Wrocław, 29 października 2024

Programowanie **r**ównoległe i **r**ozproszone

Laboratorium 3

Programowanie rozproszone

prowadzący: *Dr inż. Radosław Idzikowski*

---

1 Wprowadzenie

Celem laboratorium jest wprowadzenie do obliczeń rozproszonych na przykładzie przetwarzania obrazów w językach `Python` oraz `C#`. Do obliczeń zostanie wykorzystanych więcej niż jeden komputer w tej samej sieci. Czas przewidziany na wykonanie zadania to termin dwóch zajęć wraz z ocenieniem pracy. Praca będzie oceniana na bieżąco w trakcie zajęć wraz z postępem wykonywania kolejnych przykładów. **Po ukończeniu każdego zadania należy zawołać prowadzącego w celu zaakceptowania etapu i odnotowania postępów.**

2 Zadania

W ramach zajęć należy w zespołach wykonać następujące zadania:

1. Podstawowa aplikacja w języku `Python`.
2. Pierwszy system rozproszony.
3. Klient w innym języku.

*. Akceleracja `CUDA`.

Za wykonanie zadania nr 1 jest ocena dostateczna, za każde kolejne zadanie jest +1 do oceny. Na ocenę bardzo dobrą (5.0) należy wykonać wszystkie trzy zadania. Zadanie oznaczone gwiazdką jest dla chętnych. Po zakończeniu zajęć należy kody źródłowe napisanych programów przesłać do prowadzącego.

3 Opis zadań

3.1 Zadanie 1

W ramach tego etapu należny napisać aplikację w języku `Python` do równoległego przetwarzania obrazu. Przed przystąpieniem do zadania trzeba zainstalować niezbędne biblioteki.

```
python -m pip install --upgrade pip
pip install pillow
pip install numpy
```

Zasadniczym celem zadania jest wczytanie pliku graficznego (`Image.open(image_path)`), a następnie podzielenie go na mniejsze fragmenty. Każdy fragment obrazu ma zostać niezależnie przetworzony w osobnym procesie, np.: poprzez zastosowanie filtru krawędziowego (wymagana jest własna implementacja z wykorzystaniem `NumPy`). Na koniec wszystkie fragmenty należy scalić w jeden obraz wyjściowy.

`Pool` to funkcja w języku `Python`, która ułatwia równoległe przetwarzanie danych poprzez automatyczne zarządzanie procesami w puli. Działa w oparciu o tworzenie i kontrolowanie grupy

procesów, które mogą wykonywać zadania równocześnie, dzięki czemu przyspiesza operacje wymagające intensywnego przetwarzania. Przykład prostego zastosowania pokazano na listingu 1. `Poll(4)` tworzy pulę czterech procesów, które mogą być wykonywane równocześnie. Fragment kodu: `p.map(square, numbers)` równolegle stosuje funkcję `square` do każdego elementu listy `numbers`. Ponadto `map` automatycznie dzieli listę `numbers` na fragmenty i przypisuje je do procesów, które wyliczają kolejne potęgi.

```
from multiprocessing import Pool
def square(x):
    return x * x
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
with Pool(4) as p:
    results = p.map(square, numbers)
```

Listing 1: Przykład działania `Pool`

W zadaniu można użyć m.in. filtr *Sobela*, który jest jednym z najczęściej używanych detektorów krawędzi. Opiera się on na splocie obrazu z małym, rozdzielnym filtrem (maską) o wartościach całkowitych w kierunku poziomym i pionowym, a zatem jest stosunkowo niedrogi pod względem obliczeń. Istotną zaletą wspominanego filtru jest zapewnione różnicowanie oraz jednocześnie wygładzanie. Warto jednak użyć bardziej złożonych operacji, aby wykorzystać dostępne narzędzia, żeby przyśpieszenie było bardziej widoczne.

3.2 Zadanie 2

Zadanie polega na implementacji dwóch oddzielnych aplikacji: aplikacji głównej (serwera) oraz aplikacji klienckiej. Celem jest wykorzystanie połączeń sieciowych `sockets` do rozdzielenia fragmentów obrazu na różne maszyny. Aplikacja główna zarządza podziałem obrazu oraz wysyła fragmenty do klientów, którzy następnie przetwarzają je (stosując np. filtr krawędziowy ten sam co w poprzednim zadaniu) i zwracają wyniki do serwera. Finalnie, aplikacja główna scala przetworzone fragmenty w jeden obraz wyjściowy. Do przesyłania danych między aplikacjami można skorzystać z funkcji zamieszczonych w listingach. Funkcja *send_all(sock, data)* (listing 2) wysyła kompletne dane przez gniazdo, najpierw przekazując rozmiar danych, a potem przesyłając je w całości, co zapobiega ucięciom danych.

```
def send_all(sock, data):
    data = pickle.dumps(data)
    sock.sendall(len(data).to_bytes(4, byteorder='big'))
    sock.sendall(data)
```

Listing 2: Funkcja wysyła pełne dane przez `socket`

Funkcja *receive_all(sock)* (listing 3) odbiera pełne dane przez gniazdo (`socket`), w pierwszej kolejności pobierając rozmiar danych, a następnie odbierając je w częściach aż do uzyskania pełnego komunikatu.

```
def receive_all(sock):
    length = int.from_bytes(sock.recv(4), byteorder='big')
    data = b''
    while len(data) < length:
        packet = sock.recv(4096)
        if not packet:
            break
        data += packet
    return pickle.loads(data)
```

Listing 3: Funkcja odbiera pełne dane przez `socket`

Struktura aplikacji klienta jest bardzo prosta, co zostało pokazane na listingu 4. W pierwszej kolejności należy utworzyć gniazdo. Następnie metodą `connect` możemy połączyć się z serwerem (jeśli jest w trybie nasłuchiwania) podając jego `IP` oraz wcześniej zdefiniowany port. Potem wystarczy już pobrać fragment danych do przetworzenia, wywołać funkcje do przetwarzania danych oraz przesłać. Na koniec należy zamknąć połączenie.

```python
def client_main():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('192.168.1.3', 2040))
    fragment = receive_all(client_socket)
    processed_fragment = edge_filter(fragment)
    send_all(client_socket, processed_fragment)
    client_socket.close()
    print("Fragment przetworzony i wysłany z powrotem do serwera")
```

Listing 4: Kod bazowy klienta

Struktura aplikacji głównej jest trochę bardziej złożona. Podstawy kod został zamieszczony w listingu 5, zawiera on niezbędne instrukcje do utworzenia gniazda oraz wywołanie nasłuchiwania. Rzeczywisty adres serwera będzie się różnił od tego w kodzie. Samodzielnie należy napisać funkcje do odpowiedniego dzielenia obrazu oraz jego scalania.

```python
def server_main(image_path, n_clients):
    image = Image.open(image_path)
    fragments = split_image(image, n_clients)
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('192.168.1.3', 2040))
    server_socket.listen(n_clients)
    print("Serwer nasłuchuje...")
    processed_fragments = []
    for i in range(n_clients):
        client_socket, client_adress = server_socket.accept()
        print(f"Połączono z klientem {i+1}: {client_adress}")
        send_all(client_socket, fragments[i])
        processed_fragment = receive_all(client_socket)
        processed_fragments.append(processed_fragment)
        client_socket.close()
    result_image = merge_image(processed_fragments)
    result_image.save("processed_image.png")
    print("Obraz przetworzony zapisany jako processed_image.png")
```

Listing 5: Kod bazowy serwera

3.3 Zadanie 3

Głównym celem tego zadania jest stworzenie aplikacji w języku `C#`, która będzie pełniła te same funkcje co aplikacja kliencka w języku `Python`. Dla przypominana, aplikacja ma za zadanie łączyć się z serwerem, odbierać fragment obrazu, stosować filtr krawędziowy (np. *Sobela*), a następnie przesyłać przetworzony fragment obrazu z powrotem do serwera. Do nawiązania połączenia będzie niezbędna klasa `TcpClient`.

3.4 Zadanie *

Jeśli w ramach "*klastra obliczeniowego*" któryś z węzłów (w tym wypadku komputer) będzie dysponował kartą graficzną to można zrobić implementację wykorzystującą technologię `CUDA`. Aplikacją powinna działać niezależnie od tego czy jest `GPU`, więc należny to poprawnie obsłużyć.