеки MPI демонструє досить ефективне використання ядер.

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

# ОСНОВНІ РЕЗУЛЬТАТИ І ВИСНОВКИ ПО РОБОТІ

Проведено огляд та порівняння реалізації семафорів у різних мовах програмування та бібліотеках. Механізм семафорів реалізований у більшості сучасних мов програмування високого рівня.

Розроблені алгоритми програм для багатоядерних обчислювальних систем, що вирішують поставлені математичні задачі. Реалізація задач відбувалась у системах з спільною та локальною пам’яттю.

У розділі 2 курсової роботи розроблено програму обчислення математичної задачі у системі з спільною пам’яттю. Програма створена на мові Ada із використанням механізму моніторів – захищеного модулю.

У розділі 3 курсової роботи розроблено програму обчислення математичної задачі у системі з локальною пам’яттю. Використовувалась мова програмування С++ та бібліотека MPІ.

Перевагою системи зі спільною пам’яттю над системою з локальною пам’яттю є спільні ресурси, які не копіюються для кожного процесу. Також системи зі спільною пам’яттю не потребують додаткових часових затрат на пересилку даних.

Перевагою системи із локальною пам’яттю над системою зі спільною пам’яттю є швидкодія та ефективність, що було продемонстровано в даній роботі.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гавва А.Е., Адское программирование[Текст] – Май 2004. – 433 с.
2. Лекції з операційних систем. [Електронний ресурс] <http://moodle.kstu.ru/mod/page/view.php?id=49>
3. Операційні системи мультипроцесорних ЕОМ [Електронний ресурс] <https://parallel.ru/krukov/lec2.html>
4. Жуков І.А., Корочкін О.В., Паралельні та розподілені обчислення: Навч. посібник [Текст]. – К.: Корнійчук, 2014. – 284 с.
5. Шилдт Г., Java. Полное руководство, 8-е изд.: Пер. с англ.[Текст] – М.: ООО “И.Д. Вильямс”,2012. – 1104 с .
6. Сайт про программування [Електронний ресурс] <http://metanit.com/java/tutorial/8.6.php>
7. Таненбаум Э., Современные операционные системы. 4-е изд.[Текст] — СПб.: Питер, 2015. — 1120 с.
8. Ada Conformity Assessment Authority. [Електронний ресурс] [http://www.ada-auth.org/](http://www.ada-auth.org/standards/12rm/html/RM-D-10.html)
9. Alan Burns, Concurrent and Real-Time Programming in Ada[eBook] – July 2007. – 478p.
10. Eckel, Bruce, Thinking in Java. – 4th ed. – 2006. – 1079p.

# ДОДАТКИ

## Додаток А. Структурна схема ПКС СП

# 

## Додаток Б. Схема алгоритму головної програми зі вказанням паралельних ділянок для ПРГ1

# 

## Додаток В. Схема алгоритму процесів для програми ПРГ1

## 

## Додаток Г. Лістинг програми ПРГ1 для ПКС СП

GNAT GPL 2015 (20150428-49)

Copyright 1992-2015, Free Software Foundation, Inc.

Compiling: coursework.adb

Source file time stamp: 2016-03-20 14:03:38

Compiled at: 2016-03-20 17:07:52

1. ----------------------------------------------------------------

2. -- Programming for parallel computer systems --

3. -- Coursework. Ada.Protected unit --

4. -- Func: A = (B\*MO)\*alfa + C\*(MT\*MR) --

5. -- IO-34 Krivonosov Oleksii --

6. ----------------------------------------------------------------

7.

8. with Ada.Text\_IO, Ada.Integer\_Text\_IO;

9. use Ada.Text\_IO, Ada.Integer\_Text\_IO;

10. with ada.calendar;

11. use ada.calendar;

12.

13. procedure Coursework is

14. N: constant := 320;

15. P: constant := 2;

16. H: integer := N/P;

17. subtype Index is Positive range 1..N;

18. type Vector is array (1..N) of Integer;

19. type Matrix is array (1..N) of Vector;

20.

21. MT,MO,MR: Matrix;

22. A, B, C:Vector;

23. alpha : Integer;

24. t: duration;

25. protected Control is

26. entry WaitForInput;

27. entry WaitForFinishCalc;

28.

29.

30.

31. procedure SetB\_C\_MT (B : in Vector; C : in Vector; MT : in Matrix);

32. function Get\_B return Vector;

33. function Get\_C return Vector;

34. function Get\_MT return Matrix;

35.

36. procedure Set\_alpha (alpha : in Integer);

37. function Get\_alpha return Integer;

38.

39.

40. procedure InputFinished;

41. procedure CalcFinished;

42.

43. private

44. MTm: Matrix;

45. alpha\_m: Integer;

46. Bm, Cm : Vector;

47. input: Integer := 0;

48. calc: Integer := 0;

49. end Control;

50. -- body of protected unit

51. protected body Control is

52.

53. entry WaitForInput when input = 2 is

54. begin

55. null;

56. end WaitForInput;

57.

58. entry WaitForFinishCalc when calc = P-1 is

59. begin

60. null;

61. end WaitForFinishCalc;

62.

63. procedure SetB\_C\_MT (B : in Vector; C : in Vector; MT : in Matrix) is

64. begin

65. MTm := MT;

66. Bm := B;

67. Cm := C;

68. end SetB\_C\_MT;

69.

70. procedure Set\_alpha( alpha : in Integer) is

71. begin

72. alpha\_m := alpha;

73.

74. end Set\_alpha;

75.

76. function Get\_B return Vector is

77. begin

78. return Bm;

79. end;

80. function Get\_C return Vector is

81. begin

82. return Cm;

83. end;

84. function Get\_MT return Matrix is

85. begin

86. return MTm;

87. end Get\_MT;

88.

89.

90.

91. function Get\_alpha return integer is

92. begin

93. return alpha\_m;

94. end Get\_alpha;

95.

96.

97. procedure InputFinished is

98. begin

99. input := input + 1;

100. end InputFinished;

101.

102. procedure CalcFinished is

103. begin

104. calc := calc + 1;

105. end CalcFinished;

106.

107. end Control;

108.

109. task type MyTask(id:integer);

110.

111. type MyTask\_ptr is access MyTask;

112. --Task body

113. task body MyTask is

114. MTi, tempMatrix :Matrix;

115. Bi, Ci, tempVector1, tempVector2 : Vector;

116. alphai : Integer;

117. left, right:integer;

118. temp, temp2: integer;

119. begin

120. -- Input in first task

121. if id=1 then

122. for i in 1..n loop

123. for j in 1..n loop

124. MT(i)(j) :=1;

125.

126. end loop;

127. B(i) := 1;

128. C(i) := 1;

129. end loop;

130.

131. Control.SetB\_C\_MT(B,C, MT);

132. Control.inputFinished;

133. end if;

134. --input in last task

135. if id=p then

136. for i in 1..n loop

137.

138. for j in 1..n loop

139. MO(i)(j):= 1;

140. MR(i)(j):=1;

141. end loop;

142. end loop;

143. alpha :=2;

144. Control.Set\_alpha(alpha);

145. Control.inputFinished;

146. end if;

147.

148. Control.WaitForInput;

149.

150. MTi:= control.Get\_MT;

151. Bi:= control.Get\_B;

152. Ci := control.Get\_C;

153. alphai := control.Get\_alpha;

154.

155.

156. left:= (id - 1) \* h + 1;

157. right:= id \* h;

158.

159.

160. --calculating main function in tasks 1..p

161. for i in left..right loop

162.

163. For j in 1..N loop

164.

165. temp := 0;

166. For k in 1..N loop

167. temp:= temp + MR(i)(k)\*MTi(k)(j);

168. end loop;

169. tempMatrix(i)(j) := temp;

170.

171. end loop;

172.

173. end loop;

174.

175. for i in left..right loop

176. temp :=0;

177. temp2 :=0;

178. for j in Index loop

179. temp := temp + tempMatrix(i)(j)\*Ci(j);

180. temp2 := temp2 + MO(i)(j) \* Bi(j);

181.

182. end loop;

183. tempVector1(i) := temp;

184. tempVector2(i) := temp2\*alphai;

185.

186.

187. end loop;

188.

189. for i in left..right loop

190. A(i) := tempVector1(i) + tempVector2(i);

191. end loop;

192. if id/=1 then

193. control.CalcFinished;

194. else

195. control.waitForFinishCalc;

196. put\_Line("A:");

197. if N < 8 then

198. For i in 1..N loop

199.

200. put(A(i));

201.

202.

203.

204. end loop;

205. else

206. put\_Line("Vector: ");

207. put(N);

208. end if;

209. Put\_Line(" ");

210. Put("Duration:");

211. Put\_Line(Duration'Image(seconds(Clock) - T));

212. end if;

213. end MyTask;

214.

215. procedure StartTasks is

216. tasks: array(1..p) of MyTask\_ptr;

217. begin

218. for i in 1..p loop

219. tasks(i):= new MyTask(id=>i);

220. end loop;

221. end StartTasks;

222. --------------------Main Program-----------------------

223. begin

224. put\_line("Course work. Ada protected unit");

225. t:= seconds(clock);

226. startTasks;

227. null;

228.

229. end Coursework;

229 lines: No errors

## Додаток Д. Структурна схема ПКС ЛП



## Додаток Е. Схема алгоритму головної програми зі вказанням паралельних ділянок для ПРГ2



## Додаток Ж. Лістинг програми ПРГ2 для ПКС СП

/\*\*

\* -------------------------------------------------------------------------

\* Паралельні та розподілені обчислення

\*

\*

\* Завдання: A = (B\*MO)\*α + C\*( MT \* MR)

\*

\* Виконав: Кривоносов Олексій

\* Група: ІО-34

\* Файл: Source.cpp

\* --------------------------------------------------------------------------

\*/

#define ignore MPI\_STATUS\_IGNORE

#include <iostream>

#include <mpi.h>

#include "operations.h"

#include <time.h>

using namespace std;

int N = 100,

P, H, k;

int main(int args, char\* argv[]) {

MPI\_Init(&args, &argv);

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &P);

H = N / P;

k = (int)sqrt((double)P);

int start = clock();

cout << "Task " << rank << " started" << endl;

int sizeA = H, sizeGeneralResources = H, offsGeneralResources = rank\*H;

if (rank == P - 1) {

sizeGeneralResources = N;

}

else if ((rank + 1) % k == 0) {//right

sizeGeneralResources = (rank + 1)\*H;

}

else {

sizeGeneralResources = ((rank + 1) % k) \* H;

sizeA = (k - (rank % k))\*H;

if (rank % k == 0) {//left

sizeA = (P - rank)\*H;

}

offsGeneralResources = rank % k;

}

int alpha;

Vector A(sizeA), B(N), C(N);

Matrix MR(sizeGeneralResources, N), MO(sizeGeneralResources, N), MT(N);

if (rank % k == 0) {// left

if (rank == 0) {

B.fill(1);

C.fill(1);

MT.fill(1);

MPI\_Send(B.get\_adr(0), N, MPI\_INT, k, 0, MPI\_COMM\_WORLD);

MPI\_Send(C.get\_adr(0), N, MPI\_INT, k, 0, MPI\_COMM\_WORLD);

MPI\_Send(MT.get\_adr(0), N\*N, MPI\_INT, k, 0, MPI\_COMM\_WORLD);

}

else {

MPI\_Recv(B.get\_adr(0), N, MPI\_INT, rank - k, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Recv(C.get\_adr(0), N, MPI\_INT, rank - k, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Recv(MT.get\_adr(0), N\*N, MPI\_INT, rank - k, 0, MPI\_COMM\_WORLD, ignore);

if (rank != P - k) {

MPI\_Send(B.get\_adr(0), N, MPI\_INT, rank + k, 0, MPI\_COMM\_WORLD);

MPI\_Send(C.get\_adr(0), N, MPI\_INT, rank + k, 0, MPI\_COMM\_WORLD);

MPI\_Send(MT.get\_adr(0), N\*N, MPI\_INT, rank + k, 0, MPI\_COMM\_WORLD);

}

}

MPI\_Send(B.get\_adr(0), N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD);

MPI\_Send(C.get\_adr(0), N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD);

MPI\_Send(MT.get\_adr(0), N\*N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD);

MPI\_Recv(&alpha, 1, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Recv(MO.get\_adr(0), H\*N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Recv(MR.get\_adr(0), H\*N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD, ignore);

}

else if ((rank + 1) % k == 0) {

if (rank == P - 1) {

MO.fill(1);

MR.fill(1);

MO.transpose();

MR.transpose();

MPI\_Send(MO.get\_adr(0), (P - k)\*H\*N, MPI\_INT, rank - k, 0, MPI\_COMM\_WORLD);

MPI\_Send(MR.get\_adr(0), (P - k)\*H\*N, MPI\_INT, rank - k, 0, MPI\_COMM\_WORLD);

}

else {

MPI\_Recv(MO.get\_adr(0), (P - k)\*H\*N, MPI\_INT, rank + k, 0, MPI\_COMM\_WORLD, ignore);

if (rank != k - 1) {

MPI\_Send(MO.get\_adr(0), (rank + 1 - k)\*H\*N, MPI\_INT, rank - k, 0, MPI\_COMM\_WORLD);

}

MPI\_Recv(MR.get\_adr(0), (P - k)\*H\*N, MPI\_INT, rank + k, 0, MPI\_COMM\_WORLD, ignore);

if (rank != k - 1) {

MPI\_Send(MR.get\_adr(0), (rank + 1 - k)\*H\*N, MPI\_INT, rank - k, 0, MPI\_COMM\_WORLD);

}

}

MPI\_Send(MT.get\_adr((rank + 1 - k)\*H\*N), (k - 1)\*H, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD);

MPI\_Send(MO.get\_adr((rank + 1 - k)\*H\*N), (k - 1)\*H\*N, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD);

MPI\_Send(&alpha,1, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD);

MPI\_Recv(B.get\_adr(0), N, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Recv(C.get\_adr(0), N, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Recv(MT.get\_adr(0), N\*N, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD, ignore);

}

else {

MPI\_Recv(MO.get\_adr(0), ((rank + 1) % k)\*H\*N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Send(MO.get\_adr(0), (((rank + 1) % k) - 1)\*H\*N, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD);

MPI\_Recv(MR.get\_adr(0), ((rank + 1) % k)\*H\*N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Send(MR.get\_adr(0), (((rank + 1) % k) - 1)\*H\*N, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD);

MPI\_Recv(&alpha, 1, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Send(&alpha, 1, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD);

MPI\_Recv(B.get\_adr(0), N, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Send(B.get\_adr(0), N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD);

MPI\_Recv(C.get\_adr(0), N, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Send(C.get\_adr(0), N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD);

MPI\_Recv(MT.get\_adr(0), N\*N, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Send(MT.get\_adr(0), N\*N, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD);

}

// Calculations

calculation(H, offsGeneralResources, alpha, B, C, MO, MR,MT, A);

if (rank % k == 0) {

MPI\_Recv(A.get\_adr(H), sizeA - H, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD, ignore);

if (rank != P - k) {

MPI\_Recv(A.get\_adr(k\*H), sizeA - k\*H, MPI\_INT, rank + k, 0, MPI\_COMM\_WORLD, ignore);

}

if (rank != 0) {

MPI\_Send(A.get\_adr(0), sizeA, MPI\_INT, rank - k, 0, MPI\_COMM\_WORLD);

}

}

else if ((rank + 1) % k == 0) {

MPI\_Send(A.get\_adr(0), sizeA, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD);

}

else {

MPI\_Recv(A.get\_adr(H), (P - k)\*H, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD, ignore);

MPI\_Send(A.get\_adr(0), sizeA, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD);

}

cout << "Task " << rank << " finished" << endl;

MPI\_Finalize();

if (rank == 0) {

if (N < 100) {

cout << A;

}

cin >> N;

system("pause");

}

return 0;

}

/\*\*

\* -------------------------------------------------------------------------

\* Паралельні та розподілені обчислення

\*

\*

\* Завдання: A = (B\*MO)\*α + C\*( MT \* MR)

\*

\* Виконав: Кривоносов Олексій

\* Група: ІО-34

\* Файл: Matrix.cpp

\* --------------------------------------------------------------------------

\*/

#include "matrix.h"

Matrix::Matrix(size\_t rows, size\_t cols) :

Vector(rows \* cols),

rows(rows),

cols(cols) { }

Matrix::Matrix(size\_t N) :

Vector(N \* N),

rows(N),

cols(N)

{ }

Matrix::Matrix(const Matrix &other) :

Vector(other.rows \* other.cols),

rows(other.rows),

cols(other.cols) {

for (size\_t i = 0; i < this->rows; i++) {

for (size\_t j = 0; j < this->cols; j++) {

Matrix::set(i, j, other.get(i, j));

}

}

}

Matrix::~Matrix(){}

std::ostream &operator<<(std::ostream &ostr, const Matrix &matrix) {

for (size\_t i = 0; i < matrix.rows; i++) {

for (size\_t j = 0; j < matrix.cols; j++) {

ostr << matrix.get(i, j) << "\t";

}

ostr << endl;

}

return ostr;

}

void Matrix::transpose() {

Matrix copy(\*this);

for (size\_t i = 0; i < this->rows; i++) {

for (size\_t j = 0; j < this->cols; j++) {

Matrix::set(j, i, copy.get(i, j));

}

}

}

/\*\*

\* -------------------------------------------------------------------------

\* Паралельні та розподілені обчислення

\*

\*

\* Завдання: A = (B\*MO)\*α + C\*( MT \* MR)

\*

\* Виконав: Кривоносов Олексій

\* Група: ІО-34

\* Файл: operations.cpp

\* --------------------------------------------------------------------------

\*/

#include "operations.h"

void calculation(

const size\_t H,

const size\_t offsGR,

const int alpha,

const Vector &B,

const Vector &C,

const Matrix &MO,

const Matrix &MR,

const Matrix &MT,

Vector &A)

{

Matrix Buf1(MT.cols, MT.cols);

Buf1.fill(1);

Vector Buf1V(offsGR);

Vector Buf2V(offsGR);

long int sum1, sum2, sum3;

for (size\_t i = 0; i < H; i++) {

for (size\_t j = 0; j < MT.cols; j++) {

sum1 = 0;

for (size\_t k = 0; k < MT.cols; k++) {

sum1 += MR.get(i + offsGR, k) \* MT.get(j, k);

}

Buf1.set(i + offsGR, j, sum1);

}

}

for (size\_t i = 0; i < H; i++){

sum2 = 0;

sum3 = 0;

for (int j = 0; j < B.cols; j++){

sum2 += Buf1.get(i + offsGR, j) \* C.get(j);

sum3 += MO.get(i + offsGR, j) \* B.get(j);

}

Buf1V.set(i, sum2\*alpha);

A.set(i, sum3 + Buf1V.get(i));

}

}

/\*\*

\* -------------------------------------------------------------------------

\* Паралельні та розподілені обчислення

\*

\*

\* Завдання: A = (B\*MO)\*α + C\*( MT \* MR)

\*

\* Виконав: Кривоносов Олексій

\* Група: ІО-34

\* Файл: vector.cpp

\* --------------------------------------------------------------------------

\*/

#include "vector.h"

Vector::Vector(size\_t cols) :

cols(cols),

data(new long int[cols])

{

}

Vector::~Vector() {

delete[] data;

}

void Vector::fill(long int value) {

for (size\_t i = 0; i < this->cols; i++) {

set(i, value);

}

}

std::ostream &operator<<(std::ostream &ostr, const Vector &vector) {

for (size\_t i = 0; i < vector.cols; i++) {

ostr << vector.data[i] << " ";

}

return ostr;

}