WOJSKOWA AKADEMIA TECHNICZNA

Obliczenia Równoległe i Rozproszone

Sprawozdanie z zadania laboratoryjnego nr 1

Prowadzący: mgr inż. Piotr Stąpor

Zadanie wykonali: Igor Sokół i Daniel Filipek

Grupa: WCY18IJ6S1

Zadanie: Mnożenie macierzy 2x2

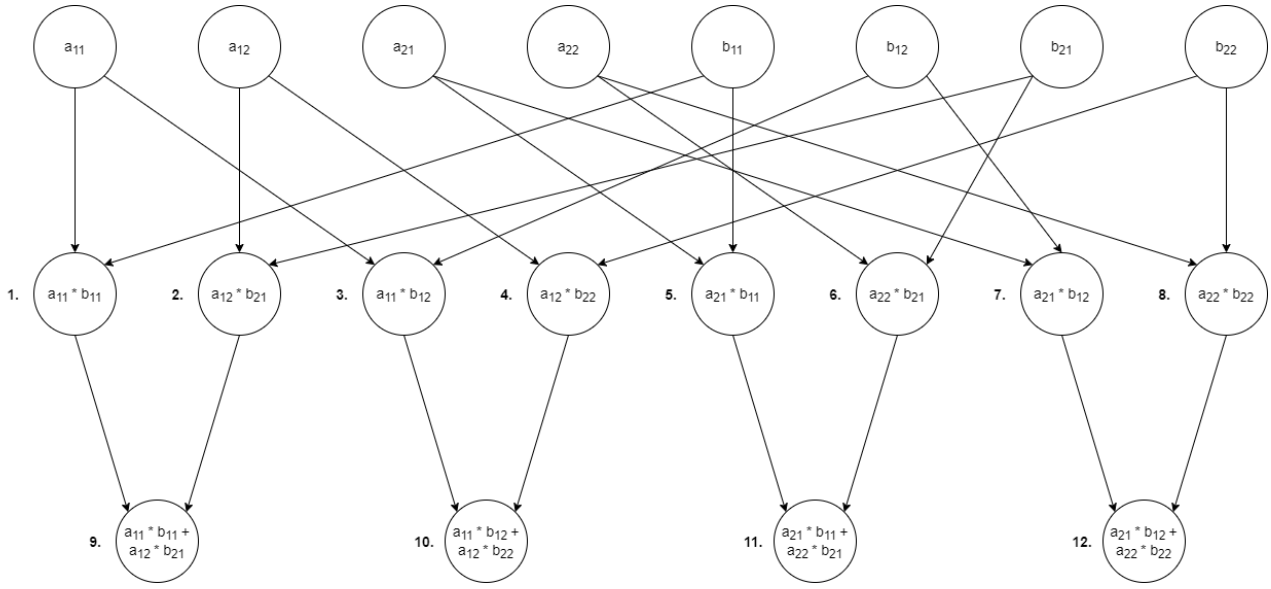
Treść zadnia:
Zaimplementować program mnożący dwie macierze 2x2 każda. Program ma wykorzystywać pakiet RMI (Java) w celu zrównoleglenia obliczeń. Należy również zadbać o synchronizację wykonywanych obliczeń.

1. Graf AGS

Zadanie obliczeniowe, które należy wykonać:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

Poniżej przedstawiono graf AGS obrazujący kolejność wykonywania działań.



2. Analiza kodu

Agent.class

```java
public class Agent {
    public static void main(String[] args) {
        if(args.length == 0) {
            System.err.println("Nie podano id agenta. Zamykam!");
            return;
        }

        int agentId;

        try {
            agentId = Integer.parseInt(args[0]);
        } catch (final NumberFormatException exception) {
            System.err.println("Podane id nie jest liczbą. Zamykam!");
            return;
        }

        if(agentId < 1 || agentId > Configuration.MAX_AGENT_ID) {
            System.err.println("Maksymalne id agenta to " + Configuration.MAX_AGENT_ID + ". Zamykam!");
            return;
        }

        try {
            // Instantiating the implementation class
            final CalculationServiceImpl service = new CalculationServiceImpl();

            // Exporting the object of implementation class
            // (here we are exporting the remote object to the stub)
            final CalculationService serviceStub = (CalculationService) UnicastRemoteObject.exportObject(service, Configuration.REMOTE_OBJECT_PORT);

            // Binding the remote object (stub) in the registry
            final Registry registry = LocateRegistry.getRegistry(Configuration.RMI_REGISTRY_PORT);

            registry.bind( name: "Agent" + agentId, serviceStub);
            System.out.println("Agent(id=" + agentId + ") gotowy!");
        } catch (final AlreadyBoundException exception) {
            System.err.println("Agent(id=" + agentId + ") jest już zbindowany!");
        } catch (final Exception exception) {
            System.err.println("Agent(id=" + agentId + ") exception: " + exception.toString());
            exception.printStackTrace();
        }
    }
```

Agent posiada metodę *main* która jako parametr przyjmuje ID agenta, czyli liczbę między 1 a 100. W pierwszej części metody następuje walidacja danych. Następnie w bloku *try/catch* agent eksportuje do rejestru powołany obiekt zawierający metody używane do obliczeń. Nazwa „stuba” to połączenie nazwy „Agent” i ID agenta.

CalculationServiceImpl.class

```java
public class CalculationServiceImpl implements CalculationService{

    @Override
    synchronized public int calculate(final OperationDto operation) throws RemoteException {
        System.out.println("Started Working...");
        try {
            Thread.sleep(Utils.getRandomLong(1000, 1000));
        } catch (final InterruptedException exception) {
            exception.printStackTrace();
        }

        int result;
        switch (operation.getOperator()) {
            case ADD -> {
                result = add(operation.getArg1(), operation.getArg2());
            }
            case MULTIPLY -> {
                result = multiply(operation.getArg1(), operation.getArg2());
            }
            default -> {
                System.out.println("Wykryto nie wspierana operacje, zwracam 0!");
                result = 0;
            }
        }

        System.out.println("Finished Working...");
        return result;
    }

    private int add(final int arg1,final int arg2) {
        return arg1 + arg2;
    }

    private int multiply(final int arg1, final int arg2) {
        return arg1 * arg2;
    }
}
```

Implementacja interfejsu *CalculationService* implementuje metodę *calculate*, która symuluje czas wykonania obliczenia (*Thread.sleep()*) i wykonuje obliczenie. Dostęp do metody jest synchronizowany za pomocą słowa kluczowego *synchronized*.

Server.class

```java
public class Server {

    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry(Configuration.RMI_REGISTRY_PORT);

            MatrixMultiplicationServiceImpl obj = new MatrixMultiplicationServiceImpl();
            MatrixMultiplicationService service = (MatrixMultiplicationService) UnicastRemoteObject.exportObject(obj,  port: 0);

            registry.bind(Configuration.SERVER_SERVICE_IMPL_NAME, service);
            System.out.println("Server ready");
        } catch (Exception e) {
            System.out.println("Server exception: " + e);
            e.printStackTrace();
        }
    }
}
```

W bloku *try/catch* agent eksportuje do rejestru powołany obiekt, który zarządzać będzie całym procesem mnożenia macierzy z wykorzystaniem agentów.

MatrixMultiplicationServiceImpl.class

```java
public class MatrixMultiplicationServiceImpl implements MatrixMultiplicationService {

    List<CalculationService> agents = new ArrayList<>();
    private final int agentsInUse;

    public MatrixMultiplicationServiceImpl() {
        try {
            Registry registry = LocateRegistry.getRegistry(Configuration.RMI_REGISTRY_PORT);
            Arrays.stream(registry.list())
                    .filter(stub -> stub.contains("Agent"))
                    .forEach(name -> {
                        try {
                            agents.add((CalculationService) registry.lookup(name));
                        } catch (RemoteException | NotBoundException e) {
                            e.printStackTrace();
                        }
                    });
            System.out.println("Wykryto " + agents.size() + " Agentów");
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        agentsInUse = agents.size();
    }
```

W konstruktorze klasy pobieramy rejestr i zbieramy z niego nazwy powiązane w rejestrze, z których filtrujemy tylko agentów i za pomocą metody *lookup* pobieramy referencję i umieszczamy ją w liście.

```java
@Override
public int[][] multiplyMatrixes(int[][] firstMatrix, int[][] secondMatrix) {
    OperationDto[] multiplications = new OperationDto[]{
            new OperationDto(firstMatrix[0][0], secondMatrix[0][0], Operator.MULTIPLY),
            new OperationDto(firstMatrix[0][1], secondMatrix[1][0], Operator.MULTIPLY),
            new OperationDto(firstMatrix[0][0], secondMatrix[0][1], Operator.MULTIPLY),
            new OperationDto(firstMatrix[0][1], secondMatrix[1][1], Operator.MULTIPLY),
            new OperationDto(firstMatrix[1][0], secondMatrix[0][0], Operator.MULTIPLY),
            new OperationDto(firstMatrix[1][1], secondMatrix[1][0], Operator.MULTIPLY),
            new OperationDto(firstMatrix[1][0], secondMatrix[0][1], Operator.MULTIPLY),
            new OperationDto(firstMatrix[1][1], secondMatrix[1][1], Operator.MULTIPLY)
    };

    int[] multiplicationResults = executeOperations(multiplications);

    OperationDto[] additions = new OperationDto[]{
            new OperationDto(multiplicationResults[0], multiplicationResults[1], Operator.ADD),
            new OperationDto(multiplicationResults[2], multiplicationResults[3], Operator.ADD),
            new OperationDto(multiplicationResults[4], multiplicationResults[5], Operator.ADD),
            new OperationDto(multiplicationResults[6], multiplicationResults[7], Operator.ADD),
    };

    int[] additionsResults = executeOperations(additions);

    return new int[][]{
            {
                    additionsResults[0],
                    additionsResults[1]
            },
            {
                    additionsResults[2],
                    additionsResults[3]
            }
    };
}
```

Implementacja metody *multiplyMatrixes* przyjmuje na wejściu obie macierze. Na początku tworzona jest lista operacji mnożenia (patrz graf AGS – pierwszy rząd działań), która następnie przekazywana jest do metody wykonującej obliczenia z listy. Analogicznie obliczane jest dodawanie. Metoda zwraca obliczoną macierz.

```java
private int[] executeOperations(OperationDto[] operations) {
    Executor[] executors = new Executor[operations.length];
    int[] results = new int[operations.length];
    for (int i = 0; i < operations.length; i++) {
        executors[i] = new Executor(agents.get(i % agentsInUse), operations[i]);
        final int index = i;
        executors[i].setOnCalculationFinishedListener(result -> results[index] = result);
        executors[i].start();
    }

    for (int i = 0; i < operations.length; i++) {
        try {
            executors[i].join();
        } catch (InterruptedException exception) {
            exception.printStackTrace();
        }
    }
    return results;
}
```

Ta metoda powołuje wątki dla każdego obliczenia z listy przekazanej w argumencie. Następnie są one „zbierane” w pętli z wykorzystaniem metody *join*, która pozwala na zaczekanie na wykonujący się wątek.

Executor.class

```java
public class Executor extends Thread {
    private final CalculationService agent;
    private final OperationDto operation;
    private int result;

    private OnCalculationFinished onCalculationFinished;

    public Executor(CalculationService agent, OperationDto operation) {
        this.agent = agent;
        this.operation = operation;
        this.result = 0;
    }

    @Override
    public void run() {
        try {
            result = agent.calculate(operation);
            Optional.of(onCalculationFinished)
                    .ifPresent(action -> action.onCalculationFinished(result));
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @FunctionalInterface
    interface OnCalculationFinished {
        void onCalculationFinished(final int result);
    }

    public void setOnCalculationFinishedListener(final OnCalculationFinished callback) {
        this.onCalculationFinished = callback;
    }
}
```

Klasa *Executor* rozszerza klasę *Thread*. W metodzie *run* wykonywana jest operacja *calculate* przekazana w agencie. W tej klasie zdefiniowany został interfejs, który służy wykorzystywany jest jako obserwator służący do zapisywania wyniku po wykonaniu obliczenia.

Client.class

```java
public class Client {

    public static void main(String[] args) {
        final MatrixHelper helper = new MatrixHelper( matrixWidth: 2, matrixHeight: 2);
        helper.promptForMatrixInput();

        helper.showMatrixes();

        try {
            // Getting the registry
            final Registry registry = LocateRegistry.getRegistry(Configuration.RMI_REGISTRY_PORT);

            // Looking up the registry for the remote object
            final MatrixMultiplicationService service = (MatrixMultiplicationService) registry.lookup(Configuration.SERVER_SERVICE_IMPL_NAME);

            final long startTime = System.currentTimeMillis();
            // Calling the remote method using the obtained object
            int[][] outputMatrix = service.multiplyMatrixes(helper.getFirstMatrix(), helper.getSecondMatrix());
            final long stopTime = System.currentTimeMillis();

            System.out.println("Macierz wynikowa :");
            helper.printMatrix(outputMatrix);
            System.out.println("Czas (ms) wykonania zadania: " + (stopTime - startTime));

        } catch (final Exception exception) {
            exception.printStackTrace();
        }
    }
}
```

Z rejestru za pomocą metody *lookup* pobierany jest udostępniony przez serwer obiekt zawierający implementację, która wykonuje mnożenie macierzy. Tutaj też odbywa się mierzenie czasu wykonania zadania przez serwer i agentów.

3. Działanie programu

1. *Uruchomienie rejestru*

Z poziomu terminala uruchamiamy rejestr komendą:

```
PS C:\Users\asus1\Desktop\programmin\Java\RMI\out\production\Java-RMI> start rmiregistry 1114
PS C:\Users\asus1\Desktop\programmin\Java\RMI\out\production\Java-RMI>
```

Port 1114 wykorzystywany jest w naszym programie.

2. *Kompilacja programu*

Z poziomu IDE lub za pomocą konsoli (javac *.java) kompilujemy program.

3. *Uruchomienie agentów*

Po kompilacji uruchamiamy agentów podając jako argument numer identyfikatora z zakresu 1-100 (tylu też agentów można uruchomić). Polecenie uruchomienia agenta to: *java Agent 1*, gdzie 1 to ID.

```
PS C:\Users\asus1\Desktop\programmin\Java\RMI\out\production\Java-RMI> java Agent 1
Agent(id=1) gotowy!
```

```
PS C:\Users\asus1\Desktop\programmin\Java\RMI\out\production\Java-RMI> java Agent 2
Agent(id=2) gotowy!
```

4. *Uruchomienie serwera*

Po uruchomieniu agentów można przejść do uruchomienia serwera za pomocą polecenia *java Server*. Serwer na początku działania przekaże nam informację o liczbie wykrytych agentów.

```
PS C:\Users\asus1\Desktop\programmin\Java\RMI\out\production\Java-RMI> java Server
Wykryto 2 Agentów
Server ready
```

5. *Uruchomienie i użycie klienta*

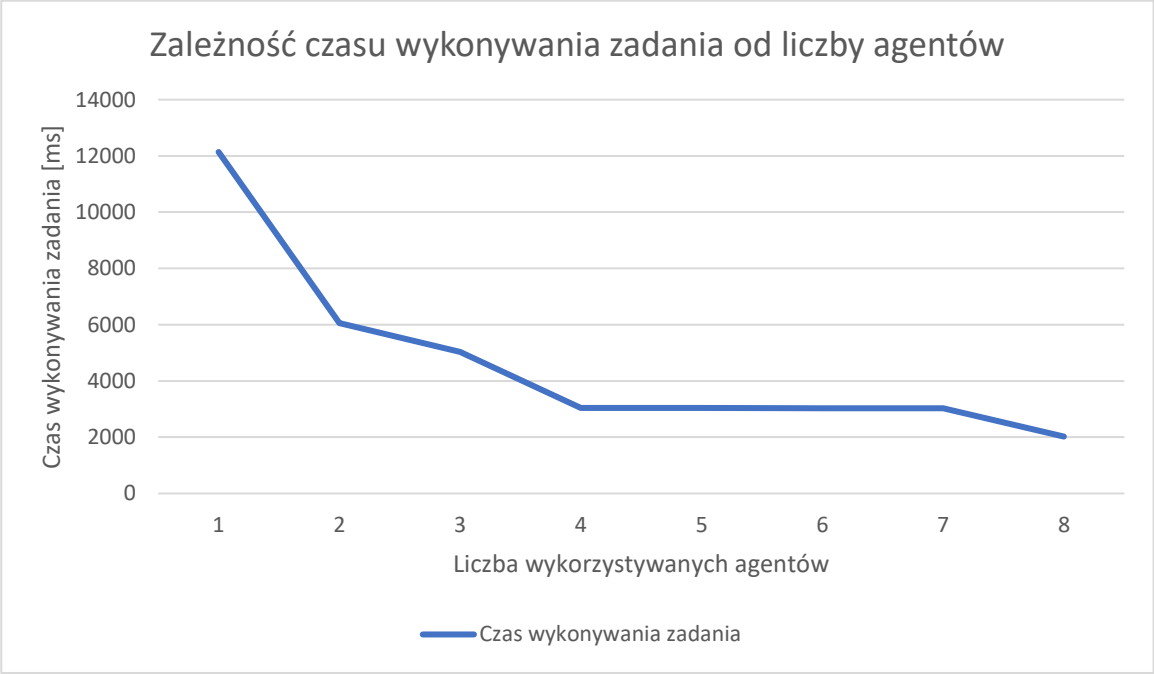
Po uruchomieniu serwera można uruchomić klienta i zacząć postępować wg instrukcji w terminalu. Po podaniu macierzy do przemnożenia program obliczy wynik i wyświetli go w terminalu razem z czasem wykonania obliczeń.

```
PS C:\Users\asus1\Desktop\programmin\Java\RMI\out\production\Java-RMI> java Client
Wprowadź pierszą macierz(ilość kolumn : 2, ilość wierszy : 2)
Przykładowa macierz :

8 8
9 5

Wprowadź poniżej :
1 1
1 1
Wprowadź drugą macierz(ilość kolumn : 2, ilość wierszy : 2)
Przykładowa macierz :

9 1
7 9

Wprowadź poniżej :
1 1
1 1
Wprowadzone macierze :

1 1
1 1

1 1
1 1

Macierz wynikowa :
2 2
2 2
Czas (w milisekundach): 6056
PS C:\Users\asus1\Desktop\programmin\Java\RMI\out\production\Java-RMI>
```

4. Analiza czasów

Poniższa tabela prezentuje czas wykonania zadania dla poszczególnych liczb wykorzystywanych agentów. Czas wykonania każdego działania zasymulowane jest na 1 sekundę (1000 milisekund).

Liczba agentów	Czas wykonania (w milisekundach)	Potwierdzenie
1	12139	`Czas (w milisekundach): 12139`
2	6056	`Czas (w milisekundach): 6056`
3	5032	`Czas (w milisekundach): 5032`
4	3045	`Czas (w milisekundach): 3045`
5	3039	`Czas (w milisekundach): 3039`
6	3031	`Czas (w milisekundach): 3031`
7	3032	`Czas (w milisekundach): 3032`
8	2024	`Czas (w milisekundach): 2024`



5. Wnioski

Patrząc na czasy wykonywania zadania, które zmieniają się w zależności od liczby wykorzystywanych agentów, można stwierdzić, że zadanie wykonane zostało poprawnie. Jeśli znaczące zmiany czasu by nie następowały oznaczałoby to, że obliczenia nie wykonują się równolegle (a np. współbieżnie).