Київський національний університет імені Тараса Шевченка

ЗВІТ

до лабораторної роботи

з дисципліни "Хмарні обчислення"

на тему:

Паралельна реалізація алгоритму решета Ератосфена з використанням PARCS та Google Cloud

Виконав

Студент четвертого курсу

Групи ТТП-41

Факультету комп'ютерних наук та кібернетики

Назарій ЯГОТІН

Київ, 2025

Зміст

1 Постановка задачі

1.1 Мета роботи

Метою даної лабораторної роботи є розробка та реалізація паралельного алгоритму решета Ератосфена з використанням системи PARCS (Parallel Computing System) та хмарної платформи Google Cloud.

1.2 Завдання

1. Реалізувати паралельний алгоритм решета Ератосфена на мові Python
2. Розгорнути обчислювальний кластер на Google Cloud
3. Провести експерименти з різною кількістю обчислювальних вузлів
4. Проаналізувати ефективність паралелізації

1.3 Теоретичні відомості

Решето Ератосфена — це алгоритм для знаходження всіх простих чисел до заданого числа n. Алгоритм працює наступним чином:

1. Створюється список чисел від 2 до n
2. Береться найменше незакреслене число (воно є простим)
3. Закреслюються всі числа з діапазону, кратні незакресленому
4. Повторюються кроки 2-3 поки не будуть опрацьовані всі числа

Часова складність класичного алгоритму: $O(n \log \log n)$

PARCS — це система для паралельних обчислень, яка дозволяє розподіляти обчислювальні задачі між декількома вузлами в мережі.

2 Пояснення та обґрунтування паралелізації алгоритму

2.1 Сегментне решето Ератосфена

Для ефективної паралелізації використовується **сегментне решето Ератосфена**, яке базується на наступній математичній властивості:

Якщо число n є складеним, то воно має простий дільник $p \leq \sqrt{n}$

2.2 Схема паралелізації

Алгоритм розділяється на три етапи:

2.2.1 Етап 1: Послідовний пошук базових простих чисел

Спочатку послідовно знаходяться всі прості числа до $\sqrt{n}$:

$$\text{small\_primes} = \text{sieve}([2, \lfloor\sqrt{n}\rfloor]) \tag{1}$$

Ці числа будуть використані для відсіювання на всіх workers.

2.2.2 Етап 2: Розподіл роботи (MAP)

Діапазон $[\sqrt{n}+1, n]$ розділяється на k сегментів, де k — кількість workers:

$$\text{segment\_size} = \frac{n - \sqrt{n}}{k} \tag{2}$$

Кожен worker отримує свій сегмент:

$$\text{segment}_i = [\sqrt{n} + 1 + i \cdot \text{segment\_size}, \sqrt{n} + 1 + (i+1) \cdot \text{segment\_size}] \tag{3}$$

2.2.3 Етап 3: Паралельна обробка сегментів

Кожен worker незалежно обробляє свій сегмент, використовуючи базові прості числа для відсіювання:

1. Створюється масив булевих значень для сегмента
2. Для кожного простого p з базового списку:
   - Знаходиться перше кратне p в сегменті
   - Всі кратні позначаються як складені
3. Повертається список простих чисел із сегмента

2.2.4 Етап 4: Збір результатів (REDUCE)

Результати від усіх workers об'єднуються з базовими простими числами:

$$\text{result} = \text{small\_primes} \cup \bigcup_{i=1}^{k} \text{segment\_result}_i \tag{4}$$

2.3 Теоретична складність

- **Послідовна версія**: $T_{\text{seq}} = O(n \log \log n)$
- **Паралельна версія**: $T_{\text{par}} = O(\sqrt{n} \log \log \sqrt{n}) + O(\frac{n \log \log n}{k})$
- **Теоретичне прискорення**: $S = \frac{T_{\text{seq}}}{T_{\text{par}}} \approx k$ при великих n

2.4 Обґрунтування ефективності

Паралелізація є ефективною завдяки таким особливостям:

1. **Незалежність сегментів** — кожен worker обробляє свій діапазон без синхронізації
2. **Мінімальна комунікація** — дані передаються тільки на початку (базові прості) та в кінці (результати)
3. **Збалансоване навантаження** — всі сегменти приблизно однакового розміру
4. **Масштабованість** — можливість додавання довільної кількості workers

3 Розбір коду

3.1 Структура програми

Програма складається з класу `Solver`, який містить всі необхідні методи для виконання паралельних обчислень.

3.2 Метод ініціалізації

```
def __init__(self, workers=None, input_file_name=None,
             output_file_name=None):
    self.input_file_name = input_file_name
    self.output_file_name = output_file_name
    self.workers = workers
```

Метод приймає список workers від PARCS та шляхи до файлів вводу/виводу.

3.3 Головний метод solve

```
def solve(self):
    n = self.read_input()
    sqrt_n = int(math.sqrt(n)) + 1
    small_primes = self.sequential_sieve(sqrt_n)

    work_range = n - sqrt_n
    step = work_range // len(self.workers)

    mapped = []
    for i in range(len(self.workers)):
        start = sqrt_n + 1 + i * step
        end = sqrt_n + 1 + (i + 1) * step if i < len(self.workers) - 1 else n
        mapped.append(self.workers[i].mymap(start, end, small_primes))

    all_primes = self.myreduce(mapped, small_primes)
    self.write_output(all_primes)
```

Цей метод координує весь процес обчислень:

1. Читає вхідне значення n
2. Обчислює базові прості числа до $\sqrt{n}$
3. Розподіляє діапазон між workers
4. Викликає метод `mymap` на кожному worker
5. Збирає та об'єднує результати

3.4 Послідовне решето

```
@staticmethod
@expose
def sequential_sieve(limit):
    if limit < 2:
        return []

    is_prime = [True] * (limit + 1)
    is_prime[0] = is_prime[1] = False
```

```
    for i in range(2, int(math.sqrt(limit)) + 1):
        if is_prime[i]:
            for j in range(i * i, limit + 1, i):
                is_prime[j] = False

    return [i for i in range(2, limit + 1) if is_prime[i]]
```

Класична реалізація решета Ератосфена для знаходження базових простих чисел.

3.5 Метод обробки сегмента (MAP)

```
@staticmethod
@expose
def mymap(start, end, small_primes):
    size = end - start + 1
    is_prime = [True] * size

    for prime in small_primes:
        first_multiple = ((start + prime - 1) // prime) * prime
        if first_multiple == prime:
            first_multiple += prime

        current = first_multiple
        while current <= end:
            is_prime[current - start] = False
            current += prime

    primes = []
    for i in range(size):
        num = start + i
        if is_prime[i] and num > 1:
            primes.append(num)

    return primes
```

Ключові моменти:

- Декоратор `@expose` дозволяє викликати метод віддалено через PARCS
- Використовується компактний масив тільки для сегмента
- Формула `((start + prime - 1) // prime) * prime` ефективно знаходить перше кратне

3.6 Метод збору результатів (REDUCE)

```
@staticmethod
@expose
def myreduce(mapped, small_primes):
    all_primes = small_primes[:]

    for result in mapped:
        all_primes.extend(result.value)

    all_primes.sort()
    return all_primes
```

Об'єднує базові прості числа з результатами від усіх workers та сортує фінальний список.

3.7 Методи роботи з файлами

```
def read_input(self):
    with open(self.input_file_name, 'r') as f:
        n = int(f.readline().strip())
    return n

def write_output(self, primes):
    with open(self.output_file_name, 'w') as f:
        f.write(', '.join(map(str, primes)))
        f.write('\n')
```

Прості методи для читання вхідного значення n та запису результату у файл.

4 Результати виконання програми

4.1 Налаштування експерименту

Апаратне забезпечення:

- Платформа: Google Cloud Platform
- Master: e2-medium (2 vCPU, 4 GB RAM)
- Workers: e2-small (2 vCPU, 2 GB RAM)
- Регіон: europe-central2-a (Warsaw)

Програмне забезпечення:

- PARCS (Docker image: hummer12007/parcs-node)
- Python 3.10

4.2 Результати вимірювань

Табл. 1: Час виконання (секунди) залежно від кількості *workers* та розміру вхідних даних n

n	**Кількість workers**						
	1	**2**	**3**	**4**	**5**	**6**	**7**
10^6	<1	<1	<1	<1	<1	<1	<1
10^7	9	5	4	4	4	4	4
5×10^7	37	22	19	17	15	15	14

5 Висновки

У ході виконання лабораторної роботи було реалізовано паралельний алгоритм решета Ератосфена з використанням сегментного підходу та системи PARCS на платформі Google Cloud.

Проведено експериментальне дослідження залежності часу виконання від кількості обчислювальних вузлів для різних розмірів задачі (10^6, 10^7, 10^8).

Результати експериментів показують, що паралелізація алгоритму дозволяє суттєво скоротити час обчислень. Зі збільшенням кількості workers спостерігається зменшення часу виконання, що підтверджує ефективність обраного підходу до розпаралелювання.

Сегментний метод виявився придатним для хмарних обчислень завдяки мінімальним витратам на комунікацію між вузлами та можливості незалежної обробки сегментів даних.

Повний код алгоритму можна знайти за посиланням:
https://github.com/8ctag8ne/cloud-tech-lab.