MODUL 6
Pemrograman Paralel Berbasis Shared Memory dengan Python (multiprocessing)

A. Materi

1. Shared Memory

   Dalam shared memory, semua *worker* (thread/proses) dapat mengakses area memori yang sama, sehingga data tidak perlu selalu disalin antar proses. Contohnya satu matriks besar dibagi beberapa baris, masing-masing proses mengerjakan sebagian baris kemudian hasil disimpan dalam satu struktur data bersama. Shared memory adalah model memori di mana beberapa prosesor (CPU core) bisa langsung membaca dan menulis ke alamat memori yang sama. Namun berbeda dengan arsitektur distributed memory, di mana setiap prosesor punya memori sendiri dan harus bertukar data melalui pesan (MPI misalnya). Shared memory adalah area memori yang bisa diakses langsung oleh beberapa prosesor/core. Dalam multiprosesor, memungkinkan proses untuk berbagi data tanpa membuat copy terpisah. Karakteristik dari Shared Memory adalah sebagai berikut:

   - Semua core mengakses memori secara global.
   - Cepat untuk komunikasi karena tidak perlu serialisasi.
   - Race condition mungkin terjadi jika beberapa core menulis sekaligus sehingga perlu sinkronisasi.
   - Cocok untuk masalah dengan data yang besar tapi sering dibaca/tulis oleh banyak core.

   Fujimoto (2000) menekankan pentingnya shared memory dalam simulasi paralel, seperti Time Warp, karena mengurangi overhead komunikasi antar prosesor dibandingkan distributed memory. Fujimoto dan Taniar menekankan bahwa shared memory cocok untuk multiprosesor dengan komunikasi cepat, sedangkan distributed memory lebih kompleks karena harus explicit message passing.

2. Multiprocesor

   Multiprosesor adalah sistem komputer yang memiliki lebih dari satu CPU/core yang bisa menjalankan instruksi secara paralel. Multiprosesor bisa berbagi satu sistem memori (shared memory) atau memiliki memori sendiri (distributed memory). Python tidak otomatis menggunakan shared memory ketika menggunakan multiprocessing. Setiap proses mendapat salinan data sendiri. Agar benar-benar shared memory ada beberapa hal yang bisa dilakukan:

   1. Gunakan Value untuk variabel tunggal.
   2. Gunakan Array untuk array atau buffer.
   3. Gunakan Lock untuk sinkronisasi.

   Kelebihan dalam menggunakan multiprosesor untuk shared memory adalah untuk akses yang lebih cepat antar core, tidak perlu menyalin data besar terutama digunakan untuk simulasi paralel atau database paralel. Namun akan muncul kondisi "race condition" dimana hal tersebut perlu sinkronisasi, selain itu jika terlalu banyak core akan memicu kotensi memori dan tidak cocok digunakan untuk cluster jauh atau sistem terdistribusi, karena semua prosesor harus dekat secara fisik.

B. Langkah Praktikum

Kasus : Menghitung jumlah kuadrat dari sejumlah besar angka dengan membandingkan serial dan paralel menggunakan multiprocessor.

1. Perhatikan gambar 1, pada gambar program melakukan looping sekuensial satu iterasi dijalankan setelah iterasi sebelumnya selesai. Pada program karena jumlah iterasi besar (50 juta) maka waktu eksekusi akan lama

multiprocesorSerial.py > ...

```python
import time

def heavy_loop(n):
    total = 0
    for i in range(n):
        total += i * i
    return total

if __name__ == "__main__":
    N = 50_000_000   # jumlah looping besar
    start = time.time()
    result = heavy_loop(N)
    end = time.time()
    print(f"Hasil: {result}")
    print(f"Waktu eksekusi (Sekuensial): {end - start:.2f} detik")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
/usr/local/bin/python3 /Users/yuliana/multiprocesorSerial.py
yuliana@MacBook-Air-Yuliana ~ % /usr/local/bin/python3 /Users/yuliana/multiprocesorSerial.py
Hasil: 41666665416666675000000
Waktu eksekusi (Sekuensial): 1.21 detik
yuliana@MacBook-Air-Yuliana ~ %
```

Gambar 1 multiproses Serial

```python
import time
import multiprocessing as mp

def heavy_loop_range(start_end):
    start, end = start_end
    total = 0
    for i in range(start, end):
        total += i * i
    return total

if __name__ == "__main__":
    N = 50_000_000
    core = 4  # Jumlah core yang dipakai (bisa diubah)
    pool = mp.Pool(processes=core)

    chunk = N // core
    ranges = [(i * chunk, (i + 1) * chunk) for i in range(core)]

    start = time.time()
    results = pool.map(heavy_loop_range, ranges)
    total_sum = sum(results)
    end = time.time()

    pool.close()
    pool.join()

    print(f"Hasil: {total_sum}")
    print(f"Waktu eksekusi (Paralel, {core} core): {end - start:.2f} detik")
```

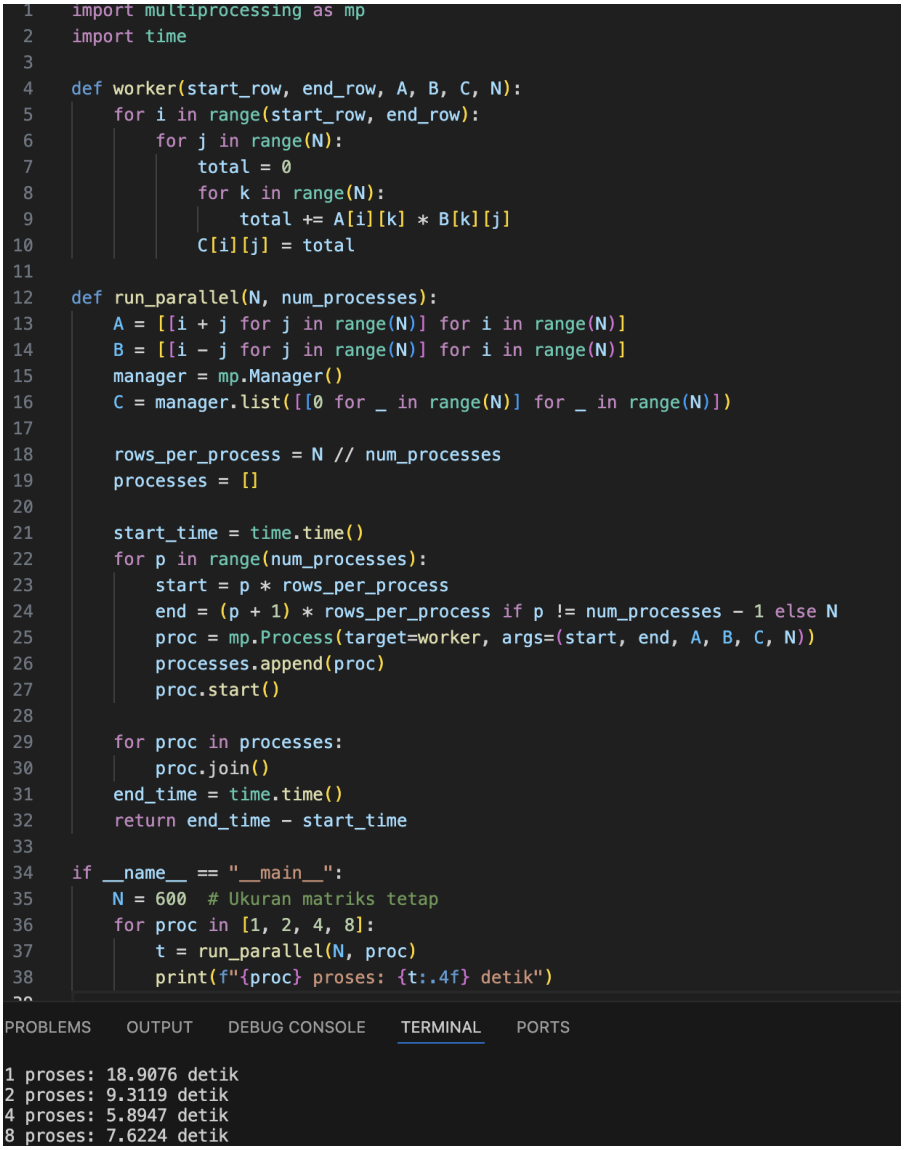
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
/usr/local/bin/python3 /Users/yuliana/multiprocesorparalel.py
yuliana@192 ~ % /usr/local/bin/python3 /Users/yuliana/multiprocesorparalel.py
Hasil: 41666665416666675000000
Waktu eksekusi (Paralel, 4 core): 0.41 detik
```

gambar 2. Multiprosesor Paralel

2. Jumlah Prosesor dan Waktu.

Perhatikan gambar 3. Kode pada program melakukan perkalian matriks NxN secara paralel menggunakan modul multiprocessing di Python. Matriks A dan B dibuat sebagai contoh, dan hasilnya C disimpan dalam Manager.list agar bisa diakses bersama oleh semua proses. Fungsi worker menghitung elemen-elemen C untuk baris tertentu yang dibagi rata ke tiap proses. run_parallel membuat sejumlah proses sesuai num_processes, masing-masing memproses sebagian baris matriks, kemudian menunggu semua proses selesai dengan join(). Waktu eksekusi dihitung menggunakan time.time(), sehingga kita bisa membandingkan performa perkalian matriks dengan 1, 2, 4, atau 8 proses, memanfaatkan paralelisme untuk mempercepat komputasi dibandingkan eksekusi sekuensial.



```python
import multiprocessing as mp
import time

def worker(start_row, end_row, A, B, C, N):
    for i in range(start_row, end_row):
        for j in range(N):
            total = 0
            for k in range(N):
                total += A[i][k] * B[k][j]
            C[i][j] = total

def run_parallel(N, num_processes):
    A = [[i + j for j in range(N)] for i in range(N)]
    B = [[i - j for j in range(N)] for i in range(N)]
    manager = mp.Manager()
    C = manager.list([[0 for _ in range(N)] for _ in range(N)])

    rows_per_process = N // num_processes
    processes = []

    start_time = time.time()
    for p in range(num_processes):
        start = p * rows_per_process
        end = (p + 1) * rows_per_process if p != num_processes - 1 else N
        proc = mp.Process(target=worker, args=(start, end, A, B, C, N))
        processes.append(proc)
        proc.start()

    for proc in processes:
        proc.join()
    end_time = time.time()
    return end_time - start_time

if __name__ == "__main__":
    N = 600  # Ukuran matriks tetap
    for proc in [1, 2, 4, 8]:
        t = run_parallel(N, proc)
        print(f"{proc} proses: {t:.4f} detik")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
1 proses: 18.9076 detik
2 proses: 9.3119 detik
4 proses: 5.8947 detik
8 proses: 7.6224 detik
```

Gambar 3. Prosesor dan waktu

3. Shared Memory dengan Multiprocessing

   Kode pada gambar 5 melakukan paralelisasi perhitungan jumlah kuadrat dari 0 hingga 50 juta menggunakan modul multiprocessing dengan membagi tugas ke beberapa core CPU.

Dengan menambah jumlah core dari 1, 2, 4 hingga 8, waktu eksekusi menurun secara signifikan, menunjukkan speedup akibat pembagian beban kerja. Namun, meskipun total core tersedia cukup banyak, peningkatan kinerja tidak selalu linear karena ada overhead pembuatan proses, pembagian data, dan penggabungan hasil, sehingga efisiensi menurun pada jumlah core tinggi. Hal ini menegaskan prinsip bahwa paralelisasi efektif hingga batas tertentu, setelah itu overhead mulai membatasi keuntungan tambahan.

```python
import time
import multiprocessing as mp
import os

# Fungsi looping berat
def heavy_loop_range(start_end):
    start, end = start_end
    total = 0
    for i in range(start, end):
        total += i * i
    return total

if __name__ == "__main__":
    N = 50_000_000  # jumlah iterasi looping
    total_core = os.cpu_count()  # total core di sistem
    print(f"Total core CPU tersedia: {total_core}\n")

    # Jumlah core yang akan diuji
    core_list = [1, 2, 4, 8] if total_core >= 8 else [1, 2, total_core]

    for core in core_list:
        # Bagi tugas looping ke beberapa core
        chunk = N // core
        ranges = [(i * chunk, (i + 1) * chunk) for i in range(core)]

        # Jalankan multiprocessing
        pool = mp.Pool(processes=core)
        start_time = time.time()
        results = pool.map(heavy_loop_range, ranges)
        total_sum = sum(results)
        end_time = time.time()

        pool.close()
        pool.join()

        print(f"Core: {core} | Waktu eksekusi: {end_time - start_time:.2f} detik | Hasil: {total_sum}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Total core CPU tersedia: 10

Core: 1 | Waktu eksekusi: 1.30 detik | Hasil: 41666665416666675000000
Core: 2 | Waktu eksekusi: 0.68 detik | Hasil: 41666665416666675000000
Core: 4 | Waktu eksekusi: 0.45 detik | Hasil: 41666665416666675000000
Core: 8 | Waktu eksekusi: 0.35 detik | Hasil: 41666665416666675000000
```

Gambar 5. Shared Memory