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»

Березинця Артема Андрійовича

(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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *Березинцю Артему Андрійовичу*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(прізвище, ім’я, по-батькові)*

1. Тема роботи «Огляд засобів роботи з потоками в мові Python»

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

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 \_\_\_\_\_\_\_\_\_ \_\_\_\_** Березинець А.А.

( підпис ) (прізвище та ініціали)

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

( підпис ) (прізвище та ініціали)

ЗМІСТ

[ВСТУП 5](#__RefHeading___Toc4683_851943916)

[РОЗДІЛ 1. ОГЛЯД ЗАСОБІВ РОБОТИ З ПОТОКАМИ В МОВІ *PYTHON* 6](#__RefHeading___Toc4685_851943916)

[1.1 Клас local 8](#__RefHeading___Toc4687_851943916)

[1.2 Клас Thread 8](#__RefHeading___Toc4689_851943916)

[1.3 Клас Lock 10](#__RefHeading___Toc4691_851943916)

[1.4 Клас RLock 12](#__RefHeading___Toc4693_851943916)

[1.5 Клас Condition 13](#__RefHeading___Toc4695_851943916)

[1.6 Клас Semaphore 15](#__RefHeading___Toc4697_851943916)

[1.5 Клас Event 17](#__RefHeading___Toc4699_851943916)

[1.6 Клас Barrier 18](#__RefHeading___Toc4701_851943916)

[1.7 Висновки до розділу 1 20](#__RefHeading___Toc4703_851943916)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 22](#__RefHeading___Toc4705_851943916)

# ВСТУП

З появою багатоядерних процесорів виникла необхідність створення відповідних програм, які матимуть змогу використовувати доступні ресурси комп’ютера на повну. В той час як звичайна (послідовна) програма виконує інструкції по черзі, одна за одною, паралельна програма має декілька потоків виконання, кожен з яких виконує свої інструкції, через що загалом отримуємо паралельну програму.

В системі, яка складається з *N* ядер, в ідеальному випадку можна отримати пришвидшення паралельної програми у *N* разів, в порівнянні з послідовно [3]. Проте це залежить від можливості розпаралелювання математичного алгоритму, від якості написання коду для програми, від процесів, що виконуються на машині окрім паралельної програми тощо. Тому в реальних умовах пришвидшення в *N* разів майже ніколи не досягається.

Існує багато мов та бібліотек програмування, що підтримують написання паралельних програм. Вони дають змогу програмісту використовувати різні інструменти для створення та синхронізації роботи потоків між собою.

У першому розділі розглядаються особливості роботи з потоками в мові програмування Python*.* Python— високорівнева мова програмування, що останнім часом набуває великої популярності. Вона має всі необхідні високорівневі інструменти для організації взаємодії потоків, що дозволяють легко написати ефективну паралельну програму.

У другому розділі...

У третьому розділі ...

# РОЗДІЛ 1. ОГЛЯД ЗАСОБІВ РОБОТИ З ПОТОКАМИ В МОВІ *PYTHON*

Python— інтерпретована, високорівнева мова програмування, що використовується як загальна, багатоцільова мова [1]. Вона була створена в 1991 році з акцентом на легкість її читання. Pythonмає динамічну типізацію та автоматичний механізм роботи з пам’яттю.

Pythonпропонує програмісту широкий та сучасний набір інструментів для створення та організації роботи потоків.

Для роботи з потоками в мові програмування Pythonвикористовується спеціальний модуль threading, що зроблений на основі більш низькорівневого модуля threadдля легкості використання. Функції, що доступні в цьому модулі [2], наведені в таблиці 1.1.

Таблиця 1.1. Функцій модуля threading

|  |  |
| --- | --- |
| active\_count() | Повертає поточну кількість живих об’єктів класу Thread. Це значення рівне розміру списку, що повертається методом enumerate(). |
| current\_thread() | Повертає об’єкт класу Thread, що відповідає поточному потоку. Якщо цей потік було створено не через модуль threading, то повертається фіктивний потік з обмеженою функціональністю. |
| get\_indent() | Повертає ідентифікатор поточного потоку, що є цілим числом, що більше за нуль. Це значення не має ніякого прямого значення. Воно використовується, наприклад, у словниках для індексування даних, специфічних для потоку. Це значення може бути використано повторно, коли попередній потік уже завершив свою роботу, і створюється новий. |
| enumerate() | Повертає список всіх потоків, що зараз живі. Цей список включає демонічні потоки, фіктивні потоки, створені current\_thread(), а також основний потік. |
| main\_thread() | Повертає основний об’єкт Thread. За звичайних умов він відповідає потоку, в якому стартував інтерпретатор пітону. |
| settrace(func) | Встановлює трасуючу функцію для всіх потоків, що були створені за допомогою модуля threading. Функція func буде передана до sys.settrace для кожного потоку, перед тим як буде запущено на виконання метод run(). |
| setprofile(func) | Встановлює профіль для всіх потоків, що були створені за допомогою модуля threading. Функція func буде передана до sys.setprofile для кожного потоку, перед тим як буде запущено на виконання метод run(). |
| stack\_size([size]) | Повертає значення стеку, що буде використовуватися у потоці при його створенні. Опціональний аргумент size використовується, якщо необхідно задати розмір стеку для наступник потоків, що будуть створюватися. Він має бути рівним 0, або бути позитивним цілим числом, як мінімум 32768. Якщо не було задано size, то використовується 0. Якщо не підтримується зміна розміру стеку, то кидається RuntimeError. Якщо задане значення розміру неправильне, кидається ValueError, а значення стеку не змінюється. 32 кілобайти — мінімальне значення розміру стеку, що гарантує правильну роботу інтерпретатора. В деяких системах можуть існувати обмеження на розмір стеку. |
| TIMEOUT\_MAX | Максимальне значення, що допускається для параметра timeout у блокуючих функціях (Lock.acquite(), RLock.acquire(), Condition.wait()). Якщо задати значення timeout, що перевищує це значення, то кидається OverflowError. |

Модуль *threading* також має декілька спеціальних класів. Далі наведений опис їх роботи [4].

## **1.1 Клас local**

Локальні дані потоку — такі дані, значення яких специфічні для кожного потоку. Для створення локальних даних потрібно створити об’єкт класу threading.local. Приклад використання наведено в Прикладі 1.1.

**Приклад 1.1.** Використанняthreading.local.

someData = threading.local()

someData.name = “first name”

someData.value = 5

## **1.2 Клас Thread**

Цей клас представляє собою потік, що виконується паралельно. Є два варіанти, як можна задати дію, що треба виконати. По-перше, передавши конструктору об’єкт, що треба виконати. По-друге, за допомогою метода run() в дочірньому класі. В дочірньому класі не повинно бути перезадано ніяких інших методів, окрім конструктора. Іншими словами, перезадавати можна тільки методи \_\_init\_\_() та run() цього класу.

Потік запускається на виконання викликанням метода start(). Це стартує метод run() в окремому потоці. Починаючи з цього моменту, потік вважається “живим”. Він перестає бути живим, коли його метод run() завершує своє виконання звичайним шляхом, або через кидання виключення. Перевірити, чи є потік живим, можна за допомогою метода is\_alive().

Кожен потік має власне ім’я. Його можна задати через аргумент конструктору, або напряму через поле name.

Потік можна зробити демоном. Це має наступний ефект: якщо залишилися тільки потоки-демони, то програма завершує своє виконання. Як і з ім’ям, цей прапорець можна задати або через аргумент конструктора, або напряму через поле daemon.

Конструктор класу Thread має наступні аргументи:

* group — повинен бути встановлений у None. Буде використовуватися у майбутньому для задання групи потоку.
* target — метод, що має бути викликаний при виклику run(). За замовчуванням дорівнює None, тобто нічого не викликається.
* name — ім’я потоку.
* args — подвійний кортеж аргументів.
* kwargs — словник аргументів.

Перелік методів класу Thread наводиться у таблиці 1.2.

Таблиця 1.2. Методи класу Thread

|  |  |
| --- | --- |
| start() | Стартує потік. Цей метод має бути викликаним не більше, ніж один раз. Цей метод, в свою чергу, викликає метод run() в окремому потоці виконання. Якщо цей метод буде викликаним більше одного раз для якогось об’єкту, то кидається RuntimeError. |
| run() | Метод, що буде виконуватися як тіло потоку. Може бути перезаданий у дочірньому класі. |
| join(timeout=None) | Чекати завершення потоку. Блокує потік, з якого був викликаний метод, до тих пір, поки не завершить виконання (звичайним способом, або аварійним) той потік, чий метод joint() було викликано, або коли пройде зазначений час. Так як цей метод завжди повертає None, то аби дізнатися, чи завершив виконання потік, або просто закінчився зазначений час, необхідно викликати метод is\_alive() після методу join(). |
| name | Ім’я потоку. Не має ніякого спеціального значення. Декілька потоків можуть мати однакове ім’я. |
| getName() | Застарілий метод, що повертає значення поля name. Зараз потрібно доступатися до поля напряму. |
| setName() | Застарілий метод, що встановлює значення поля name. Зараз потрібно доступатися до поля напряму. |
| ident | Ідентифікатор потоку, що дорівнює None якщо потік ще не стартував. Є додатнім цілим числом. Може бути використаний повторно, коли один потік завершив своє виконання і починає виконання новий. |
| is\_alive() | Повертає значення, чи завершив ще живий цей потік. Повертає True починаючи з безпосередньо старту методу run() і до моменту безпосередньо після його завершення. Функція enumerate() повертає список всіх живих на даний момент потоків. |
| daemon | Булева змінна, що показує, чи є цей потік демоном. Має бути встановлена перед викликом методу start(), інакше кидається RuntimeError. Загальна Python програма завершується, коли завершують виконання всі потоки, що не є демонами. |
| isDaemon() | Застарілий метод, що повертає поточне значення змінної daemon. Зараз потрібно доступатися до неї напряму. |
| setDaemon() | Застарілий метод, що встановлює поточне значення змінної daemon. Зараз потрібно доступатися до неї напряму. |

## **1.3 Клас Lock**

Лок (клас threading.Lock) — об’єкт синхронізації, що їм володіє якийсь потік. На даний момент це є найнижчим доступним рівнем синхронізації в Python.

Такий лок може бути в одному з двох станів - “закритий” (“захоплений”) та “відкритий” (“відпущений”). Спочатку він знаходиться в стані відкритий. У лока є дві базові операції — acquire() та release(). Якщо поточний стан локу — відкритий, то метод acquire() закриває його та повертає виконання одразу. Якщо ж поточний стан лока — закритий, то метод acquire() блокує потік, з якого він був викликаний, до того моменту, поки з якогось іншого потоку не буде викликаний метод release() для цього лока. Метод release() потрібно викликати тільки зі стану закритий. Він змінює стан на відкритий та одразу повертає виконання. Якщо ж викликати його для відкритого лока, то кидається RuntimeError.

Якщо більше одного потоку чекають на відкриття лока, то лише один з них продовжить подальше виконання, коли буде викликаний метод release() та змінить стан лока на відкритий. При цьому якийсь один з потоків знову змінить стан лока на закритий, але який саме — не відомо і може варіюватися на різних платформах.

Всі методи є атомарними.

Методи класу threading.Lock наведені в таблиці 1.3.

Таблиця 1.3. Методи класу threading.Lock

|  |  |
| --- | --- |
| acquire(blocking=True, timeout=-1) | Закриває (захоплює) лок.  Якщо метод був викликаний з аргументом blocking=True (за замовчуванням), то цей метод блокує потік, поки не буде відкрито лок, потік встановити стан локу як закритий і повернути True.  Якщо викликаний з blocking=False, то не блокує потік. При цьому якщо виклик з blocking=True заблокував би потік, то повертає False одразу. Інакше встановлює стан лока у закритий і повертає True.  Повертає значення True, якщо захват лока було проведено успішно, інакше — False (наприклад, коли завершився час очікування timeout). |
| release() | Відпустити (відкрити) лок.  Може бути викликаний з будь-якого потоку, не тобто лише того, в якому було захоплено лок.  Якщо лок закритий, но встановлює його у стан відкритий.  Якщо декілька потоків очікують відкриття лока, то дозволяє продовжити виконання лоше одному з них.  Якщо викликати цей метод для відкритого лока, то кидається RuntimeError.  Не повертає ніяке значення. |

## **1.4 Клас RLock**

Клас threading.Rlock надає клас для локу, що може бути захоплений захоплений декілька разів одним потоком. На низькому рівні, він використовує такі концепції як “потік-володар” та “рівень рекурсії”, окрім звичайних станів захоплений/відпущений. У стані закритий, деякий потік володіє локом. У відкритому стані ніякий потік не володіє ним.

Для того щоб закрити лок потрібно викликати метод acquire(), що повертає виконання як тільки потік буле володіти локом. Для того щоб відкрити лок, потрібно викликати його метод release(). Можуть існувати декілька пар acquire()/release(), але лише останній (найбільш зовнішній) release() відкриє лок. Метод release() має бути викликаний для кожного методу acquire().

Далі наведені особливості роботи методів acquire() та release() у порівнянні зі звичайним локом threading.Lock:

* acquire(blocking=True, timeout=-1) — захоплює лок. Коли викликаний без аргументів, то: якщо цей потік уже володіє локом, то збільшується рівень рекурсії та повертається управління. Інакше, якщо інший потік володіє локом, то блокує потік, поки не буде відкрито лок. Як тільки лок буде відритий (ніякий потік не буде ним володіти), то цей метод захоплює лок, встановлює рівень рекурсії рівний 1 та повертає управління. Якщо більше одного потоку блоковані та очікуються на відкриття локу, то тільки по одному вони зможуть заволодіти ним. В такому випадку ніяке значення не повертається методом.
* release() - відпускає лок та зменшує рівень рекурсії на 1. Якщо після декременту рівень рекурсії стає рівним 0. то встановлює стан лока у відкритий (ніякий потік ним не володіє), і якщо декілька потоків заблоковані і очікують на відкриття локу, то дозволяє лише одному з них продовжити виконання. Якщо ж після декременту рівень рекурсії все ще додатній, то лок залишається у стані закритий, і ним володіє потік, що викликав метод. Метод не повертає ніяке значення.

## **1.5 Клас Condition**

Змінна типу Condition завжди асоціюється з якимось годинником, який можна або передати через параметри самому, або буде створено годинник за замовчуванням. Передавання одного і того ж годинника може бути корисним, наприклад, коли декілька Condition змінних мають мати один і той же лок. Лок при цьому є частиною об’єкта Condition, тому окремо не потрібно за ним стежити.

Методи acquire() та release() самі викликають метод захоплення локу. Всі інші методи мають викликатися уже з захопленим локом. Метод wait() відпускає лок, а потім блокується, доки не буде розблокований методами notify() або notify\_all(). Як тільки потік буде розблокований, від знову захоплює лок та повертає керування. При цьому можна також вказати timeout.

Метод notify() розблокує один потік, який очікує на змінну Conditional, якщо такий потік взагалі є. Метод notify\_all() розблокує всі потоки, які очікують на змінну Conditional. Важливо розуміти, що ці два методи не відпускають лок. Це означає, що розблоковані потоки не повернуть керування одразу, а тільки тоді, коли потік, що викликав метод notify() або notify\_all(), відпустить лок.

Зазвичай змінні Confitional використовуються, коли потрібно синхронізувати доступ до спільного ресурсу або по якомусь стану. При виборі між методами notify() та notify\_all() потрібно відповісти на питання, чи потрібно, щоб рівно один потік зреагував на якусь зміну, чи всі разом.

Перелік методів класу threading.Condition наведено у таблиці 1.4.

Таблиця 1.4. Методи класу threading.Condition

|  |  |
| --- | --- |
| acquire(\*args) | Захоплює лок. Цей метод викликає відповідний метод локу, що зберігається у класі. При цьому цей метод повертає те, що повернув той відповідний метод. |
| release() | Відпускає лок. Цей метод викликає відповідний метод локу, що зберігається у класі. Метод нічого не повертає. |
| wait(timeout=None) | Чекати, поки не буде надісланий сигнал або поки не пройде timeout. Якщо на момент виклику цього методу потік не захопив лок, то кидається RuntimeError.  При старті цей метод відпускає відповідний лок та блокує метод, що його викликав. Він залишається в блокованому стані доти, поки не буде розблокований викликом методу notify() або notify\_all() з іншого потоку для відповідної змінної локу, або поки не закінчиться timeout. Як тільки цей потік розблокується, він знову захоплює лок.  Якщо наявний аргумент timeout, який не дорівнює None, то це має бути змінна з плаваючою комою, яка вказує необхідний час для очікування у секундах.  Метод повертає True, окрім випадку, коли стік timeout, тоді метод повертає False. |
| wait\_for(preficate, timeout=None) | Чекати, поки предикат не стане True. Аргумент timeout може бути використаний щоб вказати максимальний час, який потрібно очікувати.  Метод повертає останнє значення предикату, або False, якщо закінчився timeout.  Потрібно пам’ятати, що лок має бути звільненим перед викликом цього методу (як і з методом wait()), після чого лок знову захоплюється. |
| notify(n=1) | За замовчуванням, сповістити та розблокувати рівно один потік, що очікує на змінну локу, якщо такі потоки взагалі є. Якщо потік, що викликає цей метод, не володіє локом на момент виклику цього методу, то кидається RuntimeError.  Цей метод розблокує максимум n потоків, що очікують на змінну Condition. Якщо жоден потік не очікує, то метод нічого не робить. |
| nofity\_all() | Розблокує всі потоки, що очікують на змінну Condition. Цей метод працює точно так само, як і notify(), проте розблокує всі потоки, що очікують на змінну. Якщо потік, що викликав цей метод, не володіє локом на момент виклику, то кидається RuntimeError. |

## **1.6 Клас Semaphore**

Семафор є одним за найдавніших примітивів синхронізації, придуманий голландським вченим Дейкстрою [5]. Він використовував методи P() та V() замість acquire() та release().

Семафор побудований наступним чином. У ньому є лічильник, значення якого декрементується при кожному виклику метода acquire() та інкрементуєтся при кожному виклику методу release(). Цей лічильник не може стати від’ємним. Як тільки він стає нулем, то блокує потік, який його викликав, допоки якийсь інший потік не викличе метод release(), що інкрементує значення лічильнику, тобто лічильник перестає бути нулем.

Перелік методів класу threading.Semaphore наведено у таблиці 1.5.

Таблиця 1.5. Перелік методів класу threading.Semaphore

|  |  |
| --- | --- |
| acquire(blocking=True, timeout=None) | Захоплює семафор.  Якщо метод викликається без аргументів, то:  якщо значення внутрішнього лічильнику більше нуля, то зробити декремент та повернути керування одразу;  якщо значення внутрішнього лічильнику дорівнює нулю, то метод блокує потік, поки той не буде розблокований викликом методу release(). Як тільки метод буде розблокований (і значення лічильнику буде більшим за 0), то декрементувати значення лічильнику та повернути True. Кожен виклик методу release() буде розблоковувати рівно один потік. При цьому порядок, в якому потоки будуть розблокованими — непередбачений.  Якщо метод викликаний з blocking=False, то потік не буде заблокований. Якщо при цьому виклик методу без аргументу заблокувався б, то повернути False одразу. Інакше зробити те саме, як при виклику без аргументів, і повернути True.  Якщо викликаний з аргументом timeout, що не рівний None, то блокувати не більше ніж на timeout секунд. Якщо при цьому acquire() не завершиться успішно за відведений час, то повернути False. Інакше повернути True. |
| release() | Відпустити семафор, інкрементувати значення лічильнику. Якщо він до цього був нулем і інший потік при цьому очікує, поки той стане додатнім, то розблокувати цей потік. |

Різновидом класу threading.Semaphore є threading.BoundedSemaphore(value = 1), що його значення ніколи не стане більшим, ніж початкове. Якщо це значення стане більшим, ніж початкове, то кидається ValueError. У більшості випадків семафори використовуються для вирішення задачі взаємного виключення по доступу до спільного ресурсу. Тобто якщо семафор відпускається багато разів, то це ознака наявності багу. За замовчуванням значення value рівне одиниці.

Використання семафору для вирішення задачі взаємного виключення при доступі до серверу ілюструє приклад 1.2.

**Приклад 1.2**. Використання Семафору в Python

max\_connections = 3

sema = BoundedSemaphore(value=max\_connections)

with sema:

result = do\_stuff()

Використання threading.BoundedSemaphore знижує вірогідність помилки з боку програміста, коли він відпускає семафор більшу кількість разів, ніж його було захоплено.

## **1.5 Клас Event**

Клас threading.Event (від англ. Event — подія) — один з найпростіших механізмів організації взаємодій між потоками: один потік встановлює подію, а інший потік чекає на неї.

Об’єкт класу threading.Event має в собі внутрішній прапорець, що може бути встановлений у значення True за допомогою методу set() та скинутий у значення False методом clear(). Метод wait() блокує потік, що його викликав, поки значення прапорця не буде встановлено в True.

Значення прапорця за замовчуванням — False.

Перелік методів класу threading.Event наведено в таблиці 1.6.

Таблиця 1.6. Перелік методів класу threading.Event.

|  |  |
| --- | --- |
| is\_set() | Повертає True тоді і тільки тоді, коли прапорець встановлено. Інакше повертає False. |
| set() | Встановлює значення прапорця у True. Всі потоки, що були заблоковувані та очікують на встановлення цього прапорця — розблокуються. Потоки, що викликають метод wait() після того, як значення прапорця стало True — не блокуються взагалі. |
| clear() | Скинути значення прапорця на False. Подальші виклики метода wait() для цього прапорця будуть блокувати потік, що його викликав, доки значення прапорця не буде встановлено за допомогою метода set(). |
| wait(timeout=None) | Блокує потік, що його викликав, доки значення прапорця не стане True. Якщо значення прапорця на момент виклику True, то повернути керування одразу. Інакше блокувати потік, доки інший потік не встановить прапорець за допомогою метода set() або доки не вичерпається timeout.  Цей метод завжди повертає True, окрім випадку, коли цей метод завершився через вичерпання timeout, у цьому разі метод повертає False. |

## **1.6 Клас Barrier**

Цей клас надає простий примітив синхронізації, що може бути використаним певною кількістю потоків, які очікують один на одного. Кожен потік намагається пройти через бар’єр, викликаючи метод wait(), і буде заблокований доти, поки всі потоки не викликали метод wait(). Після цього моменту всі потоки пропускається через бар’єр одночасно.

Будь-який бар’єр може бути використаний повторно для тієї ж кількості потоків.

Використання бар’єру для синхронізування потоків клієнта та сервера наведено у прикладі 1.3.

**Приклад 1.3**. Використання бар’єру для синхронізації

bar = Barrier(2, timeout=5.5)

def server():

start\_server()

bar.wait()

while True:

connection = accept\_connection()

process\_server\_connection(connection)

def client():

b.wait()

while True:

connection = make\_connection()

process\_client\_connection(connection)

Для бар’єра також можна задати спеціальну дію (action) при його створенні. Тоді ця дія буде виконана одним з потоків, коли вся групі потоків пройде бар’єр.

Перелік методів класу threading.Barrier наведено в таблиці 1.7.

Таблиця 1.7. Перелік методів класу threading.Barrier.

|  |  |
| --- | --- |
| wait(timeout=None) | Пройти бар’єр. Коли всі потоки цього бар’єра викликали цей метод, то вони пропускаються одночасно. Якщо в метод переданий аргументом timeout, то бар’єр встановлюється у зламаний (broken) стан. Метод повертає значення від 0 до (parties — 1), різне для кожного потоку. Це значення може бути використано для виконання якоїсь спеціальної роботи потоком після проходження бар’єра.  Якщо було передано action до конструктора, то один з потоків виконає цю дію безпосередньо перед тим, як буде відпущений.  Цей метод кидає BrokenBarrierError якщо бар’єр буде зламаний (broken), або якщо бар’єр буде скинуто, поки на цього чекає потік. |
| reset() | Повертає бар’єр у звичайний, пустий стан. Всі потоки, що уже очікують на нього в цей момент, отримають BrokenBarrierError.  Потрібно пам’ятати, що ця функція може вимагати якоїсь зовнішньої синхронізації, якщо є інші потоки, чий стан не відомий. Якщо бар’єр зламаний, то хорошою ідеєю є просто залишення його та створення нового. |
| abort() | Встановлення бар’єр у зламаний стан. Як наслідок, всі поточні та майбутні виклики методу wait() отримають BrokenBarrierError. Цей метод можна використовувати, наприклад, якщо одному з потоків потрібно аварійно завершитися, але при цьому не потрібно привести систему у стан дедлоку.  Хорошою ідеєю є просто створення бар’єра з нормальним timeout, аби уникнути ситуації, коли один з потоків неправильно працює. |
| parties | Кількість потоків, на які буде очікувати бар’єр. |
| n\_waiting | Поточна кількість потоків, що очікують на бар’єр. |
| broken | Булеве значення, що дорівнює True, якщо бар’єр знаходиться у зламаному стані. |

## **1.7 Висновки до розділу 1**

1. Виконано аналіз інструментів роботи з потоками та їх синхронізації в мові програмування Python. Показано, що ця мова має всі необхідні сучасні інструменти, що дозволяють легко та швидко написати паралельну програму.
2. Показано, що в мові програмування Python є механізм Семафорів, що може бути використаний як для вирішення задачі взаємного виключення, так і для синхронізації потоків. Це робиться за допомогою класу threading.Semaphore.
3. Показано, що в мові Python наявний механізм посилки повідомлень, що реалізується за допомогою використання класу threading.Event. Він дозволяє встановлювати значення прапорця, на який очікує інший потоки (потоки), та які блокуються, доки не буде встановлення значення цього прапорця.
4. Наведено опис та приклади використання механізму бар’єрів у мові програмування Python. В цьому випадку кожен потік, що має пройти бар’єр, викликає метод wait(), і бар’єр пропускає всі потоки як тільки був викликаний метод wait() у всіх потоках.

# РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС З СП

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

Математична задача: *MA = MB \* MC + min(Z) \* MK*

Мова (бібліотека): *WinAPI*.

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

Структура паралельної комп’ютерної системи зі спільною пам’яттю, *Р* процесорами, 3 пристроями вводу-виводу представлена на рис. 2.1.

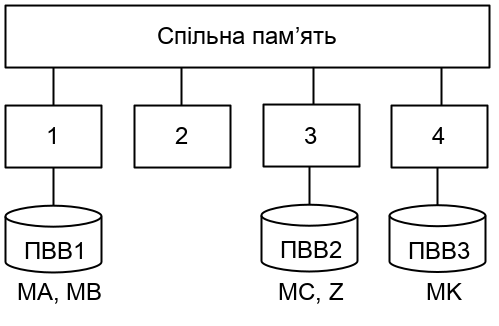


Рис. 2.1. Структура ПКС з СП, 4 процесорами, 3 ПВВ

## **2.1 Розробка паралельного математичного алгоритму**

1. xi = min(ZH), i=1..4
2. x = min(x, xi), i=1..4

СР: x

1. MAH = MBH \* MC + x \* MKH

СР: x, MC

Позначення:

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

## **2.2 Розробка алгоритмів процесів**

|  |  |  |
| --- | --- | --- |
| # | Ti (1 <= i <= 4) | КД, ТС |
| 1 | Якщо i=1: Ввід MB  Інакше якщо i=3: Ввід MC, Z  Інакше якщо i=4: Ввід MK |  |
| 2 | Якщо i=1,3,4: Сигнал T1..4 про завершення введення | S1..4\_1 |
| 3 | Чекати завершення введення в T1,3,4 | W1,3,4\_1 |
| 4 | Рахунок: xi = min(ZH) |  |
| 5 | Рахунок x = min(x, xi) | КД |
| 6 | Сигнал T1..4 про завершення рахунку1 | S1..4\_2 |
| 7 | Чекати завершення рахунку1 в T1..4 | W1..4\_2 |
| 8 | Копії: xi = x, MCi = MC | КД |
| 9 | Рахунок MAH = MBH \* MCi + xi \* MKH |  |
| 10 | Якщо i=2,3,4: Сигнал T1 про завершення рахунку2 | S1\_3 |
| 11 | Якщо i=1: Чекати завершення рахунку2 в T2..4 | W2..4\_3 |
| 13 | Якщо i=1: Вивід MA |  |

## **2.3 Розробка схеми взаємодії процесів**

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

Використовуються такі засоби для синхронізації роботи потоків та вирішення задачі взаємного виключення:

1. S\_Calculation1EndIn — масив, що зберігає семафори для синхронізації по завершенню перших обчислень.
2. M\_CopyMc — мютекс для вирішення задачі взаємного виключення при копіюванні змінної MC.
3. CS\_UpdateX — критична секція для вирішення задачі взаємного виключення при зміні значення x.
4. CS\_CopyX — критична секція для вирішення задачі взаємного виключення при копіюванні x.
5. E\_InputFinishIn1 — подія для синхронізації по завершенню вводу в першому потоці.
6. E\_InputFinishIn3 — подія для синхронізації по завершенню вводу в третьому потоці.
7. E\_InputFinishIn4 — подія для синхронізації по завершенню вводу в четвертому потоці.
8. E\_Calculation2EndIn — масив, що зберігає події для синхронізації по завершенню других обчислень.

Схема взаємодії процесів зображена на рис. 2.2.

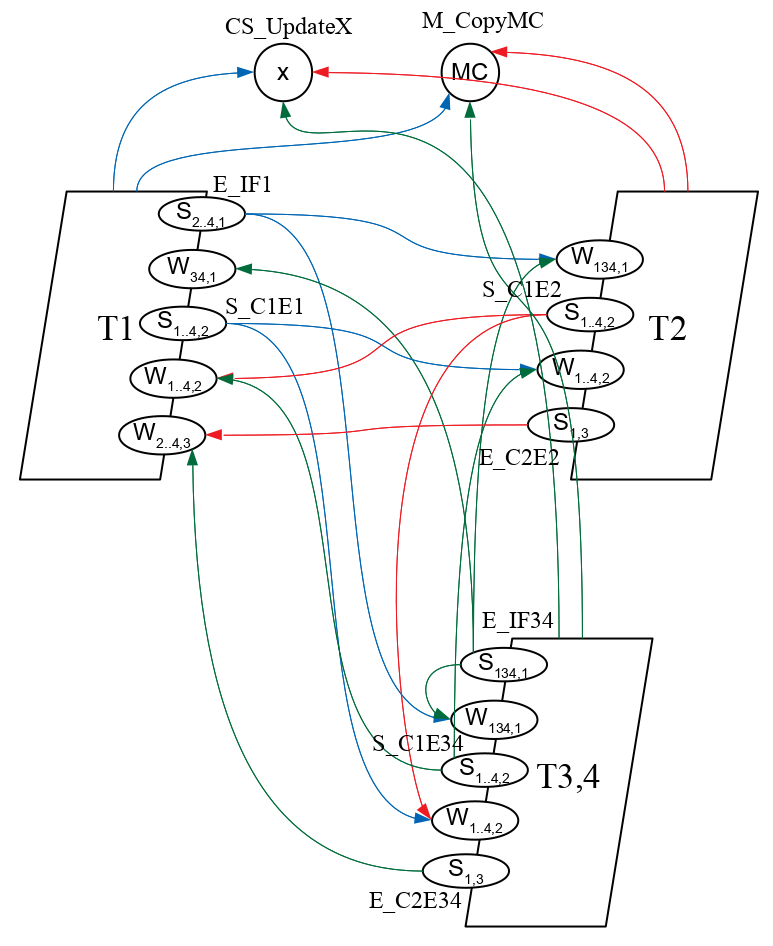


Рис. 2.2. Схема взаємодії процесів

## **2.4 Розробка програми ПРГ1**

Програма для ПКС з СП написана на мові програмування *С++* з використанням бібліотеки паралельного програмування *WinAPI*.

В бібліотеці *WinAPI* потоки створюються через задання потокової функції, якій можна передати аргументи у вигляді структури. В даному випадку таким чином передається *tid* потоку.

Програма складається з єдиного модуля main\_win.cpp. Алгоритм головної програми ПКС зі СП наведено у додатку А. Схема алгоритмів процесів для програми ПРГ1 наведено у додатку Б.

Використовувані змінні:

* N — розмірність даних.
* P — кількість потоків.
* H — розмірність даних, що буде оброблятися одним потоком.
* MA, MB, MC, MK — матриці, що приймають участь в обчисленні.
* Z — вектор, що приймає участь в обчисленні.

Створені типи:

* Vector — тип для зберігання вектору.
* Matrix — тип для зберігання матриці.

Створені методи:

1. output() — виведення значення матриці на екран.
2. fillVector() - заповнення вектора заданими значеннями.
3. fillMatrix() - заповнення матриці заданими значеннями.
4. threadFunctioin() - основна потокова функція.
5. 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 chrono.

Лістинг програми ПРГ1 наведено у додатку В.

## **2.5 Тестування програми ПРГ1**

## Метою тестування програми є замірювання часу, скільки виконується програма. Для тестування беруться три розмірності N = 600, 1200, 1800 та для кількості процесорів P = 1, 2, 3, 4.

## На основі часу роботи обчислюються такі коефіцієнти:

## Коефіцієнт прискорення *KП = Т1 / ТР .* Він показує, на скільки більше часу потрібно для виконання програми у системі з одним процесором у порівнянні з системою з кількістю процесорів, що дорівнює *Р*.

## Коефіцієнт ефективності *KЕ = КП / P \* 100%* . Він показує, на скільки відсотків використовуються *Р* процесорів системи.

## Тестування проводиться на машині з *Intel* *Core i5 3210M*, з базовою частотою *3.1 GHz* з операційною системою *Windows* *10*. Об’єм оперативної пам’яті — 4 ГБ.

## Результати тестування та розраховані коефіцієнти наведені у таблицях 2.2-2.4.

Таблиця 2.2 Час виконання програми для ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 600 | 5.91 | 3.15 | 2.47 | 1.89 |
| 1200 | 15.79 | 8.26 | 5.94 | 4.65 |
| 1800 | 54.76 | 28.3 | 20.02 | 15.86 |

Таблиця 2.3 Значення *КП* для ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 600 | 1 | 1.87 | 2.29 | 2.98 |
| 1200 | 1 | 1.91 | 2.65 | 3.39 |
| 1800 | 1 | 1.93 | 2.73 | 3.45 |

Таблиця 2.4 Значення *КЕ* для ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 600 | 100 | 93.80 | 76.55 | 74.62 |
| 1200 | 100 | 95.58 | 88.60 | 84.89 |
| 1800 | 100 | 96.74 | 91.17 | 86.31 |

На підставі таблиць 2.3 і 2.4 побудовані 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

Рис. 2.4 Програма ПРГ1. Графік зміни коефіцієнту ефективності *КЕ* в залежності від кількості ядер

**2.6 Висновки до розділу 2**

Виконано розробку програми зі спільною пам’яттю для математичної задачі, заданої по варіанту. Програма написана на *С++* з використанням бібліотеки *WinAPI*. Тестування програми показало наступне:

1. Використання багатоядерної ПКС призводить до скорочення часу виконання програми в кількість разів, що прямо пропорційна кількості ядер.
2. Значення коефіцієнта прискорення було мінімальним для *N* = 600 та *P* = 2 і дорівнювало 1.87.
3. Значення коефіцієнта прискорення було максимальним для *N* = 1800 та *P* = 4 і дорівнювало 3.45.
4. Швидкість росту коефіцієнта прискорення спадає при збільшенні кількості ядер.
5. Значення коефіцієнта ефективності було мінімальним для *N* = 600 та *P* = 4 і дорівнювало 78.17.
6. Значення коефіцієнта ефективності було максимальним для *N* = 1800 та *P* = 2 і дорівнювало 96.74.
7. Значення коефіцієнта ефективності спадає все більше для більшої кількості ядер. Це пояснюється тим, що при цьому збільшується кількість операцій по синхронізації роботи потоків між собою.
8. Коефіцієнт ефективності залишався достатньо великим навіть при роботі програми на чотирьох ядра, що свідчить про те, що задача добре підлягає розпаралеленню.

# РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС З ЛП

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

Математична задача: MA = MB \* MC + min(Z) \* MK.

Мова програмування: Ада.

Структура ПКС з ЛП: лінійна.

Засоби організації взаємодії процесів: механізм відправлення повідомлень.

Структура паралельної комп’ютерної системи зі спільною пам’яттю, 4 процесорами, 3 пристроями вводу-виводу представлена на рис. 3.1.

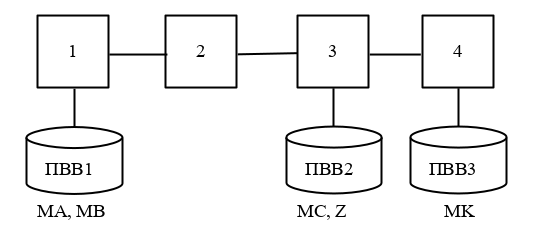


Рис 3.1. Структура ПКС з ЛП

## **3.1 Розробка паралельного математичного алгоритму**

1. xi = min(ZH), i=1..4
2. x = min(x, xi), i=1..4
3. MAH = MBH \* MC + x \* MKH

Позначення:

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

## **3.2 Розробка алгоритмів процесів**

**Задача T1**

1. Введення MB
2. Передати в Т2: MB3H
3. Прийняти з Т2: MC, ZH, MKH
4. x := min(ZH)
5. Передати в Т2: x
6. Прийняти з Т2: x\_in
7. x := x\_in
8. MAH := MBH \* MC + x \* MKH
9. Прийняти з Т2: MA3H
10. Вивести МА

**Задача T2**

1. Прийняти з Т1: MB3H
2. Прийняти з Т3: MC, Z2H, MK2H
3. Передати в Т3: MB2H
4. Передати в Т1: MC, ZH, MKH
5. x := min(ZH)
6. Прийняти з Т1: x\_in
7. x := min(x, x\_in)
8. Передати в Т3: x
9. Прийняти з Т3: x\_in
10. x := x\_in
11. Передати в Т1: x
12. MAH := MBH \* MC + x \* MKH
13. Прийняти з Т3: MA2H
14. Передати в Т1: MA3H

**Задача T3**

1. Введення MC, Z
2. Прийняти з Т4: MK3H
3. Передати в Т2: MC, Z2H,MK2H
4. Прийняти з Т2: MB2H
5. Передати в Т4: MBH, MC, ZH
6. x := min(ZH)
7. Прийняти з Т2: x\_in
8. x := min(x, x\_in)
9. Прийняти з Т4: x\_in
10. x := min(x, x\_in)
11. Передати в Т2: x
12. Передати в Т4: x
13. MAH := MBH \* MC + x \* MKH
14. Прийняти від Т4: MAH
15. Передати в Т2: MA2H

**Задача T4**

1. Введення MK
2. Передати в Т3: MK3H
3. Прийняти з Т3: MBH, MC, ZH
4. x := min(ZH)
5. Передати в Т3: x
6. Прийняти з Т3: x\_in
7. x := x\_in
8. MAH := MBH \* MC + x \* MKH
9. Передати в Т3: MAH

## **3.3 Розробка схеми взаємодії процесів**

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

Кожний потік, ввівши дані, якщо вони в нього є, одразу розсилає своїм сусідам. При складанні послідовності розсилки був взятий до уваги такий фактор, щоб зменшити очікування рандеву потоком. Тобто розсилка відбувається за найкоротший можливий час.

Для підрахунку глобального мінімуму локальні мінімуми збираються в третьому потоці, який обчислює загальний мінімум і відправляє його своїм сусідам, які потім розповсюдять його по всій системі.

Результат збираються в першому потоці та пересилається послідовно з права наліво.

Структурна схема взаємодії потоків наведена на рис. 3.2.

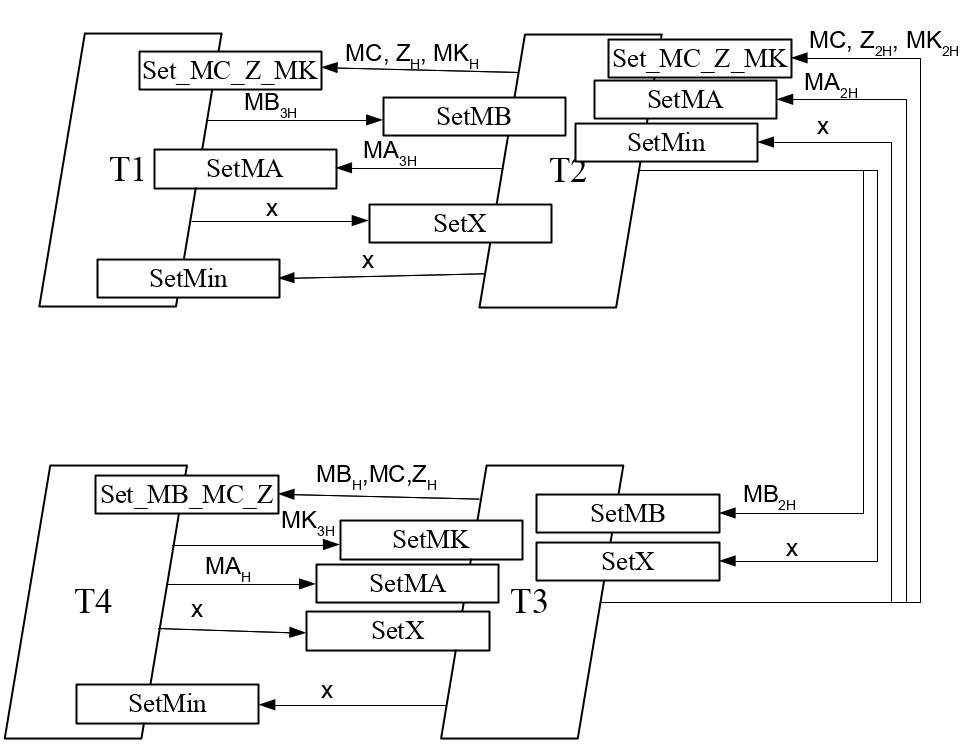


Рис 3.2. Схема взаємодії процесів

## **3.4 Розробка програми ПРГ2**

Програма для ПКС зі локальною пам’яттю реалізована на мові програмування Ада. В мові програмування Ада пересилка повідомлень виконується за допомогою механізму рандеву. В механізмі рандеву одна задача викликає захищений вхід іншої задачі. Ця інша задача приймає виклик захищеного входу, виконуючи необхідні операції над даними, що були передані через цей вхід.

Програма складається з основного модуля main, а також модуля data, що містить необхідні допоміжні функції та типи даних. Алгоритм головної програми ПКС з локальною пам’яттю наведено у додатку А. Схема алгоритмів процесів для програми ПРГ2 наведено у додатку Б.

Основні змінні:

1. N — розмірність даних.
2. P — кількість потоків.
3. H — розмірність даних, що буде оброблятися одним потоком.

Створені типи даних:

1. Vector — безрозмірний тип вектора.
2. Vector\_H — вектор розмірності *H*, що наслідується від типу вектора.
3. Vector\_2H — вектор розмірності 2 \* *H*, що наслідується від типу вектора.
4. Vector\_3H — вектор розмірності 3 \* *H*, що наслідується від типу вектора.
5. Vector\_N — вектор розмірності *N*, що наслідується від типу вектора.
6. Matrix — безрозмірний тип матриці.
7. Matrix\_H — матриця розмірності *H*, що наслідується від типу матриці.
8. Matrix\_2H — матриця розмірності 2 \* *H*, що наслідується від типу матриці.
9. Matrix\_3H — матриця розмірності 3 \* *H*, що наслідується від типу матриці.
10. Matrix\_N — матриця розмірності 4 \* *H*, що наслідується від типу матриці.

Створені методи:

1. Output\_Matrix() — виведення значення матриці на екран.
2. Fill\_Vector\_Ones() — заповнення вектора одиничними значеннями.
3. Fill\_Matrix\_Ones() — заповнення матриці одиничними значеннями.
4. Calc() - метод, що виконує основні обчислення.
5. Find\_Min() - метод, що виконує пошук мінімуму в заданому під-векторі.

Визначення часу роботи програми виконано за допомогою використання бібліотеки Ada.Calendar.

Лістинг програми ПРГ2 наведено у додатку В.

## **3.5 Тестування програми ПРГ2**

## Метою тестування програми є замірювання часу, скільки виконується програма. Для тестування беруться три розмірності N = 600, 1200, 1800 та для кількості процесорів P = 1, 2, 3, 4.

## На основі часу роботи обчислюються такі коефіцієнти:

## Коефіцієнт прискорення *KП = Т1 / ТР .* Він показує, на скільки більше часу потрібно для виконання програми у системі з одним процесором у порівнянні з системою з кількістю процесорів, що дорівнює *Р*.

## Коефіцієнт ефективності *KЕ = КП / P \* 100%* . Він показує, на скільки відсотків використовуються *Р* процесорів системи.

## Тестування проводиться на машині з *Intel* *Core i5 3210M*, з базовою частотою *3.1 GHz* з операційною системою *Windows* *10*. Об’єм оперативної пам’яті — 4 ГБ.

## Результати тестування та розраховані коефіцієнти наведені у таблицях 2.2-2.4.

Таблиця 2.2 Час виконання програми для ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 600 | 1.15 | 0.70 | 0.46 | 0.36 |
| 1200 | 14.39 | 7.66 | 5.45 | 4.21 |
| 1800 | 51.75 | 27.52 | 18.98 | 14.94 |

Таблиця 2.3 Значення *КП* для ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 600 | 1 | 1.64 | 2.5 | 3.19 |
| 1200 | 1 | 1.87 | 2.64 | 3.41 |
| 1800 | 1 | 1.88 | 2.72 | 3.46 |

Таблиця 2.4 Значення *КЕ* для ПРГ1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 600 | 100 | 82.14 | 83.33 | 79.86 |
| 1200 | 100 | 93.92 | 88.01 | 85.45 |
| 1800 | 100 | 94.02 | 90.88 | 86.59 |

На підставі таблиць 3.2 і 3.3 побудовані графіки поведінки *КП* і *КЕ* в залежності від *N* і *P* для ПРГ1.

На рис.3.3 наведено графік зміни коефіцієнту прискорення *КП* в залежності від кількості ядер, на рис.3.4 - графік зміни коефіцієнту ефективності *КЕ*.

Рис. 3.3 Програма ПРГ2. Графік зміни коефіцієнту прискорення *КП* в залежності від кількості ядер

Рис. 3.4 Програма ПРГ2. Графік зміни коефіцієнту ефективності *КЕ* в залежності від кількості ядер

**3.6 Висновки до розділу 3**

Виконано розробку програми із локальною пам’яттю по заданому алгоритму на мові програмування Ада. Тестування програми показало наступне:

1. Використання багатоядерної ПКС призводить до скорочення часу виконання програми в кількість разів, що прямо пропорційна кількості ядер.
2. Значення коефіцієнта прискорення було мінімальним для *N* = 600 та *P* = 2 і дорівнювало 1.64.
3. Значення коефіцієнта прискорення було максимальним для *N* = 1800 та *P* = 4 і дорівнювало 3.46.
4. Швидкість росту коефіцієнта прискорення спадає при збільшенні кількості ядер.
5. Значення коефіцієнта ефективності було мінімальним для *N* = 600 та *P* = 4 і дорівнювало 79.86.
6. Значення коефіцієнта ефективності було максимальним для *N* = 1800 та *P* = 2 і дорівнювало 94.02.
7. Значення коефіцієнта ефективності спадає все більше для більшої кількості ядер. Це пояснюється тим, що при цьому збільшується кількість операцій по синхронізації роботи потоків між собою.
8. Коефіцієнт ефективності залишався достатньо великим навіть при роботі програми на чотирьох ядра, що свідчить про те, що задача добре підлягає розпаралеленню.

**Основні результати і висновки по роботі**

1. Виконано аналіз потоків в мові програмування Python. Аналіз показав, що мова Python має всі необхідні інструменти для легкої та ефективної роботи з потоками. До них належать два види локу Lock та RLock, клас Semaphore для вирішення задачі взаємного виключення або синхронізації, клас Barrier для створення бар’єру для багатьох потоків, а також клас Event, що використовується для відправлення сигналів між потоками. Отже, мова програмування Python — сучасна зручна мова паралельного програмування.
2. В роботі було створена ПРГ1 з СП та ПРГ2 з ЛП. Аналізуючи результати тестування двох програм, можна дійти висновку, що в обох випадках спостерігається відносно великий коефіцієнт ефективності, навіть при чотирьох потоках, що свідчить про те, що дана математична задача добре підлягає розпаралеленню.
3. Коефіцієнт прискорення зменшувався зі зростанням кількості ядер і не досягав ідеального значення прискорення в *P* разів. Це пояснюється тим, що задана математична задача не є ідеально розпаралелюваною, а також зростанням кількості операцій, необхідних для синхронізації роботи потоків.
4. В системі з ЛП чи не найважливішим фактором, що впливає на загальний час, є розсилка даних по структурі. Саме тому важливо написати ефективний алгоритм розсилки, щоб вона відбувалася за мінімальну можливу кількість кроків.
5. ПРГ2 з ЛП дала дещо кращий час виконання для малої розмірності даних, порівняно з ПРГ1 с СП. Це пояснюється тим, що в ПРГ1 більший відсоток операцій припадає на синхронізацію роботи потоків за допомогою “важких” механізмів бібліотеки WinAPI. В той час як в ПРГ2 просто іде пересилка даних. Для інших розмірностей час виконання був приблизно однаковим.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Python[Електронний ресурс] – Режим доступу: <https://www.python.org/>
2. T[hreading](https://docs.python.org/3/library/threading.html" \l "module-threading) — Thread-based parallelism [Електронний ресурс] – Режим доступу: <https://docs.python.org/3/library/threading.html>
3. Жуков И., Корочкин А. Параллельные и распределенные вычисления. Лабораторный практикум. Учебно-методическое пособие. К.: «Корнейчук», 2008. – 224 с.
4. Python - Multithreaded Programming [Електронний ресурс] – Режим доступу: <https://www.tutorialspoint.com/python/python_multithreading.htm>
5. Едсгер Дейкстра [Електронний ресурс] — Режим доступу: <https://www.computer.org/web/awards/goode-edsger-dijkstra>
6. Multithreading in Python [Електронний ресурс] — Режим доступу: <https://medium.com/@nbosco/multithreading-vs-multiprocessing-in-python-c7dc88b50b5b>
7. Python Multithreading [Електронний ресурс] — Режим доступу: <https://www.toptal.com/python/beginners-guide-to-concurrency-and-parallelism-in-python>
8. Python Guide [Електронний ресурс] — Режим доступу: <http://www.techbeamers.com/python-multithreading-concepts/>
9. Parallelising Python with Threading and Multiprocessing [Електронний ресурс] — Режим доступу: <https://www.quantstart.com/articles/Parallelising-Python-with-Threading-and-Multiprocessing>
10. Python multiprocessing examples [Електронний ресурс] — Режим доступу: http://www.science.smith.edu/dftwiki/index.php/Python\_Multithreading/Multiprocessing\_Examples