Sprawozdanie – programowanie równolegle i rozproszone

Ćwiczenie 3 – rozproszone sieciowo procesy

Adrian Nowosielski, Cezary Skorupski

1. Zarys poruszonego problemu

Problemem poruszonym w tym ćwiczeniu jest wykorzystanie rozproszonych sieciowych procesów do rozwiązania problemu mnożenia macierzy przez wektor. Do implementacji rozwiązania użyliśmy języka Python wraz z biblioteką multiprocessing, która pozwoliła na stworzenie serwera i następnie kolejek, do których mogliśmy wysyłać poszczególne procesy. Dzięki takiemu rozwiązaniu możemy wykorzystać Python do obliczeń równoległych na kilku maszynach(serwerach) w celu szybszego rozwiązania problemów inżynieryjnych.

2. Sposób implementacji w języku Python

Do rozwiązania postawionego problemu wykorzystaliśmy język Python oraz bibliotekę multiprocessing. Na początku implementacji stworzyliśmy serwer:

2.1 – Sposób implementacji serwera

Jeśli chodzi o sposób implementacji serwera, to jest to najprostszy element programu. Jako elementy wejściowe przyjmujemy adres_ip, na którym ma działać serwer, port oraz klucz autoryzacyjny. Następnie rejestrujemy na serwerze dwie kolejki – rezultatów oraz zadań, które będziemy potem wykorzystywać do wysyłania zadań do poszczególnych robotników oraz do zarządzania procesami. Przykładowy kod z klasy MultiplicationServer prezentuje się następująco:

```python
class MatrixMultiplicationServer:
    def __init__(self, ip_address: str = "127.0.0.1", port: int = 8080, authorization_key: bytes = b"root"):
        self.__server_manager = QueueManager(address=(ip_address, port), authkey=authorization_key)
        self.__create_queues()
        self.__register_queues()

    1 usage
    def __register_queues(self):
        self.__server_manager.register(QueueNames.RESULTS_QUEUE_NAME, callable=lambda: self.__results)
        self.__server_manager.register(QueueNames.TASKS_QUEUE_NAME, callable=lambda: self.__tasks)

    1 usage
    def __create_queues(self):
        self.__results: Queue[SingleResult] = Queue()
        self.__tasks: Queue[SingleTask] = Queue()

    1 usage
    def start_server(self):
        self.__server_manager.get_server().serve_forever()
```

2.2 – Sposób implementacji klienta

Następnym krokiem w implementacji problemu była implementacja klienta, którego zadaniem jest przeczytanie wektora oraz macierzy z pliku, a następnie podzielenie danych po równo i wysłanie na kolejkę do odpowiednich workerków, żeby obliczyły poszczególne

części wyniku. Sposób, w jaki podzieliliśmy dane był zależny od argumentu wejściowego do klienta(numer_of_tasks). Parametr definiował na ile zadań chcemy podzielić zadania, które trafią potem do workerów. Przykładowy kod z podziału danych i wysłaniu zadań do workerów prezentuje się następująco:

```
1 usage
def __get_matrix_and_vector_data(self, matrix_file_path: str, vector_file_path: str):
    self.__matrix = read_matrix_of_file_path(matrix_file_path)
    self.__vector = read_vector_of_file_path(vector_file_path)

1 usage
def __partition_data_and_send_to_tasks_queue(self) -> None:
    step_per_each_task: int = int(get_matrix_number_of_rows(self.__matrix) / self.__number_of_tasks)
    tasks: Queue[SingleTask] = self.__manager.tasks_queue()
    print("Partitioning the data into the servers")

    for index in range(self.__number_of_tasks):
        starting_row: int = int(index * step_per_each_task)
        ending_row: int = int(get_matrix_number_of_rows(self.__matrix)) if index == self.__number_of_tasks - 1 else (index +
        submatrix: list[list[float]] = self.__matrix[starting_row:ending_row]
        tasks.put(SingleTask(starting_row, ending_row, submatrix, self.__vector))
```

Po wykonanych zadaniach przez workerów, zadaniem klienta było również zebranie wszystkich wyników do końcowego wyniku i wypisanie wyniku na stdout. Wykorzystaliśmy do tego wcześniej stworzoną kolejkę result_queue:

```
1 usage
def gather_results_from_workers(self) -> list[float]:
    print("Gathering results")
    results_queue_from_workers: Queue[SingleResult] = self.__manager.results_queue()
    result = [.0] * get_matrix_number_of_rows(self.__matrix)

    while not results_queue_from_workers.empty():
        current_single_result: SingleResult = results_queue_from_workers.get()
        result[current_single_result.starting_row:current_single_result.ending_row] = current_single_result.single_result

    return result
```

2.3 – Sposób implementacji workera

Ostatnim krokiem w implementacji programu było napisanie klasy workera, który na podstawie otrzymanych zadań, miał obliczyć pojedynczy wynik oraz następnie wysłać wynik do wcześniej opisanej kolejki rezultatów. W implementacji workera na początku uruchamiamy tyle procesów, ile mamy cpu na naszym komputerze. Wszystkie procesy oczekują na zadania z kolejki i je wykonują, gdy jakieś się pojawi:

```python
def __init__(self, ip_address: str = "localhost", port: int = 8080, authorization_key: bytes = b"root"
    self.__manager = get_base_queue_manager(ip_address, port, authorization_key)
    self.__manager.connect()
    self.__register_queues()
    self.__initialize_processes()

1 usage
def start_working(self) -> None:
    for process_number in range(len(self.__processes)):
        self.__processes[process_number].start()

1 usage
def wait_until_processes_stop_working(self) -> None:
    print("Waiting for processes to end")
    for process_number in range(len(self.__processes)):
        self.__processes[process_number].join(timeout=5)

    for process_number in range(len(self.__processes)):
        if self.__processes[process_number].is_alive():
            self.__processes[process_number].terminate()

1 usage
def __register_queues(self) -> None:
    self.__tasks_queue: Queue[SingleTask] = self.__manager.tasks_queue()
    self.__results_queue: Queue[SingleResult] = self.__manager.results_queue()

1 usage
def __initialize_processes(self) -> None:
    self.__processes: list[Process] = []

    for process_index in range(cpu_count()):
        current_process: Process = Process(target=execute_single_work, args=(self.__tasks_queue
                                                                              , self.__results_queue))
        self.__processes.append(current_process)
```

Każdy z procesów wykonuje pojedynczy element pracy:

```python
2 usages
def execute_single_work(tasks_queue: Queue, results_queue: Queue) -> None:
    while not tasks_queue.empty():
        current_task: SingleTask = tasks_queue.get()
        current_submatrix_result: list[float] = []

        for row in range(len(current_task.matrix_rows)):
            current_row = current_task.matrix_rows[row]
            current_value: float = 0.0

            for column_index in range(len(current_task.vector)):
                current_value += current_row[column_index] * current_task.vector[column_index]

            current_submatrix_result.append(current_value)

        print(f"Worker has calculated the single_task of len: [{len(current_submatrix_result)}]")
        results_queue.put(SingleResult(current_task.starting_row, current_task.ending_row, current_submatrix_result))
```

Do klienta zostają zwracane pojedyncze wyniki: SingleResult, które pokazują startowy wiersz, końcowy wiersz oraz podwektor, który został obliczony w celu sklejenia ostatecznego wyniku. Do wykonywania zadań mamy również klasę SingleTask, która posiada informację o podmacierzy do obliczenia oraz wektorze przez który mnożymy i oczywiście startowym wierszu i końcowym wierszu.

2.4 – Łącznie wszystkich kroków

Ostatnim etapem było połączenie wszystkich klocków oraz uruchomienie programu. Na początku musimy uruchomić klasę serwer, a następnie klasę client.py oraz przykładowych workerów, którzy będą obliczać zadania.

3. Uruchomienie programu oraz wyniki

python server.py localhost 8080

python main.py data/A.dat data/X.dat localhost 8080 4

Pisząc wynik dla x procesów oznacza, że dając macierz o rozmiarze 2000x2000, program podzieli macierz wejściową na x zadań po 2000 / x elementów do pomnożenia dla każdego workera, który pracuje.

Rozmiar macierz	Rozmiar wektora	Liczba komputerów	Wynik dla procesów[s]	Wynik dla 4 procesów [s]	Wynik dla 8 procesów [s]	Wynik dla 16 procesów [s]	Wynik dla 32 procesów [s]
2000x1000	1000x1	1	11.98	16.88	16.23	16.85	51.06
2x3	3x1	1	0.66	0.69	0.65	0.66	0.69
4000x2000	2000x1	1	65.91	49.54	69.53	44.43	66.32
2000x1000	1000x1	2	26.05	24.53	24.11	19.53	20.02
2x3	3x1	2	5.63	6.63	5.26	5.93	6.32
4000x2000	2000x1	2	74.23	63.23	64.23	57.23	60.63

Porównanie wyników dla pojedynczego komputera:

Dla macierzy o rozmiarze 2000x1000 i wektora 1000x1, wyniki pokazują wzrost czasu wykonania dla większej liczby procesów. Dla 2 procesów czas wynosi 11.98 s, a dla 16 procesów wzrasta do 51.06 s. To zaskakujące, ponieważ zwykle zwiększenie liczby procesów powinno przyspieszyć obliczenia. Jednakże, zjawisko to może wynikać z dodatkowego narzutu związanego z zarządzaniem wątkami lub konkurencją o zasoby.

Analiza wyników dla mniejszych danych (2x3 i 3x1):

Dla mniejszych rozmiarów danych (2x3 i 3x1), czas wykonania jest znacznie mniejszy (0.66 s dla 2 wątków), co jest zrozumiałe ze względu na mniejszą ilość danych do przetworzenia.

Porównanie wyników dla dwóch komputerów:

Przy zwiększonej liczbie komputerów (2), czas wykonania dla większych macierzy (4000x2000 i 2000x1) jest zazwyczaj krótszy w porównaniu do pojedynczego komputera. Na przykład, dla macierzy 4000x2000 i 2000x1, czas wykonania przy 2 wątkach zmniejsza się znacząco.

Wnioski ogólne:

W przypadku większych zestawów danych (np. 4000x2000), zwiększenie liczby wątków lub komputerów może prowadzić do poprawy czasu wykonania, ale nie zawsze. Może istnieć punkt, po którym dodatkowe zasoby (procesy, komputery) nie przynoszą już korzyści i mogą

wręcz prowadzić do wydłużenia czasu wykonania z powodu dodatkowego narzutu na zarządzanie zasobami.

Dla mniejszych zestawów danych (2x3), różnice czasowe pomiędzy różnymi konfiguracjami są minimalne ze względu na niewielką ilość danych do przetworzenia.

Wniosek: Optymalna konfiguracja (liczba procesów i komputerów) zależy od wielkości danych i charakterystyki problemu. Należy dokładnie analizować wyniki, aby zoptymalizować wydajność obliczeń macierzowo-wektorowych.