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 ФІОТ групи ІО - 22

напряму підготовки 050102 «Комп’ютерна інженерія»

\_\_\_\_\_\_\_\_\_\_\_\_Сагач Я.І.\_\_\_\_\_\_\_\_\_\_\_\_\_

(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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Сагач Ярослав Ігорович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(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

- засоби роботи з процесами у бібліотеці OpenMP ?????

- математична задача MA=MB\*(MC\*MZ)+min(E)\*MO

- структури ПКС ОП та ПКС ЛП

- мови і бібліотеки програмування: C++ (бібліотека OpenMP)

- засоби організації взаємодії процесів: критична секція, атомарна змінна, механізм бар’єрної синхронізації

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

- огляд засобів роботи з процесами у сучасних мовах програмування

- 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 \_\_\_\_\_12.02.2015\_\_\_\_\_\_\_

#### 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 \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_**Сагач Я. І**.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

( підпис ) (прізвище та ініціали)

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

( підпис ) (прізвище та ініціали)

**ЗМІСТ**

ВСТУП…………………………………………………………………………… 3

РОЗДІЛ 1. ПРОЦЕСИ У СУЧАСНИХ МОВАХ ПРОГРАМУВАННЯ …... 5

* 1. Процеси та засоби організації їх взаємодії у мові Java..…………. 5
  2. Процеси та засоби організації їх взаємодії у мові C#..…. ……… 9
  3. Процеси та засоби організації їх взаємодії у мові Python….…….13
  4. Процеси та засоби організації їх взаємодії у мові Scala…………17
  5. Висновки до розділу 1……………………………………………… 22

РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС З ОП…………………20

* 1. Розробка паралельного математичного алгоритму……………… 20
  2. Розробка алгоритмів процесів……………………………………… 22
  3. Розробка схеми взаємодії процесів…………………………………24
  4. Розробка програми ПРГ1…………………………………………… 26
  5. Тестування програми ПРГ1 …………………………………………27
  6. Висновки до розділу 2 …………………………………………… 29

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС З ЛП ……..………… 30

* 1. Розробка паралельного математичного алгоритму……………… 30
  2. Розробка алгоритмів процесів…………………………………… 32
  3. Розробка схеми взаємодії процесів………………………… … 34
  4. Розробка програми ПРГ2……………………………………… 36
  5. Тестування програми ПРГ2……………………………………… 37
  6. Висновки до розділу 3………………………………………………39

ОСНОВНІ РЕЗУЛЬТАТИ І ВИСНОВКИ ДО РОБОТИ………………………40

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ……………………………………. 43

ДОДАТКИ……………………………………………………............................. 45

**РОЗДІЛ 1. ПРОЦЕСИ У СУЧАСНИХ МОВАХ ПРОГРАМУВАННЯ**

* 1. **Процеси та засоби організації їх взаємодії у мові Java**

Java – об’єктно-орієнтована мова програмування. Була випущена компанією Sun Microsystems у 1995 році. Наразі підтримкою мови займається компанія Oracle. Синтаксис мови схожий на синтаксис С та С++. Програми написані на Java компілюються у байт-код, що при виконанні інтерпретується віртуальною машиною для конкретної платформи. [1]

У Java підтримка багатопотокового програмування є вбудованою. Кожна програма містить щонайменше один потік виконання. Виконання кожної програми починається зі створення головного потоку. Він створюється автоматично, але його роботою можна керувати через об’єкт класу Thread. Отримати даний об’єкт можна за допомогою методу currentThread(). Багатопотоковість у Java забезпечується класами Thread, ThreadGroup та інтерфейсом Runnable.

Щоб створити окремий потік, можна скористатись двома шляхами: реалізувати виконавчий інтерфейс Runnable чи розширити клас Thread, шляхом його наслідування.

У процесі створення потоку за допомогою інтерфейсу Runnable слід створити клас, що реалізує даний інтерфейс. Цей інтерфейс має єдиний метод run(), який необхідно перевизначити. У цьому разі створюється точка входу іншого, конкуруючого потоку, який завершується при завершенні методу run(). Потім необхідно оголосити змінну, яка буде містити об’єкт класу Thread. Для запуску потоку необхідно викликати метод start().

**public** **class** ThreadImplementation **implements** Runnable {

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

System.out.println("Thread started!");

}

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

Thread thread = **new** Thread(**new** ThreadImplementation());

thread.start();

}

}

У процесі розширення класу Thread слід створити клас, який розширює клас Thread. У розширеному класі необхідно перевизначити метод run(). Потім необхідно оголосити змінну, яка буде містити об’єкт нового класу, і викликати метод start(). [1]

**public** **class** NewThread **extends** Thread {

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

System.out.println("Thread started!");

}

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

NewThread thread = **new** NewThread();

thread.start();

}

}

В обох випадках для запуску потоку використовується метод start(). Цей метод викликає відповідний метод run() потоку.

Клас Thread містить набір методів для керування потоками. Клас містить один конструктор без параметрів та шість конструкторів з параметри, що дозволяє задати об’єкту класу інтерфейс Runnable, ім’я потоку або ім’я групи потоків. [2]

Клас містить три константи : MAX\_PRIORITY = 10, MIN\_PRIORITY = 1 та NORM\_PRIORITY = 5, що задають найвищий, найнижчий та середній пріоритети відповідно.

Основні методи класу Thread:

1. int activeCount() – повертає поточну кількість потоків у групі;
2. Thread currentThread() – повертає поточний об’єкт Thread;
3. boolean interrupted() – повертає true, якщо потік може бути перерваний;
4. void sleep(long millis) throws InteruptedException – переводить потік у стан очікування на millis мілісекунд;
5. void yield() – примушує потік поступитися процесором для іншого потоку з більшим чи однаковим пріоритетом;
6. final String getName() – повертає ім’я потоку;
7. final void setName(String name) – встановлює ім’я потоку;
8. final int getPriority() – повертає пріоритет потоку;
9. final void setPriority(int newPriority) – встановлює пріоритет потоку;
10. final void join() throws InterruptedException – примушує потік чекати завершення викликаного потоку;
11. final void suspend() – призупиняє виконання потоку;
12. final void resume() – відновлює роботу призупиненого потоку;
13. final void stop() – зупиняє виконання потоку;
14. final boolean isAlive() – повертає true, якщо потік діючий;
15. synchronized native void start() – запускає потік на виконання.

Засоби мови програмування Java забезпечують вирішення задач синхронізації та взаємного виключення.

Ключем до вирішення задачі взаємного виключення є концепція монітора. Монітор – це програмний модуль, доступ до якого у один момент часу може мати лише один потік. Монітор містить у собі змінні та процедури, що забезпечують роботу з ними. При спробі процесу доступитися до монітору, що вже використовується, цей процес блокується і розміщується у черзі. Концепція монітору забезпечує виконання паралельних програм без помилок, що спричинені одночасним використанням спільних ресурсів.

Монітори не є вбудованим засобом мови Java. Але є можливість створення класу. Методи даного класу повинні мати властивості процедур монітору. У Java це можна зробити за допомогою модифікатору synchronized. Даний модифікатор дозволяє організувати послідовний доступ до методів класу монітору. Тобто доступ до методу, який позначений модифікатором synchronized, може мати лише один потік. Усі інші потоки при спробі доступитись до методу будуть заблоковані доти, доки не завершиться виконання методу. [2]

**public** **class** Monitor {

**private** **int** counter **=** 0**;**

**public** **synchronized void** incCounter() {

counter++;

}

}

Для вирішення задач синхронізації процесів у мові Java використовуються такі методи, як wait(), notify() та notifyAll():

1. wait() – наказує потоку заблокуватись доти, доки інший потік не викличе метод notify();
2. notify() – активізує перший потік, який викликав метод wait() на тому ж об’єкті;
3. notifyAll() – активізує всі потоки, які викликали wait() одного і того ж об’єкта. При чому першим запускається потік з найвищим пріоритетом.

Особливістю даних методів є те, що вони можуть взаємодіяти при наявності прив’язки до одного і того ж об’єкта. А також вони можуть викликатись лише всередині синхронізованого методу. Тобто такого що містить модифікатор synchronized.

Також існує пакет java.util.concurrent для роботи з потоками у мові Java. Автором даного пакету є професор Даг Лі з університету Осуіго штату Нью-Йорк. Наразі пакет досить часто використовується у розробці програмного забезпечення. Даний пакет можна поділити на деякі базові складові, які мають різний функціонал та призначення. А саме, такі :

1. Concurrent Collections – набір колекцій, що більш ефективно працюють у багатопоточному середовищі ніж стандартні універсальні колекції з пакету java.util;
2. Queues – не блокуючі і блокуючі черги з підтримкою багато поточності;
3. Synchronizers – допоміжні утиліти для синхронізації потоків.
4. Executors – містить у собі фреймворки для створення пулів потоків, планування роботи асинхронних задач з отриманням результату;
5. Locks – містить альтернативні та більш гнучкі механізми синхронізації, ніж базові synchronized, wait, notify, notifyAll;
6. Atomics – класи з підтримкою атомарних операцій над примітивами та посиланнями. [3]
   1. **Процеси та засоби організації їх взаємодії у мові C#**

C# - об’єктно-орієнтована мова програмування з безпечною системою типізації для платформи .NET. Розроблена Андерсом Гейлсбергом, Скотом Вілтамутом та Пітером Гольде при фірмі Microsoft.

Синтаксис C# близький до С++ і Java. Мова має строгу типізацію, підтримує поліморфізм, перевантаження операторів, вказівники та функції члени класів, атрибути, події, властивості, винятки. [3]

Як і усі сучасні мови програмування, С# має засоби програмування паралельних процесів. Як і у мові Java, кожна програма містить хоча б один потік. Адже виконання програми відбувається зі створення головного потоку. Кожен з потоків виконується незалежно від інших потоків.

Для початку роботи з потоками, необхідно підключити простір імен System.Threading, додавши у початок файлу наступну директиву:

using System.Threading;

Концепція створення потоку у C# дуже схожа на створення потоків у Java, хоча дещо і відрізняється. Для створення потоку також використовується клас Thread, проте на відміну від Java, у C#, замість методу run(), потрібно визначити та передати у конструктор потоку потоковий метод. Цей метод може мати будь-яку назву. Запускається потік, аналогічно до Java, методом Start(). [3]

static void Main()

{

new Thread(Go).Start(); // Виконати Go() в новому потоці

Go(); // Виконати Go() у головному

}

static void Go()

{

Console.Write(“Go started!”);

}

Також існують інші конструктори класу Thread, наприклад такі, що працюють з делегатами. [4]

У мові C# присутні такі основні методи для роботи з потоками:

1. Start() – змінює стан даного потоку на ThreadState::Running;
2. Start(Object) - змінює стан даного потоку на ThreadState::Running, при цьому передаючи деякі дані потоку;
3. Yield() – примушує потік, що викликав метод, передати виконання іншому потоку, який готовий до виконання. Операційна система сама обирає якому потоку передати виконання;
4. Sleep(TimeSpan) ­– блокує потік на заданий проміжок часу;
5. Join() – блокує потік, що викликав метод, до закінчення іншого потоку.
6. Abort() – викликає виключення ThreadAbortException, щоб почати процес завершення роботи потоку. При цьому виконується код із блоку finally потоку.

У кожного потоку є набір властивостей, що можна переглянути чи змінити. Наприклад, за допомогою властивості Name можна дізнатись чи змінити назву даного потоку.

Перевірити стан потоку можна за допомогою методу isAlive().

Пріоритет потоку визначається типом enum ThreadPriority{Lowest, BelowNormal, Normal, AboveNormal, Highest}. Зчитати чи змінити значення пріоритету можна за допомогою властивості Priority.

Найпростішим шляхом вирішення задачі взаємного виключення є оператор lock.

lock (object1) { ... }

Діє він наступним чином. Коли один з потоків доходить до оператора lock, він перевіряє чи не заблокований object1. Якщо ні – виконує блок коду вказаний у дужках. Якщо об’єкт заблоковано, то потік блокується до тих пір, поки object1 не буде знову розблокований. [5] Цей механізм є реалізацією механізму критичних секцій.

На відміну від мови програмування Java, засоби мови C# підтримують механізми семафорів та мютексів, що робить досить нескладним вирішення задач синхронізації та взаємного виключення процесів.

Механізм семафорів реалізований у класі Semaphore, що забезпечує виконання двох основних операцій притаманних семафорам: P(S) та V(S).

Кожен окремий семафор є об’єктом класу Semaphore .

Semaphore S = new Semaphore(p1,p2);

Перший аргумент задає початкове значення семафору, другий – максимальне значення семафору.

Основними методами роботи з семафором є методи WaitOne() та Release(), що відповідають операціям P(S) та V(S). Параметром методу Release є число на яке збільшується значення семафору.

using System.Threading;

public class Example

{

private static Semaphore \_pool;

public static void Main()

{

\_pool = new Semaphore(0, 3);

for(int i = 1; i <= 5; i++)

{

Thread t = new Thread(new ParameterizedThreadStart(Worker));

t.Start(i);

}

Thread.Sleep(500);

Console.WriteLine("Main thread calls Release(3).");

\_pool.Release(3);

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

}

private static void Worker(object num)

{

Console.WriteLine("Thread {0} begins " +

"and waits for the semaphore.", num);

\_pool.WaitOne();

Console.WriteLine("Thread {0} enters the semaphore.", num);

Thread.Sleep(1000);

Console.WriteLine("Thread {0} releases the semaphore.", num);

Console.WriteLine("Thread {0} previous semaphore count: {1}",

num, \_pool.Release());

}

}

Також у мові C# є реалізація механізму мютексів. Дана реалізація дуже подібна до реалізації семафорів. Але, на відміну від семафора, мютекс може мати лише два стани і фактично є бінарним семафором. Для створення мютексу достатньо створити екземпляр класу Mutex :

Mutex M = new Mutex(p1);

Параметр конструктора визначає початковий стан мютексу. Реалізаціями операцій P(M) та V(M) є методи WaitOne() та ReleaseMutex().

Мова C# також підтримує механізм подій, що використовується для вирішення задачі синхронізації процесів. Даний механізм реалізований у класах EventWaitHandle, AutoResetEvent, ManualResetEvent. Зокрема клас ManualResetEvent дозволяє вручну скидати стан події, що забезпечує синхронізацію більш ніж двох потоків (один потік може передати сигнал більш ніж двом потокам). Ці класи є наслідками абстрактного класу WaitHandle. Для того щоб передати сигнал про подію необхідно викликати метод Set(). Для забезпечення очікування на сигнал необхідно використати метод WaitOne(). Для переведення події у несигнальний стан потрібно викликати метод Reset(). [2]

Також простір імен System.Threading містить такі базові типи, як Interlocked та Monitor. Клас Interlocked забезпечує синхронізований доступ до спільних даних. Він надає атомарні операції для змінних, які є загальнодоступними для декількох потоків. Основними методами є : Increment(), Decrement(), Exchange(), CompareExchange().

Клас Monitor забезпечує синхронізацію потоків за допомогою блокування і керування очікуванням. Об’єкти цього класу надають можливість синхронізувати доступ до блоку коду за допомогою отримання і звільнення блокування на заданому об’єкті. [6] Ці дії можна виконати за допомогою таких методів:

1. Enter() – використовується для встановлення блокування;
2. TryEnter() – намагається отримати блокування вказаного об’єкта;
3. Exit() – зняття блокування вказаного об’єкта;
4. Wait() – знімає блокування, якщо воно встановлене, та очікує сповіщення. При отриманні сигналу метод повертається і знову отримує блокування;
5. Pulse() та PulseAll() – сигналізують про перехід до наступного потоку у черзі очікування.

Дані механізми забезпечують реалізацію різноманітних схем взаємодії потоків. Проте завжди існує небезпека взаємного блокування (deadlock). Щоб цього не сталось необхідно до написання коду програми побудувати діаграму багато потокового додатку. У разі виникнення помилки, відбудеться виключення, яке потім можна буде виправити у процесі відладки коду.

**1.3 Процеси та засоби організації їх взаємоді ї умові Python**

Python – інтерпретована об’єктно-орієнтована мова програмування високого рівня з динамічною семантикою. Була розроблена у 1990 році Гвідо ван Россумом. Структури даних високого рівня разом із динамічною семантикою та динамічним зв’язуванням роблять її привабливою для швидкої розробки програм, а також як засіб поєднання уже існуючих компонентів. Python підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. [9]

Python містить декілька модулів для організації роботи з процесами. Зокрема модулі threading та multiprocessing.

Усі потоки, що створються у програмі, є класами, які наслідуються від класу threading.Thread. Головний метод потоку – run() необхідно перевизначити. Також необхідно викликати \_\_init\_\_ батьківського класу у конструкторі.

from threading import Thread

class ThreadImpl(Thread):

   def \_\_init\_\_(self, name):

        Thread.\_\_init\_\_(self, name=name)

    def run(self):

        print 'Thread started!'

t = ThreadImpl('ThreadImpl')

t.start()

Метод start() запускає потік на виконання. Для перевірки завершення дочірнього потоку необхідно викликати функцію is\_alive(), аргумент якої задає ім’я потоку. Для того щоб дізнатись, який потік зараз виконується, необхідно скористатись функцією current\_thread().

Слід зазначити, що механізми створення потоків у Python схожі на мови Java і C#. Тому існує також інший спосіб створення потоку – при створенні екземпляру класа Thread передати callable-об’єкт через параметр target. Іншими словами – передати у конструктор потоку об’єкт потокової функції.

from threading import Thread, current\_thread

def func():

    print 'Thread started!'

t = Thread(target=func)

t.start()

Метод join() дозволяє заблокувати потоки доти, доки не виконається заданий потік.

У мові Python існують деякі атомарні операції, що не потребують синхронізації. Атомарні операції – операції, що виконуються за один крок, без можливості переривання їх іншими потоками. Ось деякі з потокобезпечних операцій:

* читання чи зміна одного атрибуту об’єкта;
* читання чи зміна однієї глобальної змінної;
* вибір елемента зі списку;
* модифікація списку за допомогою методу append;
* вибір елементу зі словника;
* модифікація словника шляхом виклику методів додавання елементу чи методу clear.

Проте операції, що зчитують змінну чи атрибут, змінюють його а потім записують його знову – не є потокобезпечними. [7]

Найпростішим методом синхронізації потоків є використання типу Lock. Lock – блокування, що реалізує два методи: acquire() та release(), які захоплюють і звільняють блокування відповідно.

Існують певні проблеми з простими блокуваннями у Python. Звичайне блокування не перевіряє потік, що захопив його. Тому будь-який потік при спробі захопити блокування буде заблокований, навіть якщо цей потік вже володіє цим блокуванням на даний момент часу. Рішенням даної проблеми є використання так званого реентерабельного блокування RLock. Реалізація якого також присутня у модулі threading.

Клас RLock – варіант простого блокування, яке блокує потік лише тоді, коли блокування захоплене іншим потоком, у той час коли звичайне блокування може заблокувати той потік, який спробує захопити його повторно.

Мова Python також підтримує механізм семафорів. Семафор – є більш складним і довершенішим механізмом блокувань. В залежності від того, як буде ініціалізовано семафор, він може дати можливість декільком потокам виконувати один і той же блок коду одночасно.

Модуль threading надає дві реалізації механізму семафорів: клас Semaphore та клас BoundedSemaphore. Але клас Semaphore надає можливість викликати метод release нескінченну кількість разів. Тому, для уникнення помилок, краще використовувати клас BoundedSemaphore, який рахує кількість викликів release, для яких до цього не був викликаний метод acquire.

semaphore = threading.BoundedSemaphore()

semaphore.acquire() # зменшує лічильник

... доступ к общему ресурсу

semaphore.release() # збільшує лічильник

Метод acquire() спочатку перевіряє значення лічильника семафору. Якщо значення дорівнює 0, то потік блокується. Якщо значення лічильника більше ніж 0, то потік продовжить свою роботу, при цьому значення лічильника стане на 1 менше.

Якщо не передавати параметр у конструктор класу BoundedSemaphore, то він буде ініціалізований 1. Тобто буде створений звичайний бінарний семафор. [7]

Також одним з методів синхронізації потоків, який присутній у мові Python є механізм подій.

Подія – найпростіший об’єкт синхронізації, що є фактично, прапорцем. Потік може очікувати встановлення цього прапорця, або встановлювати і скидати його самостійно. Подія має два стани : сигнальний та не сигнальний.

event = threading.Event()

event.wait()

event.set()

event.clear()

Якщо стан події сигнальний то метод wait не заблокує потік, а якщо стан – не сигнальний, то метод wait заблокує потік до тих пір, доки стан не буде встановлений у сигнальний. Будь-яка кількість потоків може очікувати одну і те ж саму подію одночасно.

Умовна змінна – більш довершені ший варіант реалізації механізму подій. Реалізація якого дуже схожа на реалізацію цього механізму у мові Java.

condition = threading.Condition()

Метод wait звільняє блокування, блокує даний потік доти, доки інший потік не викличе метод notify чи notifyAll, на даній умовній змінній, потім знову захоплює блокування умовної змінної. Метод notify розблоковує лише один з заблокованих (чекаючих) потоків, а метод notifyAll розблоковує усі потоки, що були заблоковані.

Об’єкт умовної змінної пов’язаний з блокуванням, тому блокування має бути перехоплене раніше, ніж буде проведений доступ до умовної змінної. Аналогічно, це блокування повинно бути звільнене, коли потік більше не використовує умовну змінну.

Для того щоб пов’язати умовну змінну з блокуванням, необхідно передати об’єкт блокування у конструктор умовної змінної. [7]

lock = threading.RLock()

condition\_1 = threading.Condition(lock)

condition\_2 = threading.Condition(lock)

У модулі mutex присутній клас Mutex, що реалізує механізм вирішення задач взаємного виключення. Клас має два методи lock(function, argument) та unlock(). Метод lock() виконує функцію доти у тому разі, якщо мютекс є вільним. Якщо мютекс є заблокованим, то функція додається до черги і очікує свого виконання. Тому першою дією методу lock() є перевірка стану мютексу. Метод unlock() розблоковує мютекс. [8]

Проте існує деяка проблема з багатопотоковістю у Python. Цю проблему спричиняє така річ, як глобальне блокування інтерпретатора (GIL). Основний інтерпретатор Python при своїй роботі використовує велику кількість даних. Для уникнення пошкодження цих даних при спільній модифікації з різних потоків перед початком виконання декількох інструкцій потік інтерпретатора захоплює GIL. Внаслідок чого в кожен момент часу може виконуватись тільки один потік, навіть якщо на комп’ютері є кілька процесорів або процесорних ядер. З одного боку це добре, тому що дані є захищеними від пошкоджень без реалізації додаткових механізмів. Проте тоді втрачається основна ідея паралельного програмування і ресурси якими володіє комп’ютер будуть використовуватися не на повну потужність.

Існує декілька способів вирішення даної проблеми. Перший спосіб – відмова від спільного використання змінюваних даних. При цьому усі дані будть копіюватися для кожного з потоків.

Проте існує простіший спосіб – використання модуля multiprocessing замість threading. Даний модуль повністю імітує семантику стандартного модуля threading, але використовує процеси замість потоків.

T1 = Process(target=ThreadFunction)

Також є безліч модулів, що спрощують написання паралельних за стосунків на Python, таких як parallelpython, Pypar, pympi.

**1.4 Процеси та засоби організації їх взаємодії у мові Scala**

Scala – мультипарадигмова мова програмування, що поєднує властивості об’єктно-орієнтованого та функційного програмування. Назва Scala утворена зі слів “scalable” та “language”, для того щоб зазначити, що мова може рости разом з вимогами користувачів.

Програми на Scala запускаються на віртуальній машині Java вище версії 1.5 за умови включення до дистрибутиву файлу scala-library.jar . Також Scala сумісна із існуючими програмами на Java, тобто код Scala може викликатися із Java-програм і навпаки. Дистрибутив Scala, включаючи компілятор і бібліотеки, випущено під BSD похідною ліцензією. [9]

Паралельні обчислення у Scala побудовані поверх моделі паралельних обчислень у Java.

Для створення об’єкту потоку викликається конструктор класу Thread у який ми маємо передати параметр Runnable. Для того щоб запустити потік на виконання, потрібно викликати метод start. [9]

scala> val thread = new Thread(new Runnable {

def run() {

println("tread started!")

}

})

thread: java.lang.Thread = Thread[Thread-3,5,main]

scala> thread.start

У мові Scala наявні інструменти для синхронізації та взаємного виключення потоків. Основні механізми є незмінними, тому у мові присутні семафори, мютекси, сигнали, монітори та інші засоби синхронізації процесів.

В Scala основні засоби взаємного виключення процесів забезпечує монітор. Кожен екземпляр класу AnyRef може будти використаний у якості монітору шляхом виклику хоча б одного з цих методів:

def synchronized[a] (exec: => a): a

def wait(): unit

def wait(msec: Long): unit

def notify(): unit

def notifyAll(): unit

Метод synchronized обчислює свій аргумент в режимі взаємного виключення. Тобто у будь-який момент часу лише один потік може виконувати аргумент synchronized даного монітору.

Потік може бути призупиненим на моніторі, очікуючи сигналу. Потік, що викликав метод wait, чекає до виклику іншим потоком методу notify. Крім цього, метод wait(msec) чекає до моменту коли буде отриманий сигнал чи до того моменту коли закінчиться вказаний час. Метод notifyAll розблоковує усі потоки, що очікували на сигнал. Ці методи, а також клас Monitor є примітивами у мові Scala. [9]

Також існує метод, що має назву синхронізована змінна. Синхронизована змінна представлена у вигляді класу SyncVar пакету scala.concurrent. Клас надає методи get та put для читання та запису змінної. Операція get блокує потік доти, доки змінна невизначена. Операція unset скидає змінну в невизначений стан. Стандарта реалізація синхронізованих змінних :

package scala.concurrent

class SyncVar[A] {

private var isDefined: Boolean = false

private var value: A = \_

def get = synchronized {

while (<tt>isDefined</tt>) wait()

value

}

def set(x: A) = synchronized {

value = x ; isDefined = true ; notifyAll()

}

def isSet: Boolean = synchronized {

isDefined

}

def unset = synchronized {

isDefined = false;

}

}

Futures(майбутнє) – це значення, яке обчислюється паралельно деякому клієнтському потоку, для того щоб бути використаним ним дещо пізніше. Future використовується для полегшення паралельної обробки ресурсів. Типове використання :

import scala.concurrent.ops.\_

…

val x = future(someLengthyComputation)

anotherLengthyComputation

val y = f(x()) + g(x())

Метод future отримує у якості параметра обчислення р, яке і буде виконуватись. Потім метод визначає деякий вираз result, який приймає параметр, що являє собою результат обчислення. Потім метод відокремлює новий потік, який обчислює результат і викликає вираз result по закінченню. Паралельно цьому потоку future повертає анонімну функцію типу А. Коли ця функція викликається вона очікує виклик значення result, коли це стається вона повертає результат обчислень. [9]

У мові Scala семафор реалізований у класі Lock, і фактично є бінарним семафором. Клас надає можливості виконувати такі операції, як заняття та звільнення семафору.

package scala.concurrent

class Lock {

var available = true

def acquire = synchronized {

while (!available) wait()

available = false

}

def release = synchronized {

available = true

notify()

}

}

Існує більш складна форма синхронізації процесів у мові Scala, що має назву “читачі/письменники”. Потоки-читачі мають доступ до спільного ресурсу, але не можуть модифікувати його. Потоки-письменники можуть як читати, так і змінювати ресурс. Для синхронізації “читачів” та “письменників” необхідно реалізувати дії startRead, startWrite, endRead, endWrite так, щоб:

* одночасно може бути декілька читачів;
* у будь-який момент може бути лише один письменник;
* запис має більший пріоритет над читанням, але перервати процес читання не може.

Реалізація даного механізму основана на концепції поштової скриньки. Поштові скриньки – високорівневі гнучкі конструкції для синхронізації та взаємодії процесів. Вони дозволяють посилати і отримувати повідомлення(у даному контексті повідомлення – деякий довільний об’єкт). Поштові скриньки визначають наступні методи:

**class** MailBox {

**def** send(msg: Any): unit

**def** receive[A](f: PartialFunction[Any, A]): A

**def** receiveWithin[A](msec: Long)(f: PartialFunction[Any, A]): A

}

Метод send поміщує повідомлення у поштову скриньку додаючи його у кінець зв’язного списку, а також надсилає звіт усім отримувачам яких цікавить даний тип повідомлення. Receive блокує даний потік до моменту, доки не буде отримане повідомлення, що відповідає переданому у метод функціональному блоку.

У мові Scala існує можливість створення потоку з можливостями поштової скриньки. Такі об’єкти називають акторами. Покращена та оптимізована реалізація акторів знаходиться у пакеті scala.actors. Проте існує можливість реалізації спрощеного актора – потік, комунікаціонні примітиви якого такі ж самі, як і у поштової скриньки.

**abstract** **class** Actor **extends** Thread **with** MailBox {

**def** act(): Unit

**override** **def** run(): Unit = act()

**def** !(msg: Any) = send(msg)

}

Scala поширює підтримку одночасного виконання у Java додатковими засобами для розподілу роботи між процессорами, а також для координації результатів.

**1.5 Висновки до розділу 1**

1. Був виконаний аналіз організації процесів у різноманітних, сучасних мовах програмування. А саме Java, C#, Python, Scala. Показано, що організація роботи процесів у даних мовах забезпечується вбудованими бібліотеками та модулями. Дані модулі дозволяють створити окремий потік, надати йому пріорітет, і’мя, а також організувати злагоджену роботу між потоками.

2. Виконаний аналіз засобів організації роботи з процесами у мові Java, який показав, що мова має усі необхідні можливості для створення та синхронізації процесів. Незважаючи на те, що не всі механізми синхронізації є вбудованими, засоби мови, такі як : модифікатори volatile, synchronized, набір методів для роботи з потоками тощо, дозволяють реалізувати їх самостійно.

3. Виконаний аналіз засобів організації роботи з процесами у мові С#, який показав, що мова має достатньо засобів для організації паралельних обчислень у програмних додатках. Наприклад, клас Semaphore, клас Mutex, оператор lock, механізм подій та атомарних змінних.

4. Виконаний аналіз засобів організації роботи з процесами у мові Python, який показав, що мова має підтримку паралельних обчислень. Слід зазначити, що існують деякі неоднозначності у організації роботи потоків. Тому необхідно уважно обирати шляхи створення паралельних програм на мові Python.

5. Виконаний аналіз засобів організації роботи з процесами у мові Scala, який показав, що мова має достатню кількість засобів для роботи з процесами та їх синхронізації. Зокрема дана мова має доступ до усіх низькорівневих бібліотек мови Java і підтримує класичні механізми паралельного виконання у стилі Java, такі як : synchronized, wait, notify, notifyAll. А також надає новий рівень абстракції, на основі даних механізмів. Наприклад клас MailBox і бібліотека акторів(Actor), SyncVar.

6. На основі аналізу, що був виконаний у розділі 1можна зробити висновок, що найкращою мовою для організації процесів та роботи з ними є Scala.

**РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС З ОП**

В розділі розглянуто розробку та дослідження програми ПРГ1 для ПКС з СП.

Математична задача: MA=MB\*(MC\*MZ)+min(E)\*MO

Мова програмування: C++

Засоби взаємодії : бібліотека OpenMP

Структуру ПКС з СП представлено на рис. 2.1

1

MA, MB, MZ

2

P-1

P

MO, MC, E

Спільна пам’ять

•••

•

•

Рис. 2.1 Структура ПКС із спільною пам’яттю

* 1. **Розробка паралельного математичного алгоритму**

При розробці паралельного алгоритму передбачається, що розмірність задачі (N) більше або рівне числу процесорів (Р) (N>=P). При чому N кратне P, що дозволяє розбити задачу на окремі частини Н, кожна з яких обробляється своєю задачею (Н = N / P) .

Розрахунок даного виразу можна провести у чотири етапи:

1. MDн = MCн \* MZ;

Загальний ресурс: MZ;

1. a0 = min(Eн);
2. а = min(a, a0);

Загальний ресурс: a;

1. MAн = MBн\*MD+a\*MOн

Загальний ресурс: a, MD;

* 1. **Розробка алгоритмів процесів**

Задача Т1 ТС КД

1. Ввід MB, MZ

2. Сигнал Т2..ТР про завершення вводу S2,1..SP,(P-1)

3. Чекати сигнал від ТР про завершення вводу WP,1

4. Копіювати MZ0 = MZ КД

5. Обчислення MDн = MCн\*MZ

6. Сигнал Т2..ТР про завершення обчислень S2,P..SP,2\*(P-1)

7. Чекати сигнали від Т2..ТР про завершення обчислень W2,2..WP,P

8. Обчислення a0 = min(Eн)

9. Обчислення a = min(a,a0) КД

10. Сигнал Т2..ТР про завершення обчислень S2,2\*P..SP,3\*(P-1)

11. Чекати сигнали від Т2..ТР про завершення обчислень W2,P+1..WP,2\*P

12. Копіювати MD0 = MD, a0 = a КД

13. Обчислення MAн = MBн\*MD0+a0\*MOн

14. Чекати сигнали від Т2..ТР про завершення обчислень W2,2\*P+1..WP,3\*P

15. Вивід MA

Задача Т2 – T(P-1) ТС КД

1. Чекати сигнал від T1, ТР про завершення вводу W1,1, WP,2

2. Копіювати MZ0 = MZ КД

3. Обчислення MDн = MCн\*MZ

4. Сигнал Т1..ТР про завершення обчислень S1,1..SP,P

5. Чекати сигнали від Т1..ТР про завершення обчислень W1,3..WP,P+2

6. Обчислення a0 = min(Eн)

7. Обчислення a = min(a,a0) КД

8. Сигнал Т1..ТР про завершення обчислень S1,P+1..SP,3\*P

9. Чекати сигнали від Т1..ТР про завершення обчислень W1,P+3..WP,2\*P+3

10. Копіювати MD0 = MD, a0 = a КД

11. Обчислення MAн = MBн\*MD0+a0\*MOн

12. Сигнал Т1 про завершення обчислень S1,3\*P+1

Задача ТР ТС КД

1. Ввід MС, MO, E

2. Сигнал Т1..ТP-1 про завершення вводу S1,1..SP-1,(P-1)

3. Чекати сигнал від Т1 про завершення вводу W1,1

4. Копіювати MZ0 = MZ КД

5. Обчислення MDн = MCн\*MZ

6. Сигнал Т1..ТP-1 про завершення обчислень S1,P..SP-1,2\*(P-1)

7. Чекати сигнали від Т1..ТР про завершення обчислень W1,2..WP-1,P

8. Обчислення a0 = min(Eн)

9. Обчислення a = min(a,a0) КД

10. Сигнал Т1..ТР-1 про завершення обчислень S1,2\*P..SP-1,3\*(P-1)

11. Чекати сигнали від Т1..ТР-1 про завершення обчислень W1,P+1..W1,2\*P

12. Копіювати MD0 = MD, a0 = a КД

13. Обчислення MAн = MBн\*MD0+a0\*MOн

14. Сигнал Т1 про завершення обчислень S1,3\*(P-1)+1

* 1. **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.

Окрема лінія на схемі являє собою окрему задачу.

Існує декілька видів задач:

1. Задачі у яких організоване введення вхідних даних
2. Проміжні задачі

Процеси що знаходяться у критичній секції, беруться у окремий блок.

Так як процеси виконують однакові дії, то їх опис винесений у праву частину схеми, а блоки мають спрощений запис.



Рис. 2.2 Схема взаємодії процесів

* 1. **Розробка програми ПРГ1**

Для реалізації алгоритму обчислення на системі зі спільною пам'яттю використана мова програмування С++ та бібліотека OpenMP. Ця бібліотека дає можливість створення потоків, використання механізму критичних секцій та синхронізації потоків за допомогою бар’єрів. Для вирішення задач синхронізації, а саме: синхронізації по вводу, по виводу і по закінченню обчислень, використані бар’єри. Для вирішення задач взаємного виключення доступу до а, MZ, MD використані критичні секції.

Програма складається з одного файлу main.cpp, що містить процедури для вводу, виводу, копіювання, а також головний метод main.

Main містить основний код програми. Блок коду що знаходиться в pragma omp\_parallel є універсальним для кожного процесу.

Синхронізація процесів відбувається за допомогою pragma omp\_barrier.

Вирішення задачі виключення забезпечує pragma omp\_critital(cs), де cs – ім’я критичної секції.

Лістинг програми наведений в додатку А.

Алгоритми програми наведені у додатках Б та В.

* 1. **Тестування програми ПРГ1**

Для замірів часу виконання розробленого програмного забезпечення використовувались функції з бібліотеки time. Програмне забезпечення приймало різне значення обсягів даних та запускалось на різних кількостях ядер. Кількість ядер, які використовуються програмою встановлювалась вручну.

Основні характеристики розробленого програмного забезпечення визначаються за допомогою формул:

Коефіцієнт прискорення:



Коефіцієнт ефективності:

Тестування проводилося на системі з наступними характеристиками:

**Центральний процесор**: Intel Core i5-4210U, 2.6 GHz

**Кількість ядер : 4**

**Оперативна пам’ять:** 8192 Мб

**Операційна система**: Microsoft Windows 7 Ultimate

Таблиця 2.1. Час виконання програми обчислення на системі з СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Р | | | |
| 1 | 2 | 3 | 4 |
| 600 | 3,28 | 1,71 | 1,36 | 0,92 |
| 1200 | 39,91 | 21,37 | 17,24 | 11,68 |
| 1800 | 150,09 | 78,23 | 66,42 | 45,53 |
| 2400 | 389,12 | 220,74 | 174,53 | 124,41 |

Таблиця 2.2. Значення коефіцієнтів прискорення для системи з СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 600 | 1 | 1,918 | 2,412 | 3,565 |
| 1200 | 1 | 1,867 | 2,314 | 3,416 |
| 1800 | 1 | 1,780 | 2,259 | 3,296 |
| 2400 | 1 | 1,763 | 2,229 | 3,127 |

Таблиця 2.3 Значення коефіцієнтів ефективності для системи з СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 600 | 1 | 0,959 | 0,804 | 0,891 |
| 1200 | 1 | 0,934 | 0,771 | 0,854 |
| 1800 | 1 | 0,89 | 0,753 | 0,824 |
| 2400 | 1 | 0,882 | 0,743 | 0,782 |

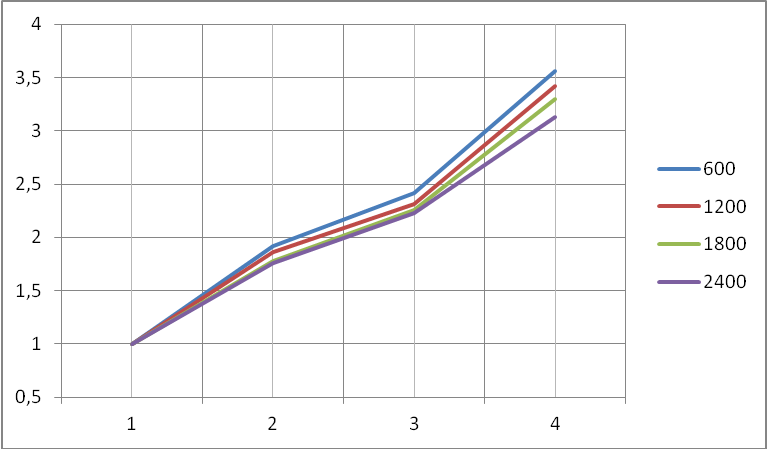


Рис. 2.3. Залежність коефіцієнту прискорення від кількості ЦП для систем з спільною пам’яттю

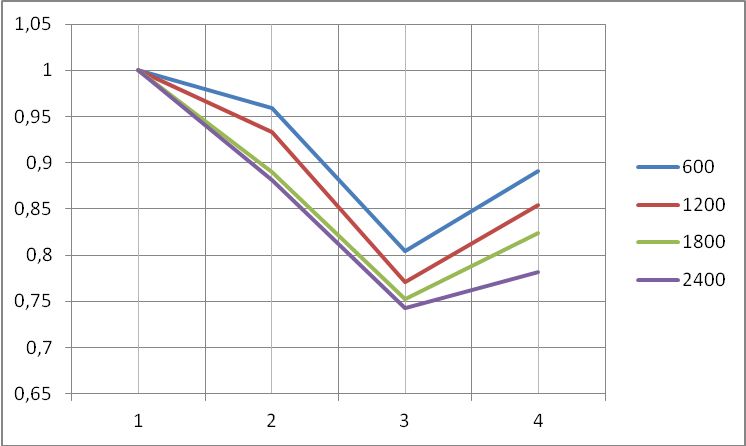


Рис. 2.4**.** Залежність коефіцієнту ефективності від кількості ЦП для системи з спільною пам’яттю

* 1. **Висновки до розділу 2**

1. Бібліотека OpenMP має достатньо механізмів для організації паралельних обчислень, таких як механізм критичних секцій, атомарних змінних, та бар’єрна синхронізація .

2. Коефіцієнт прискорення пропорційно зростає зі збільшенням кількості процесорів. Графік залежності коефіцієнту прискорення від кількості процесорів зображений на рис. 2.2.

3. Коефіцієнт ефективності у загальному випадку має властивість до зменшення із збільшенням кількості процесорів.

4. Загалом процес розробки паралельної програми для системи зі спільною пам’яттю є простим у розробці та реалізації. При аналітичному підході до розв’язання даної задачі, вона зводиться до вирішення двох основних задач паралельного програмування : задачі синхронізації та задачі взаємного виключення.

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

У розділі розглянуто розробку та дослідження програми ПРГ2 для ПКС з ЛП

Математична задача: MA=MB\*(MC\*MZ)+min(E)\*MO

Мова програмування: C++

Засоби взаємодії: бібліотека MPI

Структуру ПКС з ЛП представлено на рис. 3.1

1

MA,MB,MZ

P

MC,MO,E

2

3

•••

•

•

Рис. 3.1 Структура ПКС із локальною пам’яттю

* 1. **Розробка паралельного математичного алгоритму**

При розробці паралельного алгоритму передбачається, що розмірність задачі (N) більше або рівне числу процесорів (Р) (N>=P). При чому N кратне P, що дозволяє розбити задачу на окремі частини Н, кожна з яких обробляється своєю задачею (Н = N / P) .

Розрахунок даного виразу можна провести у чотири етапи:

1. MDн = MCн \* MZ;
2. a0 = min(Eн);
3. а = min(a, a0);
4. MAн = MBн\*MD+a\*MOн
   1. **Розробка алгоритмів процесів**

**Ti** (i=)

1. Якщо rank = 0, то введення MB,MZ
2. Якщо rank = P - 1, то введення MC,MO,E
3. Якщо rank 0 , то прийняти від попереднього MBN\*(N-rank\*H), MZ



1. Якщо rank P-1 , то переслати наступному MBN\*(N-(rank+1)\*H), MZ



1. Якщо rank P-1 , то прийняти від наступного MON\*(rank+1)\*H, MCN\*(rank+1)\*H , E(rank+1)\*H



1. Якщо rank 0 , то переслати попередньому MON\*rank\*H, MCN\*rank\*H , Erank\*H



1. Розрахунок MDн = MCн \* MZ
2. Розрахунок a = min(Eн)
3. Якщо rank P-1 , то прийняти від наступного MDN\*(rank+1)\*H, aTemp



1. Розрахунок a = min(a,aTemp)
2. Якщо rank 0 , то переслати попередньому MDN\*rank\*H, a



1. Якщо rank 0 , то прийняти від попереднього MD, a



1. Якщо rank P-1 , то переслати наступному MD, a



1. Розрахунок MAн = MBн \* MD + a \* MOн
2. Якщо rank P-1 , то прийняти від наступного MAN\*(rank+1)\*H



1. Якщо rank 0 , то переслати попередньому MAN\*rank\*H



1. Якщо rank = 0 , то вивести МА
   1. **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 («крайня зліва»), в якій вводяться дані (*MB, MZ*) та виводяться (MA).



1. Остання задача («крайня зправа»), в якій вводяться дані (*E, MC, MO*).



1. «Проміжна» задача Тk (k=). Кожна з яких містить свою частину вхідних даних та виконує свою частину обчислення. Результат обчислення пересилається згідно зі схемою взаємодії.



Рис. 3.2 Схема взаємодії процесів

* 1. **Розробка програми ПРГ2**

Для реалізації алгоритму обчислення на системі із локальною пам'яттю використана мова програмування С++ та бібліотека MPI. Ця бібліотека дає можливість створення потоків, використання механізму передачі повідомлень. Для передачі даних використовується процедура MPI\_Send, для прийому даних – процедура MPI\_Recv.

MPI\_Recv – функція блокує процес поки не будуть отриманні всі данні.

MPI\_Send – функція блокує процес поки не будуть передані всі дані.

Програма складається з файлу main.cpp, що містить головний метод main. У даному методі відбувається ініціалізація середовища MPI (за допомогою процедури MPI\_Init). Також main містить код, що є універсальним для усіх процесів, та працює згідно алгоритму.

Лістинг програми наведений в додатку Г.

Алгоритм програми наведений у додатках Д, Е.

* 1. **Тестування програми ПРГ2**

Для замірів часу виконання розробленого програмного забезпечення використовувались функції з бібліотеки time. Програмне забезпечення приймало різне значення обсягів даних та запускалось на різних кількостях ядер. Кількість ядер, які використовуються програмою встановлювалась вручну.

Основні характеристики розробленого програмного забезпечення визначаються за допомогою формул:

Коефіцієнт прискорення:



Коефіцієнт ефективності:

Тестування проводилося на системі з наступними характеристиками:

**Центральний процесор**: Intel Core i5-4210U, 2.6 GHz

**Кількість ядер : 4**

**Оперативна пам’ять:** 8192 Мб

**Операційна система**: Microsoft Windows 7 Ultimate

Таблиця 2.4. Час виконання програми обчислення на системі з СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Р | | | |
| 1 | 2 | 3 | 4 |
| 600 | 3,86 | 2,71 | 1,76 | 1,32 |
| 1200 | 39,56 | 28,23 | 18,74 | 14,42 |
| 1800 | 154,03 | 112,31 | 78,42 | 56,76 |
| 2400 | 393,28 | 287,52 | 201,57 | 147,41 |

Таблиця 2.5. Значення коефіцієнтів прискорення для системи з СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 600 | 1 | 1,424354 | 2,193182 | 2,924242 |
| 1200 | 1 | 1,401346 | 2,110993 | 2,743412 |
| 1800 | 1 | 1,371472 | 1,964167 | 2,713707 |
| 2400 | 1 | 1,367835 | 1,951084 | 2,667933 |

Таблиця 2.6 Значення коефіцієнтів ефективності для системи з СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 600 | 1 | 0,712177 | 0,731061 | 0,731061 |
| 1200 | 1 | 0,700673 | 0,703664 | 0,685853 |
| 1800 | 1 | 0,685736 | 0,654722 | 0,678427 |
| 2400 | 1 | 0,683918 | 0,650361 | 0,666983 |

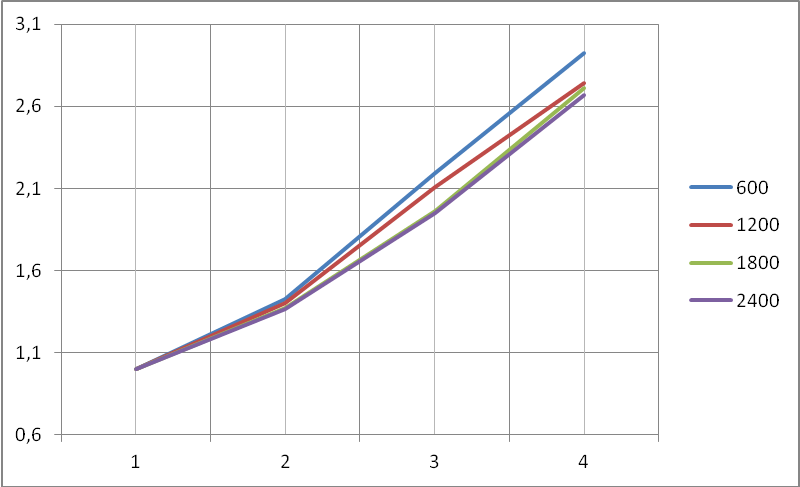


Рис. 3.3. Залежність коефіцієнту прискорення від кількості ЦП для систем з спільною пам’яттю

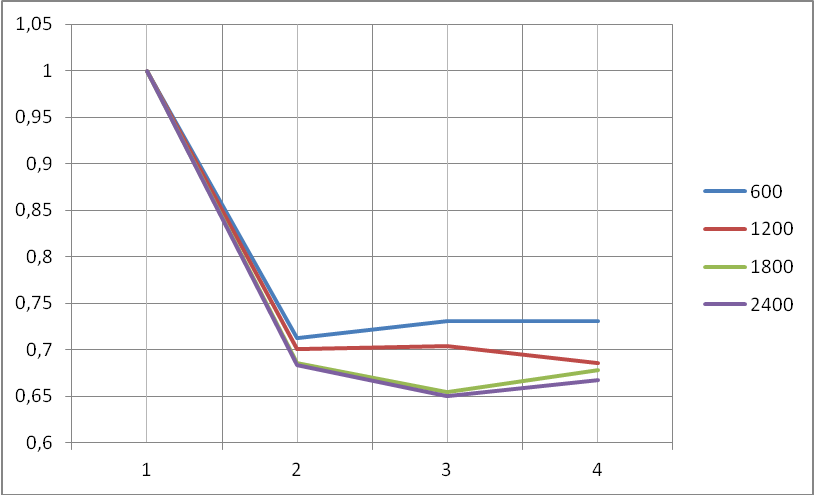


Рис. 3.4**.** Залежність коефіцієнту ефективності від кількості ЦП для системи з спільною пам’яттю

* 1. **Висновки до розділу 2**

1. Бібліотека MPI має достатньо механізмів для організації паралельних обчислень, а саме : створення процесів, та організацію пересилки повідомлень між ними.

2. Коефіцієнт прискорення пропорційно зростає зі збільшенням кількості процесорів. Графік залежності коефіцієнту прискорення від кількості процесорів зображений на рис. 3.3. ? АНАЛИЗ ЕГО ПОВЕДЕНИЯ???

3. Коефіцієнт ефективності у загальному випадку має властивість до зменшення із збільшенням кількості процесорів. Графік залежності коефіцієнту ефективності від кількості процесорів зображений на рис. 3.4.

4. Загалом процес розробки паралельної програми для системи з локальною пам’яттю є досить складним у розробці та реалізації. При аналітичному підході до розв’язання даної задачі, основним завданням є правильна організація пересилки повідомлень між процесами.

**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1) Бернакевич І.Є., Вагін П.П. Програмування мовою Java: використання фундаментальних класів: Тексти лекцій. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2002. – 48 с.

2) Корочкін О.В., Жуков І.А. Паралельні та розподілені обчислення.Навч. посіб. – 2-ге вид. виправл. І доп. К.: «Корнійчук», 2014. – 284 с.

3) C#. [Електронний ресурс] . Режим доступу: <https://uk.wikipedia.org/wiki/C_Sharp>

4) Класс Thread.C#. [Електронний ресурс] . Режим доступу: <https://msdn.microsoft.com/ru-ru/library/system.threading.thread%28v=vs.110%29.aspx>

5) Многопоточные приложения C#. [Електронний ресурс] . Режим доступу: <http://jakeroid.com/mnogopotochny-e-prilozheniya-v-c-dlya-chajnikov.html>

6) Мониторы. C#. [Електронний ресурс] . Режим доступу:

<https://msdn.microsoft.com/ru-ru/library/hf5de04k.aspx>

7) Синхронизация потоков в Python. [Електронний ресурс] . Режим доступу: <http://www.quizful.net/post/thread-synchronization-in-python>

8) Mutex. Python. [Електронний ресурс] . Режим доступу: <https://docs.python.org/2/library/mutex.html>

9) Паралельность в Scala. [Електронний ресурс] . Режим доступу: <https://twitter.github.io/scala_school/ru/concurrency.html>

**ДОДАТОК А**.

Лістінг ПРГ1

// Курсова робота(ЗП)

// MA=MB\*(MC\*MZ)+min(E)\*MO

// Сагач Ярослав Ігорович

// ІО-22

// 22.03.2015

#include <iostream>

#include <omp.h>

#include <Windows.h>

#include <fstream>

#include <ctime>

using namespace std;

int const N = 1200;

int const P = 4;

int const H = N/P;

int\*\* MD; int\*\* MA;int\*\* MB; int\*\* MZ; int\*\* MC; int\*\* MO;

int\* E;

int a = INT\_MAX;

int\* InputVector() {

int\* result = new int[N];

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

result[i] = 1;

}

return result;

}

int\* CopyVector(int\* vector) {

int\* result = new int[N];

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

result[i] = vector[i];

}

return result;

}

int\*\* InputMatrix() {

int\*\* result = new int\*[N];

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

result[i] = new int[N];

}

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

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

result[i][j] = 1;

}

}

return result;

}

int\*\* CopyMatrix(int\*\* matrix) {

int\*\* result = new int\*[N];

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

result[i] = new int[N];

}

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

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

result[i][j] = matrix[i][j];

}

}

return result;

}

void OutputVector(int\* vector) {

if (N<10){

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

int val = vector[i];

cout<<val<<" ";

}

cout<<"\n";}

else cout<<vector[0]<<"\n";

}

void OutputMatrix(int\*\* matrix) {

if (N<10){

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

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

int val = matrix[i][j];

cout<<val<<" ";

}

cout<<"\n";

}}

else cout<<matrix[0][0];

}

int main()

{

cout<<"Main thread started \n";

int\*\* MD = new int\*[N];

int\*\* MA = new int\*[N];

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

MD[i] = new int[N];

MA[i] = new int[N];

}

int ts = clock();

omp\_set\_num\_threads(P);

#pragma omp parallel

{

int Tid = omp\_get\_thread\_num();

cout<<"Thread "<<Tid+1<<" started \n";

// Ввод данных

switch (Tid){

case 0:

MB = InputMatrix();

MZ = InputMatrix();

break;

case (P-1):

MC = InputMatrix();

MO = InputMatrix();

E = InputVector();

break;

}

// Ждать завершения ввода в остальных потоках

#pragma omp barrier

// Копия MZ0 = MZ

int\*\* MZ0;

#pragma omp critical(cs1)

{

MZ0 = CopyMatrix(MZ);

}

// Счет MDн = MCн \* MZ0

for(int i = Tid\*H;i<(Tid+1)\*H;i++){

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

int z1=0;

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

z1 = z1+MC[i][k]\*MZ0[k][j];

}

MD[i][j] = z1;

}

}

// Ждать завершения счета в остальных потоках

#pragma omp barrier

// Счет a0 = min(Eн)

int a0 = INT\_MAX;

for(int i = Tid\*H;i<(Tid+1)\*H;i++){

if (E[i]<a0) a0 = E[i];

}

// Счет a = min(a,ai)

#pragma omp critical(cs2)

{

if (a0<a) a = a0;

}

// Ждать завершения счета в остальных потоках

#pragma omp barrier

// Копия a0 = a, MD0 = MD

int\*\* MD0 = new int\*[N];

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

MD0[i] = new int[N];

}

#pragma omp critical(cs3)

{

MD0 = CopyMatrix(MD);

a0 = a;

}

// Счет MAн=MBн\*MD0+a0\*MOн

for(int i = Tid\*H;i<(Tid+1)\*H;i++){

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

int z1=0;

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

z1 = z1+MB[i][k]\*MD0[k][j];

}

MA[i][j] = z1 + a0\*MO[i][j];

}

}

// Ждать завершения счета в остальных потоках

#pragma omp barrier

// Вывод А

if (Tid==0){

OutputMatrix(MA);

}

cout<<"Thread "<<Tid+1<<" finished \n";

}

int tf = clock();

cout<<"Time: "<<(tf-ts)/1000.0;

getchar();

return 0;

}

ДОДАТОК Б.



ДОДАТОК Г.

// Курсова робота(ЛП)

// MA=MB\*(MC\*MZ)+min(E)\*MO

// Сагач Ярослав Ігорович

// ІО-22

// 22.04.2015

#include <iostream>

#include "mpi.h"

#include <Windows.h>

#include <ctime>

using namespace std;

int N = 600;

int P = 2;

int H = N/P;

void OutputMatrix(int\* matrix) {

if (N<10){

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

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

int val = matrix[i\*N+j];

cout<<val<<" ";

}

cout<<"\n";

}}

else cout<<matrix[0];

}

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

{

int ts = clock();

MPI\_Init(&args,&argv);

int rank;

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

int number;

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

P = number;

H = N/P;

int rankOfMiddle;

int tid = 0;

MPI\_Status sts;

// Розподіл розмірів

int\* MB = (int\*)malloc(N\*N\*sizeof(int));

int\* MC = (int\*)malloc(N\*N\*sizeof(int));

int\* MA = (int\*)malloc(N\*N\*sizeof(int));

int\* MZ = (int\*)malloc(N\*N\*sizeof(int));

int\* MO = (int\*)malloc(N\*N\*sizeof(int));

int\* MD = (int\*)malloc(N\*N\*sizeof(int));

int\* E = (int\*)malloc(N\*sizeof(int));

int a = 0;

for(int i =0; i<N\*N;i++){

MD[i]=0;

MA[i]=0;}

if (P==1){

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

E[i] = 1;

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

MB[i\*N+j] = 1;

MZ[i\*N+j] = 1;

MC[i\*N+j] = 1;

MO[i\*N+j] = 1;

}

}

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

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

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

MD[i\*N+j] = MD[i\*N+j] + MC[i\*N+k] \* MZ[k\*N+j];

}

}

}

//min

int min = INT\_MAX;

for (int i=0;i<H;i++){

if (E[i+rank\*H]<min) min = E[i+rank\*H];

}

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

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

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

MA[i\*N+j] = MA[i\*N+j] + MB[i\*N+k] \* MD[k\*N+j];

}

MA[i\*N+j] = MA[i\*N+j] + min \* MO[i\*N+j];

}

}

OutputMatrix(MA);

}

//----------------------------------------------------------//

//----------------------------------------------------------//

else {

if (rank==0){

// Введення MB , MZ

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

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

MB[i\*N+j] = 1;

MZ[i\*N+j] = 1;

}

}

//MPI\_Send(&MB[N], N\*(N-H), MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD);

//MPI\_Send(&MZ[0], N\*N, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD);

}

else if (rank==P-1){

//Введення E , MC , MO

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

E[i] = 1;

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

MC[i\*N+j] = 1;

MO[i\*N+j] = 1;

}

}

}

if (rank!=0){

MPI\_Recv(&MB[rank\*H\*N],N\*(N-rank\*H), MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD, &sts);

MPI\_Recv(MZ,N\*N, MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD, &sts);

}

if(rank != P-1){

MPI\_Send(&MB[rank\*H\*N], (N-(rank+1)\*H)\*N, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD);

MPI\_Send(&MZ[0], N\*N, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD);

}

//Пересилка даних з Р-1

if(rank != P-1){

MPI\_Recv(MO, ((rank+1)\*H)\*N, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD, &sts);

MPI\_Recv(MC, ((rank+1)\*H)\*N, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD, &sts);

MPI\_Recv(E, ((rank+1)\*H), MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD, &sts);

}

if(rank != 0){

MPI\_Send(&MO[0], ((rank)\*H)\*N, MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD);

MPI\_Send(&MC[0], ((rank)\*H)\*N, MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD);

MPI\_Send(&E[0], ((rank)\*H), MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD);

}

//Счет MDн = MCн \* MZ

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

{

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

{

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

{

MD[rank\*H\*N+i\*N+j] = MD[rank\*H\*N+i\*N+j] + MC[rank\*H\*N+i\*N+k] \* MZ[k\*N+j];

}

}

}

//min

int min = INT\_MAX;

for (int i=0;i<H;i++){

if (E[i+rank\*H]<min) min = E[i+rank\*H];

}

a = min;

//Пересилка результату від Р до 1

if(rank != P-1){

MPI\_Recv(&MD[(rank+1)\*H\*N], (N-(rank+1)\*H)\*N, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD, &sts);

int aTemp = 0;

MPI\_Recv(&aTemp, 1, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD, &sts);

if (aTemp<a) a = aTemp;

}

if(rank != 0){

MPI\_Send(&MD[rank\*H\*N], (N-(rank)\*H)\*N, MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD);

MPI\_Send(&a, 1, MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD);

}

//Пересилка результату від 1 до Р

if (rank!=0){

MPI\_Recv(&a,1, MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD, &sts);

MPI\_Recv(MD,N\*N, MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD, &sts);

}

if(rank != P-1){

MPI\_Send(&a, 1, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD);

MPI\_Send(&MD[0], N\*N, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD);

}

// Cчет MAн = MBн \* MD + a \* MOн

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

{

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

{

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

{

MA[rank\*H\*N+i\*N+j] = MA[rank\*H\*N+i\*N+j] + MB[rank\*H\*N+i\*N+k] \* MD[k\*N+j];

}

MA[rank\*H\*N+i\*N+j] = MA[rank\*H\*N+i\*N+j] + a \* MO[rank\*H\*N+i\*N+j];

}

}

// Пересилка результату (МА)

if(rank != P-1){

MPI\_Recv(&MA[(rank+1)\*H\*N], (N-(rank+1)\*H)\*N, MPI\_INT, rank+1, tid, MPI\_COMM\_WORLD, &sts);

}

if(rank != 0){

MPI\_Send(&MA[rank\*H\*N], (N-(rank)\*H)\*N, MPI\_INT, rank-1, tid, MPI\_COMM\_WORLD);

}

if (rank==0) OutputMatrix(MA);

}

MPI\_Finalize();

int tf = clock();

cout<<(ts-tf)/1000.0;

//getchar();

return 0;

}

ДОДАТОК Д.

