НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

\_\_\_\_\_\_\_\_\_\_\_\_\_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-52

спеціальності

123 «Комп’ютерна інженерія»

Бояршина Ігора Івановича

(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 - 2018 рік

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет (інститут) інформатики та обчислювальної техніки

(повна назва)

Кафедра обчислювальної техніки

(повна назва)

Освітньо-кваліфікаційний рівень бакалавр

Напрям підготовки 123 «Комп’ютерна інженерія»

*(шифр і назва****)***

***З А В Д А Н Н Я***

НА КУРСОВУ РОБОТУ СТУДЕНТУ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *Бояршину Ігору Івановичу*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(прізвище, ім’я, по-батькові)*

1. Тема роботи «Порівняння реалізації механізму атомік - змінних секції в

мовах і бібліотеках паралельного програмування»

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

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

2. Строк подання студентом роботи 11 травня 2018 р.

3. Вхідні дані до роботи

- засоби роботи з процесами в бібліотеці WinAPI

- математична задача A = (B\*C)\*X + e\*S\*(MR\*MT) - Z

- структури ПКС ОП та ПКС ЛП

- мови і бібліотеки програмування: WinAPI

- засоби організації взаємодії процесів: семафори, мютекси, події, критичні секції

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

- огляд засобів роботи в мовах і бібліотеках паралельного програмування

- розробка і тестування програми ПРГ1 для ПКС ОП

- розробка і тестування програми ПРГ2 для ПКС ЛП

5. Перелік графічного матеріалу

- структурна схема ПКС СП

- структурна схема ПКС ЛП

- схеми алгоритмів процесів і головної програми для ПРГ1

- схеми алгоритмів процесів і головної програми для ПРГ2.

6. Дата видачі завдання \_\_\_\_\_\_\_\_\_12 02 2018\_\_\_\_\_\_\_\_\_\_\_.

***КАЛЕНДАРНИЙ ПЛАН***

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строк виконання етапів КР |
| 1 | Виконання розділу 1 | 1.03.2018 |
| 2 | Виконання розділу 2 | 23.03.2018 |
| 3 | Виконання розділу 3 | 23.04.2018 |
| 4 | Оформлення КР | 10.05.2018 |
| 5 | Перевірка КР викладачем | 17.05.2018 |
| 6 | Захист КР | 18.05.2018 |

**Студент \_\_\_\_\_\_\_\_\_ \_\_\_\_** Бояршин І.І.

( підпис ) (прізвище та ініціали)

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

( підпис ) (прізвище та ініціали)

ЗМІСТ

[ВСТУП 5](#_Toc509625462)

[РОЗДІЛ 1. ПОРІВНЯННЯ РЕАЛІЗАЦІЇ МЕХАНІЗМУ АТОМІК-ЗМІННИХ СЕКЦІЙ В МОВАХ І БІБЛІОТЕКАХ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ 6](#_Toc509625463)

[1.1 Атомарні змінні в мові програмування Ада 7](#_Toc509625464)

[1.2 Атомарні змінні в мові програмування *Java* 7](#_Toc509625465)

[1.3 Атомарні змінні в мові програмування *C#* 11](#_Toc509625466)

[1.4 Атомарні змінні в бібліотеці *OpenMP* 15](#_Toc509625467)

[1.5 Атомарні змінні в бібліотеці *Win32* 16](#_Toc509625468)

[1.6 Порівняння реалізації механізму атомік-змінних в різних мовах програмування та бібліотеках паралельного програмування 17](#_Toc509625469)

[1.7 Висновки до розділу 1 18](#_Toc509625470)

[РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС СП 20](#_Toc509625471)

[2.1 Розробка паралельного математичного алгоритму 21](#_Toc509625472)

[2.2 Розробка алгоритмів процесів 21](#_Toc509625473)

[2.3 Розробка схеми взаємодії процесів 23](#_Toc509625474)

[2.4 Розробка програми ПРГ1 26](#_Toc509625475)

[2.5 Тестування програми ПРГ1 27](#_Toc509625476)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 31](#_Toc509625477)

# ВСТУП

Практична потреба написання та використання паралельних програм виникла тоді, коли обчислювальні потужності комп’ютера перевищили можливості людини, тобто коли людина почала бути однією з найповільніших ланок в роботі комп’ютера. З тих пір з метою подальшого пришвидшення роботи та збільшення ефективності системи стали з’являтися багатоядерні обчислювальні машини. Це в свою чергу вимагало створення нових підходів в написанні програм, відмінних від звичайних (послідовних) програм. Так з’явилися паралельні програми. Такі програми потребували більшого часу та зусиль для їх написання, адже тепер потрібно було враховувати взаємодію процесів між собою. Саме на створення коректної моделі та архітектури взаємодії і йде більша частина зусиль. Проте виграш значний: в ідеальному випадку система з *N* процесів на *N* ядерній системі дозволяє вирішити задачу в *N* разів швидше, порівнюючи з аналогічною послідовною програмою. Саме тому потреба в програмах для паралельних систем найближчим часом нікуди не зникне і залишиться актуальною.

Були створені різні механізми — від низькорівневих до високорівневих, що дозволяють забезпечити правильну взаємодію процесів між собою. Про один з цих механізмів піде мова в першому розділі.

У першому розділі розглядається один з основних інструментів в паралельному програмуванні — атомік-змінні. Наводиться опис реалізації цього механізму в різних мовах та бібліотеках паралельного програмування з використанням прикладів. Порівнюються різні реалізації та виділяються спільні підходи.

У другому розділі наводиться розробка програми для паралельної комп'ютерної системи зі спільною пам'яттю, що обчислює задану математичну задачу. В якості інструменту використовується бібліотека паралельного програмування WinAPI. Виконується тестування програми та розраховуються коефіцієнти.

У третьому розділі ...

# РОЗДІЛ 1. ПОРІВНЯННЯ РЕАЛІЗАЦІЇ МЕХАНІЗМУ АТОМІК-ЗМІННИХ СЕКЦІЙ В МОВАХ І БІБЛІОТЕКАХ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ

Під час написання програм для паралельних комп’ютерних систем рано чи пізно виникає необхідність організувати правильну взаємодію процесів між собою. Існує багато можливих варіантів взаємодій процесів, проте можна виділити дві основні фундаментальні задачі, що виникають в паралельному програмуванні: задача взаємного виключення та задача синхронізації процесів.

Атомік-змінні (від греч. *atomos* — неподільне) — змінні, що реалізують в собі механізм атомарної операцій при роботі з ними, таким чином дозволяючи вирішити задачу взаємного виключення. Атомік-змінні ще називають “атомарними” або “неподільними”. Атомарна операція — така операція, що не переривається, та або виконується повністю, або не виконується взагалі. Атомарна операція не може залишитися в “підвішеному” стані. У випадку змінних розглядаються операції зчитування змінної та її запис, а атомарність змінних означає, що для будь-якого “зовнішнього спостерігача” кожна з цих операцій виконується як єдине ціле і не переривається.

Прикладом ситуації, в якій явно виникає необхідність подібної “гарантії”, може слугувати наступне: в той час, як один процес А зчитує значення змінної, інший процес Б починає змінювати її значення. Якщо не має місце гарантія атомарності операції зчитування та запису для змінної, що виступає в даному прикладі спільним ресурсом, то значення змінної, що було зчитане процесом А, може бути невірним в результаті її зміни потоком Б посеред операції зчитування.

Суть механізму атомарних змінних полягає в тому, що мова програмування або бібліотека для паралельного програмування дозволяють оголосити змінну як атомарну, після чого подальші операції з нею будуть автоматично мати в собі механізм взаємного виключення, звільняючи програміста від необхідності прописувати його кожен раз власноруч. Кожне звернення до атомарної змінної трактується системою як критична ділянка, і система самостійно контролює вхід та вихід з нього [1]. Ще однією перевагою використання атомарних змінних є те, що механізм взаємного виключення в них може бути реалізований через не блокуючі (*non-blocking*) операції, тим самим підвищуючи ефективність роботи з ними.

## **1.1 Атомарні змінні в мові програмування Ада**

В мові програмування Ада механізм атомарних змінних реалізується за допомогою прагм (*pragmas*). Спочатку змінна оголошується, як будь-яка інша змінна, а потім до неї застосовується спеціальна прагма. Для вказання компілятору, що змінна має бути атомарною, потрібно для неї використати одну з чотирьох можливих прагм:

• Pragma Atomic(Змінна);

• Pragma Atomic\_Component(Змінна);

• Pragma Volatile(Змінна);

• Pragma Volatile\_Component(Змінна).

Прагма Atomic описує змінну як атомарну.

Прагма Volatileпрацює аналогічно, проте є більш строгою. Це означає, що операції над нею мають виконуватися безпосередньо в оперативній пам’яті, тобто без використання кешу чи регістрів.

Прагми Atomic та Volatile використовуються зі змінними простого типу. Для задання атомарних змінних складного типу (масиви та записи) необхідно використати аналоги цих прагм з приставкою Component.

Приклад оголошення атомарних змінних в Аді наведено в прикладі 1.1.

**Приклад 1.1.** Атомарні змінні в мові програмування Ада

-- Оголошення змінних

Counter : Integer;

Vector : array(1..10) of Integer;

-- Використання прагми Atomic для змінної простого типу

Pragma Atomic(Counter);

-- Використання прагми Atomic\_Component для змінної складного типу

Pragma Atomic\_Component(Vector);

## **1.2 Атомарні змінні в мові програмування *Java***

У мові програмування *Java* існує декілька варіантів атомарних змінних.

Перший варіант — з використанням ключового слова volatile. Написання ключового слова volatile при оголошенні змінної в *Java* дає наступний ефект: кожна операція читання чи запису даної змінної буде атомарною, причому не блокуючою (*non-blocking*). При цьому змінна буде зберігатися в основній пам’яті, і як наслідок, кожен процес завжди буде мати доступ до останнього, найновішого значення цієї змінної. Приклад використання *volatile*-змінної в *Java* наведено в прикладі 1.2.

**Приклад 1.2.** *Volatile*-змінна в мові програмування *Java*

public class ChildClass implements Runnable {

// оголошення volatile-змінної

public volatile boolean terminated = false;

public void run() {

// поки не отримали сигналу до завершення,

while (!terminated) {

doStuff(); // продовжуємо працювати

}

}

public void duStuff() {

// Якась важлива робота

}

}

В наведеному вище прикладі метод run() буде продовжувати свою роботу, доки його не зупинить встановлення змінної terminated в значення true іншим потоком зовні. При цьому гарантовано, що перший потік “побачить” зміну terminated, так як вона оголошена як volatile.

Проте використання ключового слова volatile іноді виявляється недостатнім. Наприклад, коли потрібно гарантувати атомарність операції інкременту. У прикладі 1.3 наведено такий фрагмент коду. В цьому випадку не гарантується атомарність інкременту, тому що ключове слово volatile лише гарантує атомарність зчитування та запису змінної, проте ніщо не заважає іншому потоку змінити значення змінної поки буде відбуватися операція додавання [2].

**Приклад 1.3.** Не атомарний інкремент в мові програмування *Java*

class JavaExample1 {

// в даному випадку volatile не дає бажаного результату

public volatile int x = 0;

public void inc() {

x = x + 1; // не гарантується атомарність інкременту

}

}

Для того щоб забезпечити атомарність змінних та деяких операцій над ними, в *Java* існує ряд спеціальних класів, які можна знайти в пакеті java.util.concurrent.atomic [3]. Перелік класів в цьому пакеті наведено в таблиці 1.1.

Таблиця 1.1. Склад пакету java.util.concurrent.atomic

|  |  |
| --- | --- |
| Назва класу | Короткий опис |
| AtomicBoolean | Булева змінна, що може бути оновлена атомарно. |
| AtomicInteger | Змінна цілого типу, що може оновлена атомарно. |
| AtomicIntegerArray | Масив з елементів цілого типу, що можуть бути оновлені атомарно. |
| AtomicIntegerFieldUpdater<T> | Оснований на рефлексії (reflection) клас, що дозволяє атомарне оновлення призначених volatile int полів призначених класів. |
| AtomicLong | Змінна цілого довгого типу, що може бути оновлена атомарно |
| AtomicLongArray | Масив з елементів цілого довгого типу, елементи якого можуть бути оновлені атомарно. |
| AtomicLongFieldUpdater<T> | Оснований на рефлексії (reflection) клас, що дозволяє атомарне оновлення призначених volatile long полів призначених класів. |
| AtomicMarkableReference<V> | Зберігає посилання на об’єкт, а також біт маркування, що може бути оновлений атомарно. |
| AtomicReference<V> | Посилання на об’єкт, що може бути оновлена атомарно. |
| AtomicReferenceArray<E> | Масив з посилань на об’єкти, що можуть бути оновлені атомарно. |
| AtomicReferenceFieldUpdater<T,V> | Оснований на рефлексії (reflection) клас, що дозволяє атомарне оновлення призначених volatile посилань призначених класів. |
| AtomicStampedReference<V> | Зберігає посилання на об’єкт, а також цілочислову “печатку”, що може бути оновлена атомарно. |
| DoubleAccumulator | Одна чи декілька змінних, що разом утримують double змінну, що оновлюється за допомогою заданої функції. |
| DoubleAdder | Одна чи декілька змінних, що разом утримують double суму, початкове значення якої — 0. |
| LongAccumulator | Одна чи декілька змінних, що разом утримують long змінну, що оновлюється за допомогою заданої функції. |
| LongAdder | Одна чи декілька змінних, що разом утримують long суму, початкове значення якої — 0. |

В прикладі 1.4 наведено лістинг програми, що правильно реалізує атомарний інкремент за допомогою спеціальних класів з пакету java.util.concurrent.atomic.

**Приклад 1.4.** Атомарний інкремент в мові програмування *Java*

class JavaExample2 {

// Спеціальний клас для атомарного цілочисельного типу

public AtomicInteger x = new AtomicInteger(0);

public void inc() {

x.getAndIncrement(); // гарантується атомарність інкременту

}

}

Класи з пакету java.util.concurrent.atomic реалізовані на машинному рівні, що робить їх швидкими та ефективними в порівнянні з використанням звичайних механізмів для вирішення задачі взаємного виключення. Використання методів цих класів є не блокуючим (*non-blocking*).

## **1.3 Атомарні змінні в мові програмування *C#***

Атомарні змінні в мові програмування *C#* реалізуються за допомогою методів класу Interlocked.

Клас Interlocked надає методи для роботи зі змінною, що є спільним ресурсом для декількох процесів. Методи цього класу є атомарними. На сучасних процесорах операції класу Interlocker можуть були реалізовані однією інструкцією процесора, таким чином надаючи дуже ефективний механізм для вирішення задачі взаємного виключення.

Розглянемо детальніше основні методи з цього класу:

• Методі Increment та Decrement інкрементують чи декрементують змінну та зберігають результат однією операцією. Звичайна реалізація такої операції складалася б з таких етапів: завантажити значення змінної до регістру, інкрементувати чи декрементувати значення регістру, записати нове значення з регістру до змінної. Якщо не використовувати спеціальні методи Increment та Decrement, поточний процес може бути переведений до стану. Готовності, а його місце займе другий процес, і при цьому виконає всі три операції. Коли після цього перший процес продовжить свою роботу, в змінну запишеться неправильне значення, нехтуючи роботою другого процесу.

• Метод Exchange атомарно міняє місцями значення переданих змінних.

• Метод CompareExchange комбінує дві операції: порівняння двох змінних та зберігання значення третьої змінної в одній з перших двох змінних. В залежності від результату порівняння. Ця комбінація виконується як атомарна операція.

В таблиці 1.2 наведено повний перелік методів класуInterlocked [5].

Таблиця 1.2. Склад пакету java.util.concurrent.atomic

|  |  |
| --- | --- |
| Назва методу | Опис роботи |
| Add(Int32, Int32) | Додає два 32-бітних числа та замінює перше число їх сумою. Атомарна операція. |
| Add(Int64, Int64) | Додає два 64-бітних числа та замінює перше число їх сумою. Атомарна операція. |
| CompareExchange(Double, Double, Double) | Порівнює два Double числа з плаваючою комою та, якщо вони рівні, замінює перше значення. |
| CompareExchange(Int32, Int32, Int32) | Порівнює два Int32 числа та, якщо вони рівні, замінює перше значення. |
| CompareExchange(Int64, Int64, Int64) | Порівнює два Int64 числа та, якщо вони рівні, замінює перше значення. |
| CompareExchange(IntPtr, IntPtr, IntPtr) | Порівнює два платформо-залежних посилання числа та, якщо вони рівні, замінює перше значення. |
| CompareExchange(Object, Object, Object) | Порівнює два об’экта на рівність посилання та, якщо вони рівні, замінює перше значення. |
| CompareExchange(Single, Single, Single) | Порівнює два Single числа з плаваючою комою та, якщо вони рівні, замінює перше значення. |
| CompareExchange<T>(T, T, T) | Порівнює два посилання типу *Т* та, якщо вони рівні, замінює перше значення. |
| Decrement(Int32) | Зменшує значення вказаної змінної типу Int32 на один (робить декремент) та зберігає результат. Атомарна операція. |
| Decrement(Int64) | Зменшує значення вказаної змінної типу Int64 на один (робить декремент) та зберігає результат. Атомарна операція. |
| Exchange(Double, Double) | Встановлює Double число з плаваючою комою вказаній змінній та повертає попереднє значення. Атомарна операція. |
| Exchange(Int32, Int32) | Встановлює Int32 число вказаній змінній та повертає попереднє значення. Атомарна операція. |
| Exchange(Int64, Int64) | Встановлює Int64 число вказаній змінній та повертає попереднє значення. Атомарна операція. |
| Exchange(IntPtr, IntPtr) | Встановлює платформо-залежне посилання вказаній змінній та повертає попереднє значення. Атомарна операція. |
| Exchange(Object, Object) | Встановлює об’єкт вказаній змінній та повертає попередній об’єкт. Атомарна операція. |
| Exchange(Single, Single) | Встановлює Single число з плаваючою комою вказаній змінній та повертає попереднє значення. Атомарна операція. |
| Exchange<T>(T, T) | Встановлює змінну типу *Т* вказаній змінній та повертає попереднє значення. Атомарна операція. |
| Increment(Int32) | Збільшує значення вказаної змінної типу Int32 на один (робить інкремент) та зберігає результат. Атомарна операція. |
| Increment(Int64) | Збільшує значення вказаної змінної типу Int64 на один (робить інкремент) та зберігає результат. Атомарна операція. |
| MemoryBarrier() | Синхронізує доступ до пам’яті наступним чином: процесор, що виконує поточний процес, не може міняти місцями інструкції таким чином, що доступи до пам’яті, що відбуваються до виклику MemoryBarrier виконуються після доступів до пам’яті, що слідують за викликомMemoryBarrier. |
| Read(Int64) | Повертає Int64 значення, що було завантажено атомарною операцією. |

В прикладі 1.5 наведено лістинг програми на *С#*, що використовує операції з класу Interlocked.

**Приклад 1.5.** Атомарний інкремент в мові програмування *Java*

public class CSharpExample {

// totalValue може бути оновлено декількома процесами

private double totalValue = 0;

public double Total { get {return totalValue;} }

public double AddToTotal(double value) {

double initial, computed;

do {

initial = computed;

computed = initial + value;

// якщо initial не рівне total, то інший процес встиг змінити

// значення с тих пір, як почався цикл. Total не оновлюється,

// і цикл повториться знову.

} while (initial != Interlocked.CompareExchange(

ref totalValue, computed, initial));

return computed;

}

}

*C#* також підтримує використання ключового слова volatile, що працює так само, як і в мові програмування *Java*. Як і в *Java*, коректне застосування ключового слова volatile буде тільки тоді, коли змінна буде використана лише для атомарного запису та зчитування. Інакше потрібно використовувати методи класу Interlocked [6].

## **1.4** **Атомарні змінні в бібліотеці *OpenMP***

В бібліотеці *OpenMP* механізм атомарних змінних реалізується за допомогою директиви omp atomic [7]. Ця директива дозволяє безпечно змінити значення змінної, що слугує спільним ресурсом для декількох процесів, за допомогою апаратних засобів системи, що підтримують використання атомарних змінних.

Директива omp atomic застосовується лише до безпосередньо наступної за нею операції присвоєння. При цьому атомарною стає тільки змінна, що знаходиться по ліву сторону від оператора присвоєння. Це означає, що змінні, що знаходяться по праву сторону від оператора присвоєння, атомарними не будуть.

Для мови *С++* потрібно використати наступний синтаксис директиви:

#pragma omp atomic

Для мови *Fortran* потрібно використати наступний синтаксис директиви:

!$omp atomic

В прикладі 1.6 наведено фрагмент програми на мові *С++*, що використовує прагму atomic бібліотеки *OpenMP* [8].

**Приклад 1.6.** Прагма atomic в бібліотеці *OpenMP*

#define N 5

int main () {

int numberOfThreads = 0;

#pragma omp parallel num\_threads(N)

{

#pragma omp atomic

numberOfThreads++;

}

std::cout << “Threads: “ << numberOfThreads << std::endl;

}

## **1.5 Атомарні змінні в бібліотеці *Win32***

Механізм роботи з атомарними змінними в бібліотеці *Win32* реалізується за допомогою спеціальних функцій. Вони дозволяють легко працювати зі змінними, що є спільним ресурсом для декількох процесів. Гарантується, що ці операції будуть виконані атомарно.

Далі наведений перелік основних функцій [9]:

• InterlockedIncrement та InterlockedDecrement поєднують у собі етапи, необхідні для інкрементування чи декрементування змінної в атомарному стилі. При цьому функції повертають нове значення змінної;

• InterlockedExchange та InterlockedExchangePointer атомарно міняють місцями значення переданих змінних. Функція InterlockedExchangeAdd поєднує у собі дві операції: вона додає дві змінні та зберігає результат в одній з них;

• Функції InterlockedCompareExchange, InterlockedCompare64Exchange128 та InterlockedCompareExchangePointer роблять наступне: вони порівнюють значення двох змінних та зберігають результат в одній з них, в залежності від результату порівняння;

• InterlockedAnd, InterlockedOr та InterlockedXor атомарно виконують відповідні побітові операції;

• Деякі функції були створені спеціально для роботи з 64-бітними змінними та адресами, та є оптимізованими для використання у 64-бітних версіях операційної системи *Windows*. Кожна така функція має “64” у своїй назві. Прикладами можуть слугувати InterlockedDecrement64 та InterlockedCOmpareExchangeAcquire64;

Також існують функції, що комбінують в собі базові Interlocked функції для роботи зі змінними з операціями по захвату та звільненню пам’яті, що їх підтримують деякі процесори. Такі функції мають у своїй назві “*Acquire*” чи “*Release*”. Наприклад, InterlockedDecrementAcquire та InterlockedDecrementRelease. Перша з функцій вказує, що операції по роботі з пам’яттю, що виконуються поточним процесом будуть видимі до того, як почнуть виконуватимуться будь-які інші операції з пам’яттю. Друга функція вказує, що операції по роботі з пам’яттю, що виконуються поточним процесом будуть видимі після того, як завершаться всі операції з пам’яттю. За допомогою цих функцій можна домогтися результату, щоб операції з пам’яттю виконувалися у певному порядку.

## **1.6 Порівняння реалізації механізму атомік-змінних в різних мовах програмування та бібліотеках паралельного програмування**

На основі теоретичних відомостей з попередніх підпунктів можна сказати, що часто в мовах програмування реалізація атомарних змінних зроблена за допомогою спеціальних класів, низькорівнева реалізація функцій котрих дозволяє виконувати операції з атомарними змінними у неблокуючому вигляді. Ці класи надають можливість використовувати як атомарні змінні більшість простих типів, а також масиви з простих типів. Це стосується мови програмування *Java*.

В мові програмування *C#* використовуються спеціальні функції класу Interlocked, що беруть в якості аргументів змінні, операції з якими потрібно зробити атомарно.

Також ці мови підтривують механізм атомарних змінних за допомогою *volatile-*змінних. Це дозволяє бути впевненим, що всі потоки будуть завжди бачити останню версію змінної. Проте для реалізації більш складних операцій *volatile*-змінні вже не підходять.

Щодо мови програмування Ада, то там можна зробити атомарною будь-яку змінну простого чи складного типу за допомогою використання прагми. При цьому операції, що будуть атомарними з цими змінними — лише читання та запис. Це схоже на механізм *volatile*-змінних, що можна знайти в *Java* та *C#.*

В бібліотеці параллельного програмування *Win32* використовується схожий механізм, як і у мові програмування *C#*, а саме використання функцій для реалізації механізму атомарних змінних.

В бібліотеці *OpenMP* використовується зовсім інший підхід — там потрібно написати спеціальну прагму, що дозволяє зробити атомарною операцію присвоєння, що йде безпосередньо після прагми, при цьому тільки стосовно змінної, що стоїть зліва від оператора присвоєння.

## **1.7 Висновки до розділу 1**

1. Виконано аналіз механізму використання атомарних змінних в мові програмування Ада. Показано, що змінні оголошуються як атомарні за допомогою спеціальних прагм. В залежності від типу змінної (простий чи складний), використовується або прагма Atomic, або Atomic\_Component відповідно.
2. На основі аналізу атомарних змінних в мові програмування *Java* можна зробити висновок, що в мові *Java* атомарна змінна позначається за допомогою ключового слова volatile при її оголошенні, що дозволяє усім процесам завжди “бачити” найновішу версію цієї змінної. Проте, якщо потрібно виконати більш складні операції, ніж читання та запис, наприклад, інкремент, то volatile можливостей змінних вже недостатньо. Тоді потрібно використовувати спеціальні Atomic класи, що реалізують необхідні атомарні операціями на низькому рівні.
3. Виконано аналіз роботи з атомарними змінними в мові програмування *C#*, який показам, що використання ключового слова volatile аналогічно його використанню його в мові *Java*, а для більш серйозних атомарних операцій, таких як інкремент, взаємообмін чи порівняння, потрібно використовувати методи класу Interlocked.
4. Виконано аналіз бібліотеки для паралельного програмування *OpenMP*, який показав, що їх реалізація зроблена за допомогою використання прагм бібліотеки omp atomic. Ця прагма робить змінну, що стоїть по ліву сторону від оператора присвоєння, що йде безпосередньо після прагми, атомарною. Завдяки цьому вирішується задача взаємного виключення.
5. На основі аналізу можливостей бібліотеки для паралельного програмування *Win32* можна зробити висновок, що ця бібліотека має всі необхідні інструменти для зручної роботи з атомарними змінними. В бібліотеці *Win32* операції над атомарними змінними виконуються за допомогою спеціальних функцій, назва яких починається з префіксу Interlocked. Бібліотека також має оптимізовані варіанти функцій для 64-бітної системи.
6. На основі порівняння механізму реалізації атомік-змінних в різних мовах та бібліотеках паралельного програмування можна зробити висновок, що реалізація в кожному випадку відрізніється, проте деякі концепції, як-от аналоги *volatile*-змінних, можна зустріти багато в більшості мов програмування (*Java, C#,* Ада). Інколи для атомарних операцій використовуються спеціальні функції (*Win32, C#*), а інколи — прагми (*OpenMP, Ада*).

# РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС З СП

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

Математична задача: A = (B\*C)\*X + e\*S\*(MR\*MT) - Z.

Мова (бібліотека): WinAPI.

Засоби організації взаємодії процесів: семафори, мютекси, події, критичні секції.

Структура паралельної комп’ютерної системи зі спільною пам’яттю, Р процесорами, 2 пристроями вводу-виводу представлена на рис. 2.1.



Рис. 2.1. Структура ПКС з СП, Р процесорами, 2 ПВВ

Для підрахування мінімального часу, згідно концепції нескінченного паралелізму, необхідного для обчислення заданої математичної задачі, необхідно прийняти кількість процесорів рівну нескінченності, час виконання однієї операції — одиничний, а також не враховувати час, що витрачається на пересилання інформації. Кількість кроків, необхідна для виконання скалярного добутку двох векторів розмірності N — ([log2 N] + 1), де [x] — найближче ціле число до х, що більше або дорівнює х.

Таблиця 2.1 Етапи алгоритму

|  |  |  |
| --- | --- | --- |
| № | Етап | Кількість операцій |
| 1 | Обрахунок скалярного добутку d = В\*С | [log2 N] + 1 |
| 2 | Обрахунок одного рядка добутку V = MR\*MT | [log2 N] + 1 |
| 3 | Обрахунок скалярного добутку m = S\*V | [log2 N] + 1 |
| 4 | Обрахунок кінцевого елементу Аі = d\*Xi + e\*m - Zi | 3 |

Етапи 2-4 виконується паралельно для кожного відповідного елемента структури.

Сумарна кількість операцій і, відповідно, мінімальний час, буде: ([log2 N] + 1) + ([log2 N] + 1) + ([log2 N] + 1) + 3 = 3 \* [log2 N] + 6.

## **2.1 Розробка паралельного математичного алгоритму**

1. di = BH \* CH , (i = 1, P), де P — кількість ядер в ПКС.
2. d = d + di , (i = 1, P)

СР: d

1. AH = d \* XH + e \* S \* (MRH \* MT) - ZH

СР: d, e, S, MT

## **2.2 Розробка алгоритмів процесів**

|  |  |  |
| --- | --- | --- |
|  | Т1 | КД, ТС |
| 1 | Введення C, e, MR |  |
| 2 | Сигнал T2...Tp про завершення вводу | S2..p,1 |
| 3 | Чекати завершення вводу в Tp | Wp,1 |
| 4 | Рахунок d1 = BH \* CH |  |
| 5 | Рахунок d = d + d1 | КД |
| 6 | Сигнал T2...Tp про завершення рахунку1 | S2..p,2 |
| 7 | Копіювання e1:=e, S1:=S, MT1:=MT | КД |
| 8 | Чекати завершення рахунку1 в T2...Tp | W2..p,2 |
| 9 | Копіювання d1:=d | КД |
| 10 | Рахунок AH = d1 \* XH + e1 \* S1 \* (MRH \* MT1) - ZH |  |
| 11 | Чекати завершення рахунку2 в T2..Tp | W2..p,3 |
| 12 | Виведення А |  |

|  |  |  |
| --- | --- | --- |
|  | Тi (1 < i < P) | КД, ТС |
| 1 | Чекати завершення вводу в T1, Tp | W1,1  Wp,1 |
| 2 | Рахунок di = BH \* CH |  |
| 3 | Рахунок d = d + di | КД |
| 4 | Сигнал T1...Ti-1, Ti+1...Tp про завершення рахунку1 | S1..i-1,1  Si+1..p,1 |
| 5 | Копіювання ei:=e, Si:=S, MTi:=MT | КД |
| 6 | Чекати завершення рахунку1 в T1...Ti-1, Ti+1...Tp | W1..i-1,2  Wi+1..p,2 |
| 7 | Копіювання di:=d | КД |
| 8 | Рахунок AH = di \* XH + ei \* Si \* (MRH \* MTi) - ZH |  |
| 9 | Сигнал T1 про завершення рахунку2 | S1,2 |

|  |  |  |
| --- | --- | --- |
|  | Тp | КД, ТС |
| 1 | Введення B, S, X, MT, Z |  |
| 2 | Сигнал T1...Tp-1 про завершення вводу | S1..p-1,1 |
| 3 | Чекати завершення вводу в T1 | W1,1 |
| 4 | Рахунок dp = BH \* CH |  |
| 5 | Рахунок d = d + dp | КД |
| 6 | Сигнал T1...Tp-1 про завершення рахунку1 | S1..p-1,2 |
| 7 | Копіювання ep:=e, Sp:=S, MTp:=MT | КД |
| 8 | Чекати завершення рахунку1 в T1...Tp-1 | W1..p-1,2 |
| 9 | Копіювання dp:=d | КД |
| 10 | Рахунок AH = dp \* XH + ep \* Sp \* (MRH \* MTp) - ZH |  |
| 11 | Сигнал T1 про завершення рахунку2 | S1,3 |

## **2.3 Розробка схеми взаємодії процесів**

Введення даних відбувається у першому та останньому потоці, виведення результату — у першому потоці.

Засоби синхронізації роботи потоків та вирішення задачі взаємного виключення:

* Semaphore\_Calculation1EndIn — масив, що зберігає семафори для синхронізації по завершенню перших обчислень.
* Semaphore\_CopyMt — семафор для вирішення задачі взаємного виключення при копіюванні матриці MT.
* Mutex\_CopyE — мютекс для вирішення задачі взаємного виключення при копіюванні змінної e.
* Mutex\_CopyS — мютекс для вирішення задачі взаємного виключення при копіюванні вектора S.
* CriticalSection\_UpdateD — критична секція для вирішення задачі взаємного виключення при зміні значення D.
* CriticalSection\_CopyD — критична секція для вирішення задачі взаємного виключення при копіюванні D.
* Event\_InputFinishIn1 — подія для синхронізації по завершенню вводу в першому потоці.
* Event\_InputFinishInP - подія для синхронізації по завершенню вводу в P-му потоці.
* Event\_Calculation2EndIn — масив, що зберігає події для синхронізації по завершенню других обчислень.

На рис. 2.2 наведена схема взаємодії процесів.



Рис. 2.2. Cхема взаємодії процесів

## **2.4 Розробка програми ПРГ1**

Програма для ПКС зі спільною пам’яттю реалізована на мові програмування *С++* з використанням бібліотеки паралельного програмування *WinAPI*.

Бібліотека паралельного програмування *WinAPI* дозволяє створювати потоки за допомогою задання потокової функції.

Програма складається з єдиного модуля main.cpp. Алгоритм головної програми ПКС зі спільною пам’яттю наведено у додатку А. Схема алгоритмів процесів для програми ПРГ1 наведено у додатку Б.

Основні змінні:

* N — розмірність даних.
* P — кількість процесорів.
* H — розмірність даних, що буде оброблятися одним потоком.
* STACK\_SIZE — кількість байт пам’яті, що необхідно виділити для роботи потока.
* DO\_PRINT — допоміжна змінна, встановлення якої у значення true виводить інформацію про стартування та закінчення потоків.

Основні створені типи:

* Vector — тип для зберігання вектору, а також деякі операції з ним.
* Matrix — тип для зберігання матриці, а також деякі операції з нею.

Основні методи:

* outputVector() — виведення значення вектора на экран.
* fillVectorOnes() - заповнення вектора одиничними значеннями.
* fillMatrixOnes() - заповнення матриці одиничними значеннями.
* ThreadFunctioin() - потокова функція.
* Main() - основний, стартовий метод.

Назви змінних для синхронізації роботи потоків та вирішення задачі взаємного виключення аналогічні з зазначеними на схемі взаємодії процесів.

Визначення часу роботи програми виконано за допомогою використання бібліотеки time.h.

Лістинг програми ПРГ1 наведено у додатку В.

## **2.5 Тестування програми ПРГ1**

Таблиця 2.2 Час виконання програми для ПРГ1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 | T5 | T6 |
| 1000 | 76 | 38 | 26 | 22 | 18 | 15 |
| 1250 | 156 | 79 | 53 | 45 | 37 | 31 |
| 1500 | 272 | 137 | 92 | 80 | 65 | 54 |

Таблиця 2.3 Значення Кп для ПРГ1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 | T5 | T6 |
| 1000 | 1.0 | 2.0 | 2.92 | 3.45 | 4.22 | 5.07 |
| 1250 | 1.0 | 1.97 | 2.94 | 3.47 | 4.22 | 5.03 |
| 1500 | 1.0 | 1.99 | 2.96 | 3.4 | 4.18 | 5.04 |

Таблиця 2.4 Значення Кe для ПРГ1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 | T5 | T6 |
| 1000 | 100 | 100 | 97 | 86 | 84 | 85 |
| 1250 | 100 | 98 | 98 | 87 | 84 | 84 |
| 1500 | 100 | 99 | 99 | 85 | 84 | 84 |

На підставі таблиць 2.2 і 2.3 побудуємо графіки поведінки Кп і Ке в залежності від N і P для ПРГ1.

На рис.2.3 наведено графік зміни коефіцієнту прискорення Кп в залежності від кількості ядер, на рис.2.4 - графік зміни коефіцієнту ефективності Ке.

Рис. 2.3 Програма ПРГ1. Графік зміни коефіцієнту прискорення Кп в залежності від кількості ядер. Операція

A = (B\*C)\*X + e\*S\*(MR\*MT) - Z. N = 1000/1250/1500

Рис. 2.4 Програма ПРГ1. Графік зміни коефіцієнту ефективності Ке в залежності від кількості ядер. Операція

A = (B\*C)\*X + e\*S\*(MR\*MT) - Z. N = 1000/1250/1500

**2.6 Висновки до розділу 2**

Виконано розробку програми зі спільною пам’яттю по заданому алгоритму на мові програмування *С++* з використанням бібліотеки паралельного програмування *WinAPI*. Тестування програми для різної кількості ядер та різної розмірності даних показало наступне:

1) При використанні багатоядерної ПКС час виконання програми і, як наслідок, обчислення математичної задачі скорочується.

2) Значення коефіцієнта прискорення належать проміжку від 1.97 (для конфігурації ПКС з Р = 2, N = 1250), до 5.07 (для конфігурації ПКС з Р = 6, N = 1000. При цьому тенденція зміни коефіцієнту прискорення однакова для різної розмірності даних.

3) Можна зробити висновок, що при збільшенні кількості ядер в системі прискорення обчислення математичної задачі зростає, проте зі зменшенням темпу росту.

3) Коефіцієнта ефективності приймає значення від 100% (для конфігурації ПКС з Р = 2, N = 1000) до 84% (для конфігурації ПКС з Р = 6, N = 1500). При цьому тенденція зміни коефіцієнту ефективності однакова для різної розмірності даних

4) Можна зробити висновок, що при збільшенні кількості ядер доступних для обчислення математичної задачі, коефіцієнт ефективності спадає. Це пояснюється тим, що при більшій кількості потоків збільшуються затримки на синхронізацію потоків між собою.

5) У вимірюваннях присутні похибки, що пояснюється неоднаковістю середовища роботи програми через сторонні чинники(операційну систему), а також тим, що характер стартування та роботи потоків буде різним для різних запусків.

# РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС З ЛП

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

Математична задача: A = (B\*C)\*X + e\*S\*(MR\*MT) - Z.

Бібліотека: MPI.

Мова програмування: С++.

Структура ПКС з ЛП: решітка.

Засоби організації взаємодії процесів: механізм відправлення повідомлень.

Структура паралельної комп’ютерної системи зі спільною пам’яттю, Р процесорами, 2 пристроями вводу-виводу представлена на рис. 3.1.

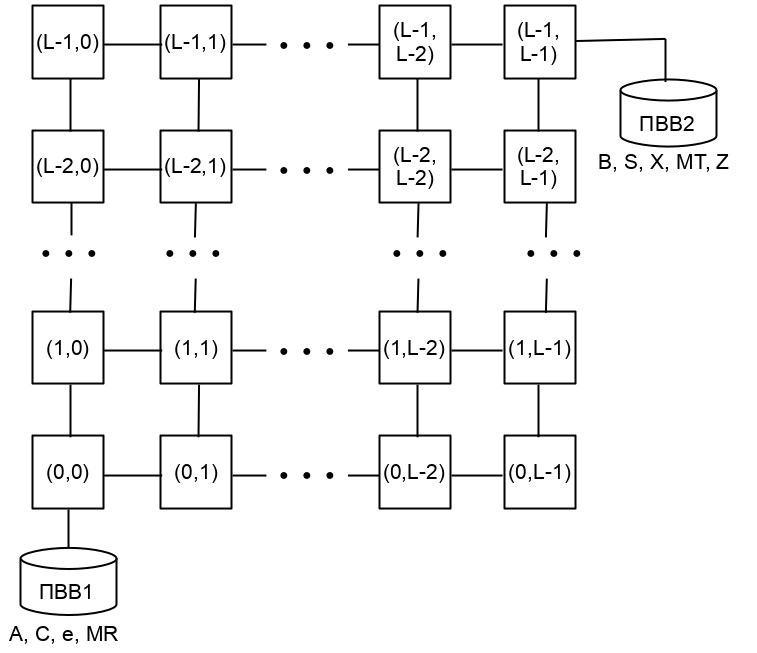


Рис 3.1. Структура ПКС з ЛП

L позначає розмірність решітки по кожній з координат. Кожна вершина має два індекси: порядковий номер рядка та колонки.

## **3.1 Розробка паралельного математичного алгоритму**

1. di = BH \* CH , (i = 1, P)
2. d = d + di , (i = 1, P)
3. AH = d \* XH + e \* S \* (MRH \* MT) - ZH

Позначення:

* N — розмірність даних.
* P — кількість ядер в ПКС.
* H = N / P — розмірність шматочків (під-векторів та під-матриць).

## **3.2 Розробка алгоритмів процесів**

|  |  |
| --- | --- |
|  | Тi,j (i=0..L-1, j=0..L-1) |
| 1 | **Якщо** i=0 **та** j=0 **то**:   * Введення C, e, MR   **Інакше** **Якщо** i=L-1 **та** j=L-1 **то**:   * Введення S, MT, B, X, Z |
| 2 | **Якщо** i=0 **та** 0<=j<=L-2 **то**:   * Передати в T0,j+1:e, C(L-j-1)\*L\*H, MR(L-j-1)\*L\*H   **Якщо** i=0 **та** 1<=j<=L-1 **то**:   * Прийняти від T0,j-1: e, C(L-j)\*L\*H, MR(L-j)\*L\*H   **Якщо** i=L-1 **та** 1<=j<=L-2 **то**:   * Передати в TL-1,j-1:S, MT, Bj\*L\*H, Xj\*L\*H, Zj\*L\*H   **Якщо** i=L-1 **та** 0<=j<=L-2 **то**:   * Прийняти від TL-1,j+1: S, MT, B(j+1)\*L\*H, X(j+1)\*L\*H, Z(j+1)\*L\*H |
| 3 | **Якщо** i=0 **то**:   * Передати в Ti+1,j:e, C(L-i-1)\*H, MR(L-i-1)\*H * Прийняти від Ti+1,j: S, MT, B(i+1)\*H, X(i+1)\*H, Z(i+1)\*H   **Інакше Якщо** i<[L/2] **то**:   * Прийняти від Ti-1,j: e, C(L-i)\*H, MR(L-i)\*H * Передати в Ti+1,j:e, C(L-i-1)\*H, MR(L-i-1)\*H * Прийняти від Ti+1,j: S, MT, B(i+1)\*H, X(i+1)\*H, Z(i+1)\*H * Передати в Ti-1,j:S, MT, Bi\*H, Xi\*H, Zi\*H   **Якщо** i=L-1 **то**:   * Передати в Ti-1,j:S, MT, Bi\*H, Xi\*H, Zi\*H * Прийняти від Ti-1,j: e, C(L-i)\*H, MR(L-i)\*H   **Інакше Якщо** i>[L/2] **то**:   * Прийняти від Ti+1,j: S, MT, B(i+1)\*H, X(i+1)\*H, Z(i+1)\*H * Передати в Ti-1,j:S, MT, Bi\*H, Xi\*H, Zi\*H * Прийняти від Ti-1,j: e, C(L-i)\*H, MR(L-i)\*H * Передати в Ti+1,j:e, C(L-i-1)\*H, MR(L-i-1)\*H   **Якщо** i > 2 **та** i=[L/2] **то**:   * Прийняти від Ti+1,j: S, MT, B(i+1)\*H, X(i+1)\*H, Z(i+1)\*H * Прийняти від Ti-1,j: e, C(L-i)\*H, MR(L-i)\*H * Передати в Ti-1,j:S, MT, Bi\*H, Xi\*H, Zi\*H * Передати в Ti+1,j:e, C(L-i-1)\*H, MR(L-i-1)\*H |
| 4 | Рахунок d := BH \* CH |
| 5 | **Якщо** i <= L-2 **то**:   * Прийняти від Ti+1,j: d\_other * d := d + d\_other   **Якщо** i >= 1 **то**:   * Передати в Ti-1,j:d |
| 6 | **Якщо** i = 0 **та** j <= L-2 **то**:   * Прийняти від Ti,j+1: d\_other * d := d + d\_other   **Якщо** i = 0 **та** j >= 1 **то**:   * Передати в Ti,j-1:d |
| 7 | **Якщо** i = 0 **та** j >= 1 **то**:   * Прийняти від Ti,j-1: d\_new * d := d\_new   **Якщо** i = 0 **та** j <= L-2 **то**:   * Передати в Ti,j+1:d |
| 8 | **Якщо** i >= 1 **то**:   * Прийняти від Ti-1,j: d\_new * d := d\_new   **Якщо** i <= L-2 **то**:   * Передати в Ti+1,j:d |
| 9 | Рахунок AH = d \* XH + e \* S \* (MRH \* MT) - ZH |
| 10 | **Якщо** i <= L-2 **то**:   * Прийняти від Ti+1,j: A(L-i-1)\*H   **Якщо** i >= 1 **то**:   * Передати в Ti-1,j:A(L-i)\*H |
| 11 | **Якщо** i = 0 **та** j <= L-2 **то**:   * Прийняти від Ti,j+1: A(L-i-1)\*L\*H   **Якщо** i = 0 **та** j >= 1 **то**:   * Передати в Ti,j-1:A(L-i)\*L\*H |
| 12 | **Якщо** i=0 **та** j=0 **то**:   * Виведення А |

Позначення:

* [x] — найбільше ціле число, таке що [x] <= x.

## **3.3 Розробка схеми взаємодії процесів**

Введення даних відбувається у першому та останньому потоці, виведення результату — у першому потоці.

На рис. 3.2 наведена структурна схема взаємодії задач, на якій наочно видно, як саме відбувається пересилка даних у ПКС.

Передача та прийняття повідомлень відбуваються за допомогою використання функцій MPI\_Recv, MPI\_Send, які блокують процес, в якому вони були викликані, доки не завершиться прийняття чи передача повідомлень, відповідно, а також функції MPI\_Isend, яка не блокує процес, що дозволяє продовжити виконання програми поки йде пересилка.

При складанні алгоритму були взяті до уваги такі важливі пункти:

* Розсилання даних по всій системі робиться рівно за кількість етапів, що дорівнює діаметру системи (в даному випадку це 2\*(L-1)), що забезпечую найшвидше можливе розсилання.
* Черговість операцій прийняття та відправлення були зроблені такими, аби виключити можливість дедлоків. Вони могли виникнути, наприклад, коли два сусідні потоки обидва очікують повідомлень один від одного, проте жоден не може продовжити, адже спочатку одному з них треба було б прийняти повідомлення від свого “напарника”.

Схема взаємодії потоків наведена на рис 3.2.

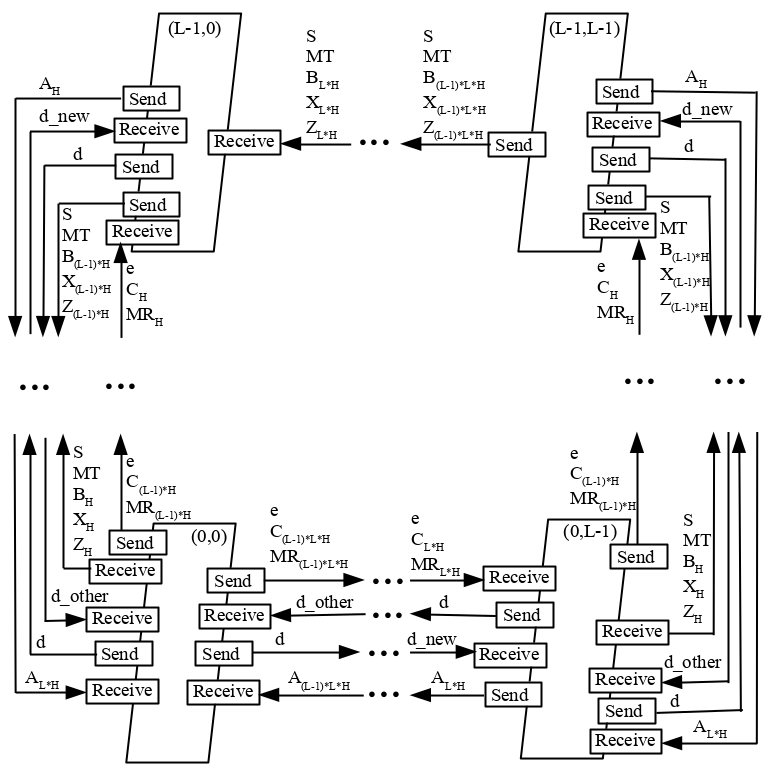


Рис 3.2. Схема взаємодії процесів

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Жуков И., Корочкин А.* Параллельные и распределенные вычисления. Лабораторный практикум. Учебно-методическое пособие. К.: «Корнейчук», 2008. – 224 с.
2. *Volatile Vs Atomic* [Електронний ресурс] – Режим доступу: <https://stackoverflow.com/questions/19744508/volatile-vs-atomic>
3. *Package java.util.concurrent.atomic* [Електронний ресурс] – Режим доступу: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html>
4. *Interlocked Operations* [Електронний ресурс] – Режим доступу: <https://docs.microsoft.com/en-us/dotnet/standard/threading/interlocked-operations>
5. *Interlocked Class* [Електронний ресурс] – Режим доступу: [https://msdn.microsoft.com/en-us/library/system.threading.interlocked%28v=vs.110%29.aspx?f=255&MSPPError=-2147217396](https://msdn.microsoft.com/en-us/library/system.threading.interlocked(v=vs.110).aspx?f=255&MSPPError=-2147217396)
6. *Volatile vs. Interlocked vs. lock* [Електронний ресурс] – Режим доступу: <https://stackoverflow.com/questions/154551/volatile-vs-interlocked-vs-lock>
7. *Basic OpenMP Atomic Operations* [Електронний ресурс] – Режим доступу: <https://software.intel.com/en-us/node/608159>
8. *atomic* [Електронний ресурс] – Режим доступу: <https://msdn.microsoft.com/en-us/library/8ztckdts.aspx>
9. *Interlocked Variable Access* [Електронний ресурс] – Режим доступу: <https://msdn.microsoft.com/en-us/library/windows/desktop/ms684122(v=vs.85).aspx>

ДОДАТОК А. Алгоритм головної програми ПКС зі спільною пам’яттю



ДОДАТОК Б. Схема алгоритмів процесів для програми ПРГ1



ДОДАТОК В. Лістинг програми ПРГ1

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

// Course Project: WinAPI. Semaphores, Events, Mutexes, Critical Sections

// Task: A = (B\*C) \* X + e \* S \* (MR\*MT) - Z

// Author: Igor Boyarshin

// Date: 23.03.2018

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

#include "windows.h"

#include <iostream>

#include <time.h>

// Constants

const unsigned int N = 2000;

const unsigned int P = 4;

const unsigned int H = N / P;

const unsigned int STACK\_SIZE = 100000000;

const bool DO\_PRINT = true;

// Types

struct Vector {

private:

unsigned int elements[N];

public:

Vector() {

std::fill(elements, elements + N, 0);

}

unsigned int& operator[](unsigned int row) {

return elements[row];

}

const unsigned int& operator[](unsigned int row) const {

return elements[row];

}

Vector copy() const {

Vector v;

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

v[i] = elements[i];

}

return v;

}

};

struct Matrix {

private:

struct Vector elements[N];

public:

struct Vector& operator[](unsigned int row) {

return elements[row];

}

const struct Vector& operator[](unsigned int row) const {

return elements[row];

}

Matrix copy() const {

Matrix m;

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

const struct Vector& v = elements[i];

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

m[i][j] = v[j];

}

}

return m;

}

};

// Functions

void fillVector(struct Vector& vector, unsigned int value);

void fillMatrix(Matrix& matrix, unsigned int value);

void outputVector(const struct Vector& vector);

// Data

int e, d;

Vector A, B, C, X, S, Z;

Matrix MR, MT;

// Semaphores

HANDLE Semaphores\_Calculation1EndIn[P];

HANDLE Semaphore\_CopyMt;

// Mutexes

HANDLE Mutex\_CopyE;

HANDLE Mutex\_CopyS;

// Critical Sections

CRITICAL\_SECTION CriticalSection\_UpdateD;

CRITICAL\_SECTION CriticalSection\_CopyD;

// Events

// manual

HANDLE Event\_InputFinishIn1;

HANDLE Event\_InputFinishInP;

// automatic

HANDLE Event\_Calculation2EndIn[P - 1];

typedef struct ThreadParameters {

unsigned int tid;

} \*Pointer;

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

// Threads

void ThreadFunction(LPVOID arguments) {

const unsigned int tid = (static\_cast<Pointer>(arguments))->tid;

const unsigned int low = (tid - 1) \* H;

const unsigned int high = tid \* H;

if (DO\_PRINT) {

std::cout << "Thread " << tid << " started..." << std::endl;

}

// Input

switch (tid) {

case 1:

e = 1;

fillVector(C, 1);

fillMatrix(MR, 1);

break;

case P:

fillVector(B, 1);

fillVector(S, 1);

fillVector(X, 1);

fillMatrix(MT, 1);

fillVector(Z, 1);

break;

default:

break;

}

// Synchronization on input

switch (tid) {

case 1:

SetEvent(Event\_InputFinishIn1);

WaitForSingleObject(Event\_InputFinishInP, INFINITE);

break;

case P:

SetEvent(Event\_InputFinishInP);

WaitForSingleObject(Event\_InputFinishIn1, INFINITE);

break;

default:

WaitForSingleObject(Event\_InputFinishIn1, INFINITE);

WaitForSingleObject(Event\_InputFinishInP, INFINITE);

}

// Calculation1

int di = 0;

for (unsigned int i = low; i < high; i++) {

di += B[i] \* C[i];

}

// Update common d

EnterCriticalSection(&CriticalSection\_UpdateD);

d += di;

LeaveCriticalSection(&CriticalSection\_UpdateD);

// Signal about the end of Calculations1

ReleaseSemaphore(Semaphores\_Calculation1EndIn[tid - 1], P, NULL);

// Copy1

WaitForSingleObject(Mutex\_CopyE, INFINITE);

const int ei = e;

ReleaseMutex(Mutex\_CopyE);

WaitForSingleObject(Mutex\_CopyS, INFINITE);

const Vector Si = S.copy();

ReleaseMutex(Mutex\_CopyS);

WaitForSingleObject(Semaphore\_CopyMt, INFINITE);

const Matrix MTi = MT.copy();

ReleaseSemaphore(Semaphore\_CopyMt, 1, NULL);

// Wait for the end of Calculations1 in other threads

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

WaitForSingleObject(Semaphores\_Calculation1EndIn[i], INFINITE);

}

// Copy d

EnterCriticalSection(&CriticalSection\_CopyD);

di = d;

LeaveCriticalSection(&CriticalSection\_CopyD);

// Calculations2

for (unsigned int h = low; h < high; h++) {

const Vector& row = MR[h];

int elem = 0;

for (unsigned int i = 0; i < N; i++) { // cols 2

int temp = 0;

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

temp += row[j] \* MTi[j][i];

}

elem += temp \* Si[i];

}

A[h] = di \* X[h] + ei \* elem - Z[h];

}

// Synchronization of Calculation2 end

switch (tid) {

case 1:

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

WaitForSingleObject(Event\_Calculation2EndIn[i], INFINITE);

}

break;

default:

// -2: one for zero-based, one for T1 not being accounter for here

SetEvent(Event\_Calculation2EndIn[tid - 2]);

}

if (tid == 1) {

// Output A

if (N <= 8) {

outputVector(A);

}

}

if (DO\_PRINT) {

std::cout << "Thread " << tid << " finished." << std::endl;

}

};

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

int main() {

// Preparations

fillVector(A, 0);

// Semaphores

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

Semaphores\_Calculation1EndIn[i] = CreateSemaphore(

NULL, // default security attributes

0, // initial count

P, // maximum count

NULL // unnamed semaphore

);

}

Semaphore\_CopyMt = CreateSemaphore(NULL, 1, 1, NULL);

// Mutexes

Mutex\_CopyE = CreateMutex(

NULL, // default security attributes

FALSE, // initially not owned

NULL // unnamed

);

Mutex\_CopyS = CreateMutex(NULL, FALSE, NULL);

// Critical Sections

InitializeCriticalSection(&CriticalSection\_UpdateD);

InitializeCriticalSection(&CriticalSection\_CopyD);

// Events

// manual reset

Event\_InputFinishIn1 = CreateEvent(

NULL, // default security attibutes

TRUE, // manual reset

FALSE, // initial - not signaled

NULL // unnamed event

);

Event\_InputFinishInP = CreateEvent(NULL, TRUE, FALSE, NULL);

// automatic reset

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

Event\_Calculation2EndIn[i] = CreateEvent(NULL, FALSE, FALSE, NULL);

}

// Threads

struct ThreadParameters arguments[P];

const time\_t start = clock();

DWORD Tids[P];

HANDLE Threads[P];

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

arguments[i].tid = (i + 1);

Threads[i] = CreateThread(

NULL, // Default secutiry attributes

STACK\_SIZE, // stack size

(LPTHREAD\_START\_ROUTINE)ThreadFunction, // thread function name

&arguments[i], // argument to thread function

0, // creating flags(0 => run immediately after creation)

&Tids[i] // thread identifier

);

}

// Cleanup

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

WaitForSingleObject(Threads[i], INFINITE);

CloseHandle(Threads[i]);

}

const time\_t finish = clock() - start;

std::cout << "Elapsed time = " << static\_cast<double>(finish) / 1000.0

<< std::endl;

std::cin.get();

}

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

// Function definitions

void fillVector(struct Vector& Vector, unsigned int value) {

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

Vector[i] = value;

}

void fillMatrix(Matrix& matrix, unsigned int value) {

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

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

matrix[i][j] = value;

}

void outputVector(const struct Vector& vector) {

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

std::cout << vector[i] << " ";

}

std::cout << std::endl;

}

ДОДАТОК В. Лістинг програми ПРГ2

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

// Course Project: MPI

// Task: A = (B\*C) \* X + e \* S \* (MR\*MT) - Z

// Author: Igor Boyarshin

// Date: 15.04.2018

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