

Екзаменаційний білет № 17

1. Пули потоків.
2. Стандарти та технології обміну повідомленнями в розподілених системах.
3. З використанням методів MPI обміну повідомленнями один до одного напишіть паралельний алгоритм сортування бакетами масиву символів.
4. Напишіть фрагмент коду для пулу 8 потоків, що виконують паралельне обчислення суми N елементів масиву дійсних чисел, розділяючи обчислення рекурсивно на задачі розміром не більше 100 елементів.

Затверджено на засіданні кафедри ІПІ
Протокол № 5 від « 8 » листопада 2023 р.

Зав. кафедри інформатики та програмної інженерії _____ Едуард ЖАРІКОВ

Виконав Мешков Андрій Ігорович групи ІП-15

1. Пули потоків

У теорії паралельних обчислень пули потоків відіграють ключову роль у підвищенні ефективності роботи багатопоточних програм. Пули потоків дозволяють зменшити накладні витрати, пов'язані зі створенням та знищенням великої кількості потоків, що особливо важливо для застосунків, які виконують багато короточасних завдань.

Пул потоків (Thread Pool) – це колекція попередньо створених потоків, які використовуються для виконання задач (tasks). Задача (Task) є об'єктом типу Runnable або Callable, призначеним для виконання окремої невеликої підзадачі.

Основні кроки створення та використання пулів потоків:

1. Створення задач: Необхідно визначити завдання, які будуть виконуватися потоками. Це можуть бути об'єкти типу Runnable або Callable. Runnable використовується для задач, які не повертають результат, тоді як Callable може повертати результат і може кидати виключення.

2. Створення пулу потоків: Пули потоків створюються за допомогою класів, які реалізують інтерфейси Executor та ExecutorService. Серед найпоширеніших реалізацій – ForkJoinPool, ScheduledThreadPoolExecutor та ThreadPoolExecutor. Фабрики пулів потоків можна знайти у класі Executors.

- ForkJoinPool

- Використовує алгоритм work-stealing для ефективного завантаження процесорних ядер.
- Підходить для задач типу divide-and-conquer.
- Оптимальний для рекурсивних алгоритмів та великих масивів даних.

- ScheduledThreadPoolExecutor

- Підтримує планування задач з фіксованою затримкою або періодичним виконанням.
- Ідеальний для регулярного виконання задач.
- Використовується для планування завдань та періодичного оновлення даних.

- ThreadPoolExecutor

- Гнучкий у налаштуванні розміру пулу та чергування задач.
- Підтримує corePoolSize, maximumPoolSize та keepAliveTime.
- Використовується для широкого спектру задач, що потребують контролю над багатопоточністю.

3. Завантаження задач у пул: Завдання додаються до пулу потоків через методи інтерфейсів Executor та ExecutorService. Найбільш використовувані методи – execute(), submit(), invokeAll() та invokeAny(). Метод execute() використовується для запуску задач типу Runnable, тоді як submit() може приймати як Runnable, так і Callable та повертає об'єкт типу Future, що дозволяє відстежувати стан виконання задачі.

4. Завершення роботи пулу потоків: Після завершення всіх задач пул потоків потрібно коректно завершити. Для цього використовуються методи інтерфейсу ExecutorService: shutdown(), shutdownNow() та awaitTermination().

Метод `shutdown()` ініціює м'яке завершення роботи, дозволяючи поточним задачам завершитися, тоді як `shutdownNow()` намагається негайно зупинити всі активні задачі.

5. Отримання результатів: Якщо використовувалися задачі типу `Callable`, можна отримати результати їх виконання через об'єкти типу `Future`, які повертаються методами `submit()`, `invokeAll()` та `invokeAny()`.

Використання пулів потоків дозволяє ефективно керувати обчислювальними ресурсами, знижуючи витрати на управління потоками та покращуючи загальну продуктивність багатопотокових програм.

2. Стандарти та технології обміну повідомленнями в розподілених системах.

Існують різні реалізації стандарту MPI та CORBA, які відіграють ключову роль у забезпеченні ефективного обміну повідомленнями в розподілених системах.

1. MPI (Message Passing Interface) – це стандарт для обміну повідомленнями між процесами в паралельних обчислювальних системах. Він широко використовується для створення високопродуктивних обчислювальних кластерів і суперкомп'ютерів.

Основні характеристики:

- Підтримує точковий і груповий обмін повідомленнями.
- Забезпечує синхронний та асинхронний обмін даними.
- Дозволяє масштабувати додатки на тисячі процесорів.

Використання в паралельних обчисленнях:

- Використовується для розподілу задач на кілька процесорів, забезпечуючи високий рівень продуктивності.
- Застосовується в наукових обчисленнях, моделюванні, аналізі великих даних і машинному навчанні.

2. CORBA (Common Object Request Broker Architecture) – це стандарт, розроблений Object Management Group (OMG), який дозволяє об'єктам, розташованим в різних середовищах, взаємодіяти один з одним. CORBA забезпечує платформно-незалежний механізм для віддаленого виклику методів.

Основні характеристики:

- ORB (Object Request Broker): Центральний компонент, який обробляє всі запити на виклик методів від одного об'єкта до іншого.
- IDL (Interface Definition Language): Мова для визначення інтерфейсів об'єктів, які можуть взаємодіяти через CORBA.
- Незалежність від мови програмування: CORBA підтримує багато мов програмування, включаючи C++, Java, Python тощо.

Основні компоненти:

- - ORB (Object Request Broker): Посередник для обміну запитами між об'єктами.
- - IDL (Interface Definition Language): Використовується для визначення інтерфейсів об'єктів.
- - POA (Portable Object Adapter): Управляє об'єктами-сервантами і забезпечує їхню життєдіяльність.

Використання в паралельних обчисленнях:

- Забезпечує інтеграцію різних компонентів розподіленої системи, що працюють на різних платформах і мовах програмування.
- Використовується в складних корпоративних системах, де необхідна висока ступінь взаємодії між різними програмними компонентами.

3. 3 використанням методів MPI обміну повідомленнями один до одного напишіть паралельний алгоритм сортування бакетами масиву символів.

```
from mpi4py import MPI
import numpy as np

def bucket_sort(data):
    # Знаходимо мінімальний і максимальний символи в даних
    min_value = ord(min(data))
    max_value = ord(max(data))

    # Ініціалізуємо бакети
    bucket_range = max_value - min_value + 1
    buckets = [[] for _ in range(bucket_range)]

    # Розподіляємо символи по бакетах
    for char in data:
        index = ord(char) - min_value
        buckets[index].append(char)

    # Об'єднуємо бакети
    sorted_data = []
    for bucket in buckets:
        sorted_data.extend(bucket)

    return sorted_data

def main():
    comm = MPI.COMM_WORLD # Створюємо комунікатор MPI
    rank = comm.Get_rank() # Отримуємо ранг поточного процесу
    size = comm.Get_size() # Отримуємо загальну кількість процесів
```

```

if rank == 0:
    # Вхідні дані в процесі з рангом 0
    data =
["p","a","r","a","l","l","e","l","c","o","m","p","u","t","i","n","g"]

    # Розділяємо дані між процесами
    chunks = np.array_split(data, size)

    # Відправляємо частини даних іншим процесам
    for i in range(1, size):
        comm.send(chunks[i], dest=i, tag=77)

    # Присвоюємо частину даних процесу з рангом 0
    local_data = chunks[0]
else:
    # Інші процеси отримують свою частину даних від процесу з рангом 0
    local_data = comm.recv(source=0, tag=77)

# Кожен процес сортує свою частину масиву
local_sorted_data = bucket_sort(local_data)

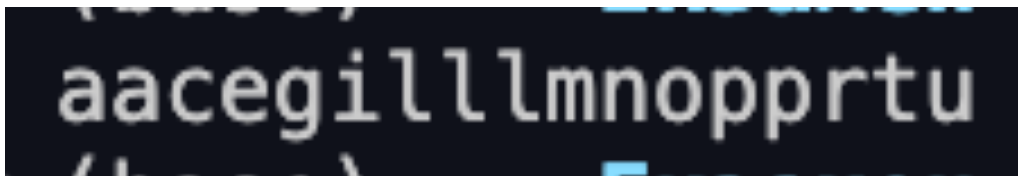
if rank == 0:
    # Процес з рангом 0 збирає відсортовані частини від інших процесів
    gathered_data = local_sorted_data
    for i in range(1, size):
        part = comm.recv(source=i, tag=77)
        gathered_data.extend(part)

    # Підсумкове сортування для забезпечення правильного порядку
    final_sorted_data = bucket_sort(gathered_data)

    # Виводимо остаточно відсортовані дані як рядок символів
    print(''.join(final_sorted_data))
else:
    # Інші процеси відправляють свої відсортовані дані процесу з рангом 0
    comm.send(local_sorted_data, dest=0, tag=77)

if __name__ == "__main__":
    main()

```



aacegilllmnopprt

4. Напишіть фрагмент коду для пулу 8 потоків, що виконують паралельне обчислення суми N елементів масиву дійсних чисел,

розділяючи обчислення рекурсивно на задачі розміром не більше 100 елементів.

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class ParallelSum {

    // Рекурсивна задача для обчислення суми
    static class SumTask extends RecursiveTask<Double> {
        private static final int THRESHOLD = 100;
        private final double[] array;
        private final int start;
        private final int end;

        public SumTask(double[] array, int start, int end) {
            this.array = array;
            this.start = start;
            this.end = end;
        }

        @Override
        protected Double compute() {
            int length = end - start;
            if (length <= THRESHOLD) {
                // Пряме обчислення суми для невеликих масивів
                double sum = 0;
                for (int i = start; i < end; i++) {
                    sum += array[i];
                }
                return sum;
            } else {
                // Розділення задачі на дві підзадачі
                int mid = start + length / 2;
                SumTask leftTask = new SumTask(array, start, mid);
                SumTask rightTask = new SumTask(array, mid, end);

                // Запускаємо підзадачі
                leftTask.fork();
                double rightResult = rightTask.compute();
                double leftResult = leftTask.join();

                // Комбінуємо результати
                return leftResult + rightResult;
            }
        }
    }

    public static double parallelSum(double[] array, int n) {
        ForkJoinPool pool = new ForkJoinPool(8);
        SumTask task = new SumTask(array, 0, n);
    }
}
```

```

        return pool.invoke(task);
    }

    public static void main(String[] args) {
        // Приклад використання
        double[] array = new double[10000];
        for (int i = 0; i < array.length; i++) {
            array[i] = Math.random();
        }

        int n = 2345;
        double sum = parallelSum(array, n);
        System.out.println("Sum of first " + n + " elements: " + sum);
    }
}

```

```

Sum of first 2345 elements: 1205.0710060727802

```