# Zadanie laboratoryjne nr 6-7

Programowanie Współbieżne  
2014-12-20

Łukasz Ochmański 183566  
Marcel Wieczorek 173526

<https://code.google.com/p/programowanie-wspolbiezne/>

### Sprawozdanie

Celem zadania było zbadanie procesu mnożenia macierzy wykorzystując architekturę rozproszoną. Do tego celu można użyć jednego klienta i kilka węzłów (ang. nodes), są one także nazywane klastrami (ang. clusters). Stosowany przeze mnie algorytm mnoży macierz o rozmiarach NxN w czasie O(n3). Następnie wykorzystuje macierz wynikową do przemnożenia przez ostatnią macierz C o tym samym rozmiarze co również skutkuje przeprowadzeniem n3 operacji.

Za n przyjmujemy 1024 mnożenia i 1023 dodania. Zatem do n3 operacji w jednym procesie na jednym wątku program napisany w języku Java potrzebował jedynie 3 sekund. Jeśli natomiast do rozwiązania problemu użyliśmy interfejsu gniazd, dwa procesy, które wymieniały sie informacjami wymagały 16 sekund. Dodatkowy narzut był spowodowany użyciem dwóch procesów, które obciążały procesor. Wydajność spadała, a dodatkowo scheduler musiał się przełączać pomiędzy każdym z nich. Poza tym rozmiar bufora w potoku wynosił jedynie 65 kB, zatem procesor musiał wykonać dodatkowe czynności, aby przepychać dane w tzw. paczkach. W przypadku jednego procesu nie ma bufora, a proces ma bezpośredni i nieograniczony dostęp do pamięci operacyjnej. Te oraz wiele innych powodów, między innymi użycie nietypowych bibliotek np. ObjectOutpuStream, bardzo spowolniło uzyskanie wyniku. Szczegóły z przebiegu obu wersji programu zostały zamieszczone poniżej w sekcji: Profiling. Dla wersji implementującej interfejs gniazd warto nadmienić, że dane nie były przesyłane przez kartę sieciową tylko przez wewnętrzny loopback dla localhost pod adresem 127.0.0.1:4444.

Poniżej zamieszczam czasy wykonywania obliczeń dla różnych rozmiarów macierzy (N), dla jednego klienta i jednego węzła obliczeniowego. Na osi X zaznaczono rozmiar macierzy N. Na osi Y zaznaczono czas wykonania zadania w sekundach.

|  |  |
| --- | --- |
| **N (rozmiar macierzy)** | **Czas (sekundy)** |
| 64 | 1.07 |
| 128 | 1.65 |
| 256 | 2.31 |
| 512 | 4.74 |
| 1024 | 16.81 |

Aby skrócić czas obliczeń należało zmniejszyć rozmiar przesyłanych informacji. Najprostszym rozwiązaniem okazało się podzielenie macierzy A na wiersze i wysłanie do węzła tylko niezbędnej ilości informacji. Przykładowo jeśli mamy 4 węzły obliczeniowe wystarczy wysłać 256 wierszy do każdego węzła. Jeśli chodzi o macierz B, należało wysłać wszystkie wiersze i kolumny, aby dało się uzyskać sensowny wynik. Gdybyśmy wysłali mniej kolumn, wtedy nie udało by nam się uzyskać wyników mnożenia dla pełnej linii, i musielibyśmy zduplikować wysłanie danych do kolejnych węzłów. Byłoby to bardzo nieefektywne rozwiązanie, gdyż staramy się ograniczyć transfer do minimum. Poniżej zamieszczono dane wielkości przesyłanych żądań i odpowiedzi w bajtach:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar macierzy N | 1 węzeł | | 2 węzły | | 4 węzły | | 8 węzłów | |
| request | response | request | reponse | request | reponse | request | reponse |
| 1024 | 16818438  16.0 MB | 23068910  22.0 MB | 12613894  12.03 MB | 11534574  11.00 MB | 10511622  10.02 MB | 5767406  5.50 MB | 9460486  9.02 MB | 2883822  2.75 MB |

Po przemnożeniu wysłanych informacji w bajtach przez liczbę węzłów obliczeniowych otrzymujemy całkowitę liczbę przesłanych danych:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| nodes | request | sent to node | response | sent to client | total data |
| 1 | 16818438 | 16818438 | 23068910 | 23068910 | 39887348 |
| 2 | 12613894 | 25227788 | 11534574 | 23069148 | 48296936 |
| 4 | 10511622 | 42046488 | 5767406 | 23069624 | 65116112 |
| 8 | 9460486 | 75683888 | 2883822 | 23070576 | 98754464 |

Wszystkie operacje należy pomnożyć przez 2 ponieważ wysyłanie odbywa się drugi raz dla kolejnej macierzy.

Sprawność obliczyliśmy ze wzoru poniżej. Do obliczeń założyliśmy, że nasz program utworzy jednego klienta oraz k liczbę węzłów obliczeniowych. Jak łatwo zauważyć w sumie węzeł obliczeniowy musi wykonać ((1024\*(1024 mnożenia +1023 dodania))\*1024/k operacji na liczbach zmienno przecinkowych typu double. Dodatkowo w zależności od liczby węzłów to (1024\*1024)+(1024\*k) operacje wejścia/wyjścia.

Przykładowo dla k=1

ω(n) = 1024\*1024\*(1024+1023) = 2146435072 operacji obliczeniowych, ponieważ tyle jest operacji wewnątrz węzła obliczeniowego.

h(n,p) = (1024\*1024)\*2 = 2097152 operacji we/wy ponieważ tylko tyle liczb należy przesłać.

Stosunek liczby operacji obliczeniowych (2146435072) do liczby operacji we/wy (2097152) wynosi 0.000977

Zatem:

Niestety jednak sprawność powoli spada wraz z liczbą węzłów, gdyż wzrasta liczba przesyłanych danych. Zatem dla 1024 węzłów będzie już tylko 66%.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| węzły | liczba operacji dla jednego węzła | łączna liczba operacji | liczba I/O dla jednego węzła | I/O | sprawność | % |
| 1 | 2146435072 | 2146435072 | 2097152 | 2097152 | 0.999023914 | 99.90% |
| 2 | 1073217536 | 2146435072 | 1572864 | 3145728 | 0.998536585 | 99.85% |
| 4 | 536608768 | 2146435072 | 1310720 | 5242880 | 0.997563353 | 99.76% |
| 8 | 268304384 | 2146435072 | 1179648 | 9437184 | 0.995622568 | 99.56% |
| 16 | 134152192 | 2146435072 | 1114112 | 17825792 | 0.991763566 | 99.18% |
| 32 | 67076096 | 2146435072 | 1081344 | 34603008 | 0.984134615 | 98.41% |
| 64 | 33538048 | 2146435072 | 1064960 | 68157440 | 0.969223485 | 96.92% |
| 128 | 16769024 | 2146435072 | 1056768 | 135266304 | 0.940716912 | 94.07% |
| 256 | 8384512 | 2146435072 | 1052672 | 269484032 | 0.888454861 | 88.85% |
| 512 | 4192256 | 2146435072 | 1050624 | 537919488 | 0.799609375 | 79.96% |
| 1024 | 2096128 | 2146435072 | 1049600 | 1074790400 | 0.666341146 | 66.63% |

Powyższa zależność przedstawiona na wykresie:

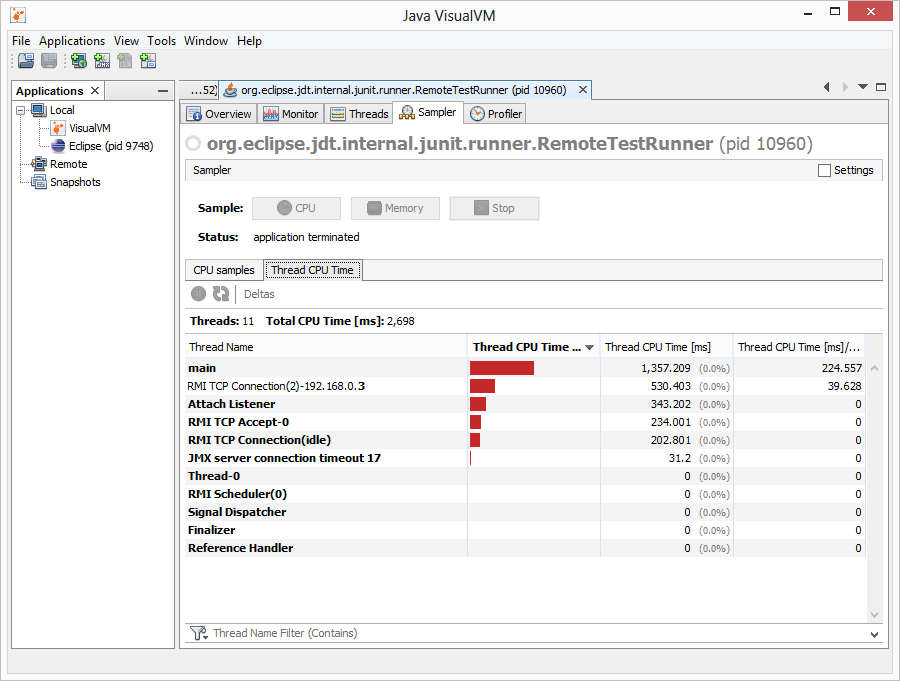
#### Wnioski

Sprawność programu mnożącego macierze jest niesamowicie duża, jednak w tym równaniu nie uwzględniono czasu jaki jest potrzebny do przesłania tak dużej liczby informacji. W tym wypadku jest to czas z jakim mierzą się rejestry procesora o częstotliwości 2 GHz z czasem przetwarzania w protokole TCP/IP. Jeśli założymy, że mnożenie wykonuje się w jednym cyklu zegara będzie to 2 miliardy operacji mnożenia/dodawania na sekundę vs. osiągnięte przez nas 35 MB/s (35 milionów bajtów procesor odczyta/przetworzy w ciągu sekundy) podczas odczytu z gniazda w naszym programie. Jest to 2000000000:35000000 = 1:57  
Oznacza to, że sprawność będzie około 57 razy mniejsza niż przewidziało równanie. Poza tym podane przez nas wartości są graniczne i nie mogą być w praktyce zrealizowane z wielu powodów. Są jeszcze opóźnienia w protokole TCP/IP związane z nawiązywaniem połączenia, przesyłaniem ramek, nagłówków IP. Dodatkowo dochodzi narzut systemu operacyjnego, który przydziela zasoby i zarządza potokami, wirtualna maszyna Javy oraz sposób skompilowania programu. Zaprezentowany przez nas model jest mocno abstrakcyjny i może być jedynie użyty do celów edukacyjnych.

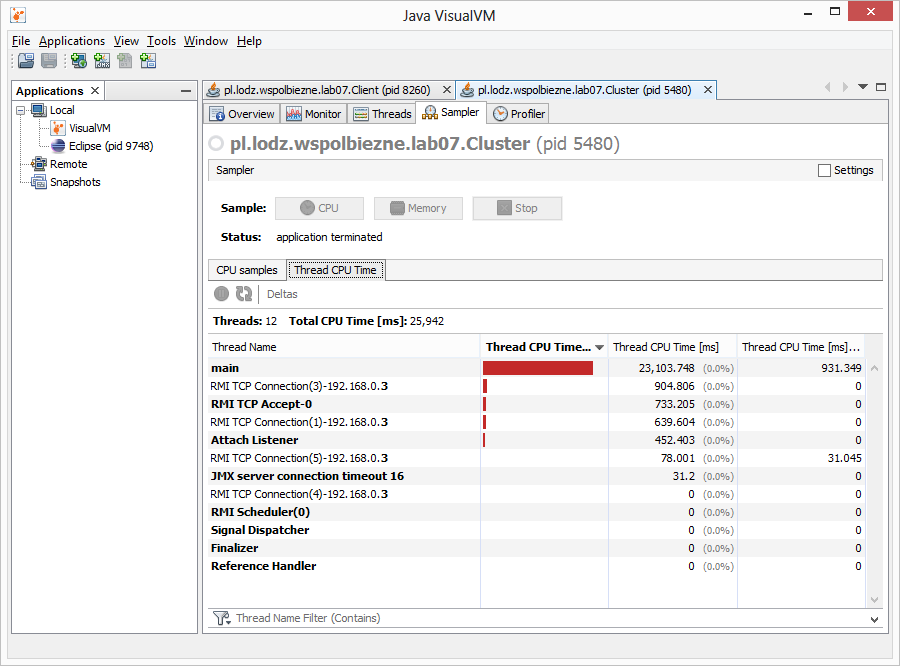
#### Profiling

Poniżej zamieszczam zrzuty ekranu z obserwacji dokonanych profilerem JVisualVM dla obu wersji programu: dla wersji jednoprocesowej i dla wersji wieloprocesowej z użyciem interfejsu gniazd.

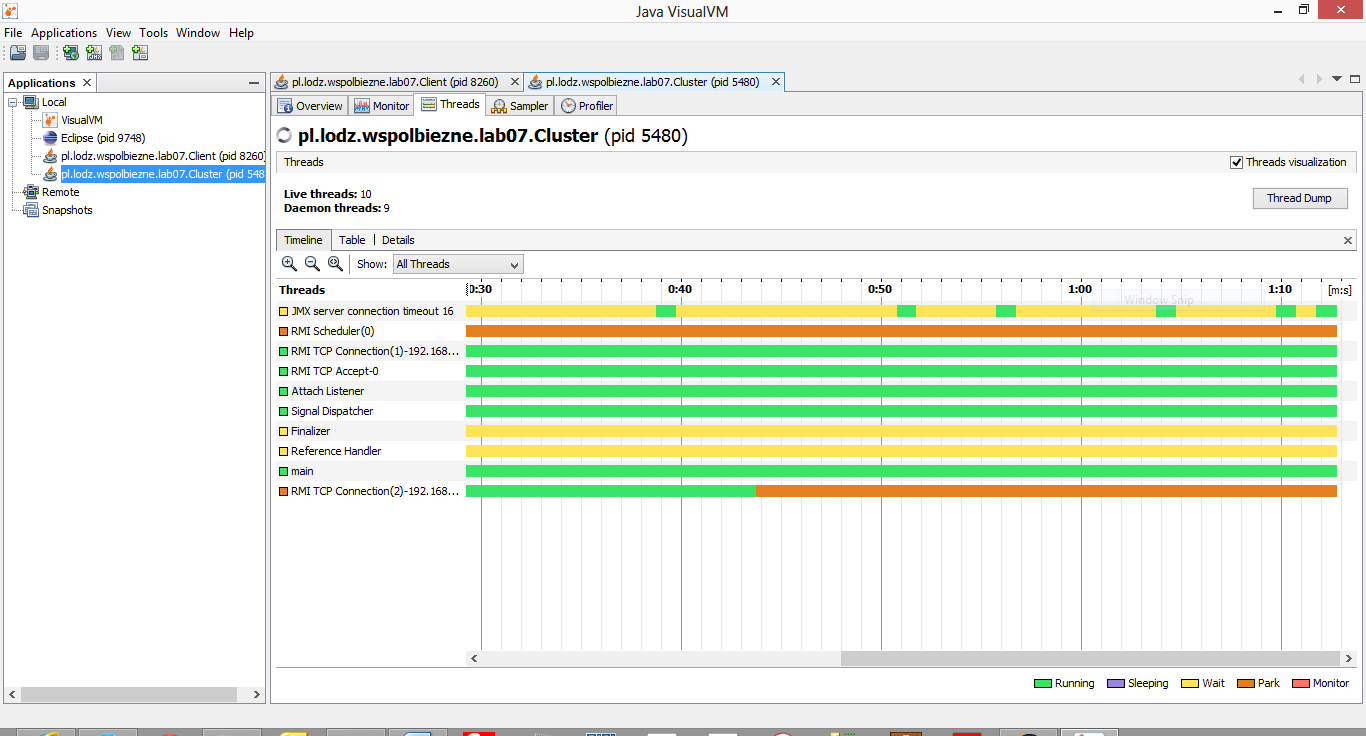
Przy jednoprocesowym mnożeniu macierzy, kompilator stworzył 11 wątków:



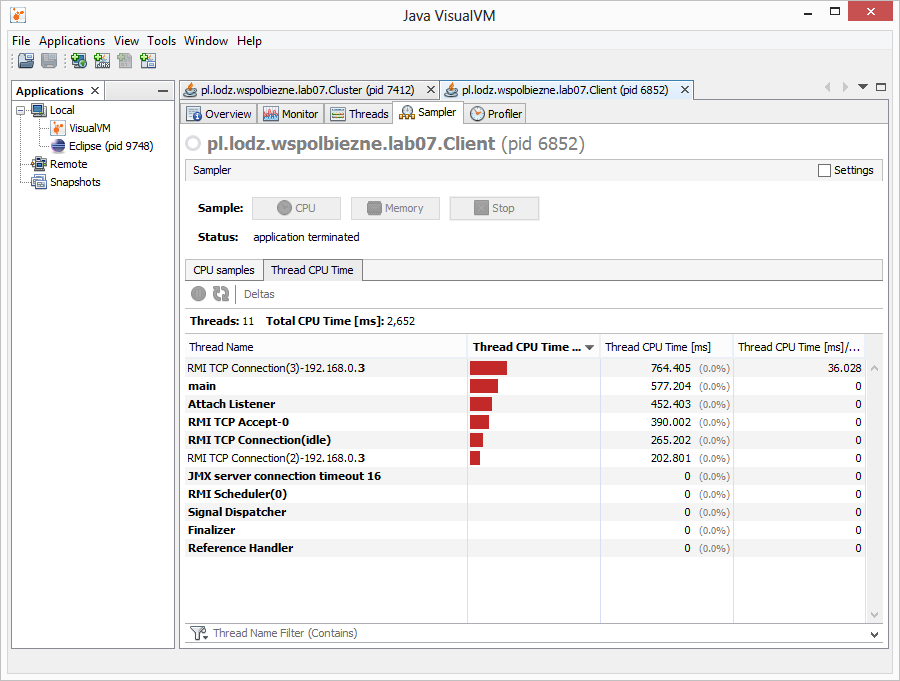
Przy użyciu interfejsu gniazd kompilator stworzył 12 wątków dla węzła obliczeniowego:



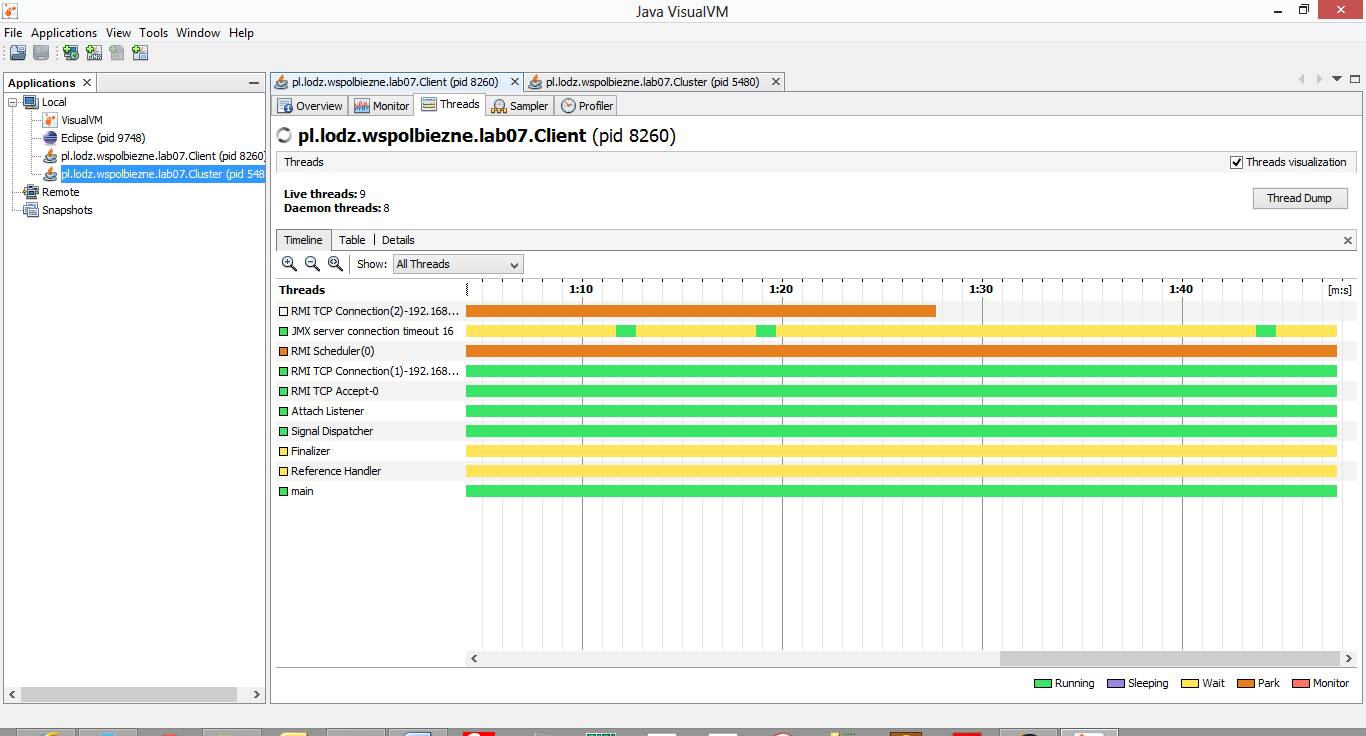
Threads:



I 11 wątków dla klienta:



Threads:



Kod programu:

**public** **class** Client {

**public** **static** **void** main(String[] args) {

**new** Client("localhost", 4444);

}

**private** **final** **int** LICZBA\_PROCESORÓW = 1;

**private** **final** **int** N = 1024;

**private** Logger logger;

**private** **long** start;

**private** ObjectInputStream ois;

**private** ObjectOutputStream oos;

// NIESTETY JAVA NIE JEST TAKA SPRYTNA I MUSZĘ POWTÓRZYĆ TO TRZY RAZY,

// ABY KOMPILATOR PRZYPISAŁ INNE ADRESY

**private** **double**[][] A = **new** **double**[N][N];

**private** **double**[][] B = **new** **double**[N][N];

**private** **double**[][] C = **new** **double**[N][N];

**public** Client(String hostName, **int** portNumber) {

start = System.*currentTimeMillis*();

logger = Obliczenia.*getCustomLogger*();

logger.info("Connecting to server at port: " + portNumber + " ...");

**for** (**int** i = 0; i < N; i++) {

A[i] = **new** Random().doubles(N).toArray();

B[i] = **new** Random().doubles(N).toArray();

C[i] = **new** Random().doubles(N).toArray();

}

**try** {

Socket kkSocket = **new** Socket(hostName, portNumber);

**int** receiveBufferSize = kkSocket.getReceiveBufferSize();

logger.info("Rozmiar bufora: ("

+ receiveBufferSize

+ " bytes) ("

+ Obliczenia.*humanReadableByteCount*(receiveBufferSize,

**false**) + ")");

dispatch(kkSocket);

logger.info("Zakończono obliczanie.");

System.***out***.println("Całkowity czas wykonania: "

+ (**double**) ((**double**)(System.*currentTimeMillis*() - start) / 1000.00)

+ " sekund.");

} **catch** (UnknownHostException e) {

logger.severe("Don't know about host " + hostName);

System.*exit*(1);

} **catch** (StreamCorruptedException e) {

logger.severe("This constructor will block until the corresponding"

+ " ObjectOutputStream has written and flushed the header.");

logger.severe(e.getMessage());

System.*exit*(1);

} **catch** (IOException e) {

logger.severe("Couldn't get I/O for the connection to " + hostName);

logger.severe(e.getMessage());

System.*exit*(1);

} **catch** (ClassNotFoundException e) {

logger.severe("Źle skastowany typ int[][][] / double[][][]");

System.*exit*(1);

}

}

@SuppressWarnings("unused")

**private** **void** dispatch(Socket kkSocket) **throws** IOException,

ClassNotFoundException {

InputStream inputStream = kkSocket.getInputStream();

BufferedInputStream bufferedIn = **new** BufferedInputStream(inputStream,

kkSocket.getReceiveBufferSize());

OutputStream outputStream = kkSocket.getOutputStream();

BufferedOutputStream bufferedOut = **new** BufferedOutputStream(

outputStream, kkSocket.getSendBufferSize());

logger.info("Trwa mnożenie macierzy AxB");

Obliczenia obliczenia = **new** Obliczenia(A, B);

**double**[][] AB = multiply(bufferedIn, bufferedOut, obliczenia);

logger.info("Trwa mnożenie macierzy ABxC");

obliczenia = **new** Obliczenia(AB, C);

**double**[][] ABC = multiply(bufferedIn, bufferedOut, obliczenia);

ois.close();

oos.close();

kkSocket.close();

**if** (N <= 8) {

System.***out***.println(Obliczenia.*toString*(ABC));

}

}

**private** **double**[][] multiply(BufferedInputStream bis,

BufferedOutputStream bos, Obliczenia obliczenia)

**throws** IOException, ClassNotFoundException {

**if** (oos == **null**) {

oos = **new** ObjectOutputStream(bos);

}

**for** (**int** proces = 0; proces < LICZBA\_PROCESORÓW; proces++) {

**int** start = getBeginningOfInterval(proces, LICZBA\_PROCESORÓW);

**int** end = getEndOfInterval(proces, LICZBA\_PROCESORÓW);

MacierzeDto C = obliczenia.getBlock(start, end);

Logger.*getGlobal*().info("Trwa wysyłka bloku nr " + proces);

**long** startTime = System.*currentTimeMillis*();

oos.writeObject(C);

oos.flush();

**int** sizeOfC = Obliczenia.*sizeOf*(C);

**long** duration = System.*currentTimeMillis*() - startTime;

logger.info("Zakończono przesyłanie bloku nr " + proces + " ("

+ Obliczenia.*humanReadableByteCount*(sizeOfC, **false**) + ")");

**long** speed = (**long**) (sizeOfC / (duration / 1000d));

logger.info("Write speed: " + sizeOfC + " bytes in " + duration

+ "ms");

logger.info("Write speed: "

+ Obliczenia.*humanReadableByteCount*(speed, **false**) + "/s");

}

**if** (ois == **null**) {

ois = **new** ObjectInputStream(bis);

}

ResultDto macierze;

**double**[][] AB = **new** **double**[N][N];

**int** i = LICZBA\_PROCESORÓW;

**int** size;

**while** (**true**) {

**if** (bis.available() != 0) {

logger.info("Stream available");

**long** startTime = System.*currentTimeMillis*();

**if** ((macierze = (ResultDto) ois.readUnshared()) != **null**) {

size = Obliczenia.*sizeOf*(macierze);

logger.info("Trwa odbieranie "

+ Obliczenia.*humanReadableByteCount*(size, **false**));

**long** duration = System.*currentTimeMillis*() - startTime;

**long** speed = (**long**) (size / (duration / 1000d));

logger.info("Read speed: " + size + " bytes in " + duration

+ "ms");

logger.info("Read speed: "

+ Obliczenia.*humanReadableByteCount*(speed, **false**)

+ "/s");

obliczenia.merge(macierze, AB);

**if** (--i == 0)

**break**;

}

}

}

**return** AB;

}

**public** **int** getBeginningOfInterval(**int** interval, **int** totalIntervals) {

**if** (totalIntervals <= interval) {

**throw** **new** IllegalArgumentException(

"Przedział nie może być większy niż: " + totalIntervals

+ " a podano: " + interval);

}

**double** fraction = (**double**) interval / (**double**) totalIntervals;

**return** (**int**) (fraction \* N);

}

**public** **int** getEndOfInterval(**int** interval, **int** totalIntervals) {

**if** (totalIntervals <= interval) {

**throw** **new** IllegalArgumentException(

"Przedział nie może być większy niż: " + totalIntervals

+ " a podano: " + interval);

}

**double** rozmiarPrzedzialu = (**double**) N / (**double**) totalIntervals;

**double** fraction = (**double**) interval / (**double**) totalIntervals;

**return** (**int**) ((fraction \* N) + rozmiarPrzedzialu);

}

}

**public** **class** Cluster {

Logger logger;

**public** **static** **void** main(String[] args) {

**new** Cluster(4444);

}

**public** Cluster(**int** portNumber) {

logger = Obliczenia.*getCustomLogger*();

logger.info("Server started at 127.0.0.1:" + portNumber);

**try** {

ServerSocket serverSocket = **new** ServerSocket(portNumber);

Socket clientSocket = serverSocket.accept();

processInput(clientSocket);

serverSocket.close();

logger.info("Connection closed");

} **catch** (IOException e) {

logger.severe("Exception caught when trying to listen on port "

+ portNumber + " or listening for a connection");

logger.severe(e.getMessage());

System.*exit*(1);

} **catch** (ClassNotFoundException e) {

logger.severe("Źle skastowany typ int[][][] / double[][][]");

System.*exit*(1);

}

}

**private** **void** processInput(Socket kkSocket) **throws** IOException,

ClassNotFoundException {

**int** receiveBufferSize = kkSocket.getReceiveBufferSize();

logger.info("Rozmiar bufora: (" + receiveBufferSize + " bytes) ("

+ Obliczenia.*humanReadableByteCount*(receiveBufferSize, **false**)

+ ")");

OutputStream outputStream = kkSocket.getOutputStream();

BufferedOutputStream bufferedOut = **new** BufferedOutputStream(

outputStream, kkSocket.getSendBufferSize());

ObjectOutputStream oos = **new** ObjectOutputStream(bufferedOut);

InputStream inputStream = kkSocket.getInputStream();

BufferedInputStream bufferedIn = **new** BufferedInputStream(inputStream,

kkSocket.getReceiveBufferSize());

ObjectInputStream ois = **new** ObjectInputStream(bufferedIn);

**if** (inputStream.markSupported()) {

logger.info("mark supported");

} **else** {

logger.severe("mark not supported");

}

MacierzeDto macierze;

**long** uptime = System.*currentTimeMillis*();

**while** (**true**) {

**long** startTime2 = System.*currentTimeMillis*();

**if** (inputStream.available() != 0) {

uptime = System.*currentTimeMillis*();

**if** ((macierze = (MacierzeDto) ois.readUnshared()) != **null**) {

**long** size2 = Obliczenia.*sizeOf*(macierze);

logger.info("Przyjęto macierz do obliczenia");

logger.info("Trwa odbieranie "

+ Obliczenia.*humanReadableByteCount*(size2, **false**));

**long** duration2 = System.*currentTimeMillis*() - startTime2;

**long** speed2 = (**long**) (size2 / (duration2 / 1000d));

logger.info("Read speed: " + size2 + " bytes in "

+ duration2 + "ms");

logger.info("Read speed: "

+ Obliczenia.*humanReadableByteCount*(speed2, **false**)

+ "/s");

**long** liczenieDuration = System.*currentTimeMillis*();

Obliczenia obliczenia = **new** Obliczenia();

ResultDto result = obliczenia.processInput(macierze);

logger.info("Czas mnożenia macierzy: "

+ (System.*currentTimeMillis*() - liczenieDuration)

+ "ms");

**long** startTime = System.*currentTimeMillis*();

**int** sizeOfResult = Obliczenia.*sizeOf*(result);

logger.info("Trwa wysyłanie obliczeń ("

+ sizeOfResult

+ " bytes)"

+ " ("

+ Obliczenia.*humanReadableByteCount*(sizeOfResult,

**false**) + ")");

oos.writeObject(result);

oos.flush();

**long** duration = System.*currentTimeMillis*() - startTime;

**long** speed = (**long**) ((**long**) sizeOfResult / (duration / 1000d));

logger.info("Write speed: " + sizeOfResult + " bytes in "

+ duration + "ms");

logger.info("Write speed: "

+ Obliczenia.*humanReadableByteCount*(speed, **false**)

+ "/s");

}

}

**int** timeout = 20\_000;

**if** (System.*currentTimeMillis*() - uptime > timeout) {

ois.close();

oos.close();

kkSocket.close();

logger.info("Nastąpił timeout: " + timeout / 1000 + "s");

**break**;

}

}

}

}

**public** **class** Obliczenia {

**private** Logger logger = Logger.*getGlobal*();

**private** **int** N;

**private** **double**[][] A, B;

**public** Obliczenia(**double**[][] A, **double**[][] B) {

**if** (!(A.length == A[0].length && A[0].length == B.length

&& B.length == B[0].length)) {

**throw** **new** RuntimeException("Rozmiar macierzy ma być taki sam.");

}

N = A.length;

**this**.A = A;

**this**.B = B;

}

**public** Obliczenia() {

}

**public** MacierzeDto getBlock(**int** start, **int** end) {

MacierzeDto D = **new** MacierzeDto();

**int** l = end - start;

List<Zbiór> rows = **new** ArrayList<>(N);

List<Zbiór> columns = **new** ArrayList<>(l);

**for** (**int** j = 0; j < l; j++) {

Zbiór z1 = **new** Zbiór();

z1.setIndex(start + j);

List<Double> values1 = **new** ArrayList<>(N);

**for** (**int** i = 0; i < N; i++) {

values1.add(B[start + j][i]);

}

z1.setValues(values1);

rows.add(z1);

}

**for** (**int** j = 0; j < N; j++) {

Zbiór z2 = **new** Zbiór();

z2.setIndex(j);

**double**[] values2 = **new** **double**[N];

**for** (**int** i = 0; i < N; i++) {

values2[i] = (A[i][j]);

}

z2.setValues(values2);

columns.add(z2);

}

D.setColumns(columns);

D.setRows(rows);

**return** D;

}

**public** **int** multiply(**int**[] row, **int**[] col) {

**int** sum = 0;

**for** (**int** i = 0; i < N; i++) {

sum += row[i] \* col[i];

}

**return** sum;

}

**public** **int**[][] processInput(**int**[][] rows, **int**[][] columns) {

**int**[][] C = **new** **int**[rows.length][columns.length];

**for** (**int** i = 0; i < rows.length; i++) {

**for** (**int** j = 0; j < columns.length; j++) {

C[i][j] = multiply(rows[i], columns[i]);

}

}

**return** C;

}

**public** ResultDto processInput(MacierzeDto macierze) {

ResultDto result = **new** ResultDto();

**int** liczbaKolumn = macierze.getColumns().length;

**int** liczbaWierszy = macierze.getRows().length;

List<Element> elements = **new** ArrayList<>(liczbaKolumn \* liczbaWierszy);

**int** x = 0;

**for** (**int** i = 0; i < liczbaKolumn; i++) {

**if** ((**double**) i % ((**double**) liczbaKolumn / 10.0) < 1.0) {

**if** (x != i \* 100 / liczbaKolumn) {

x = i \* 100 / liczbaKolumn;

logger.info(x + "%");

}

}

**for** (**int** j = 0; j < liczbaWierszy; j++) {

Element e = **new** Element();

e.setKolumna(macierze.getColumn(i).getIndex());

e.setWiersz(macierze.getRow(j).getIndex());

**double** v = 0;

**int** size = macierze.getColumn(i).getValues().length;

**for** (**int** m = 0; m < size; m++) {

v += macierze.getColumn(i).getValue(m)

\* macierze.getRow(j).getValue(m);

}

e.setWartość(v);

elements.add(e);

}

}

result.setElements(elements);

**return** result;

}

**public** **void** mergeInverted(ResultDto result, **double**[][] ab) {

**for** (Element e : result.getElements()) {

ab[e.getKolumna()][e.getWiersz()] = e.getWartość();

}

}

**public** **void** merge(ResultDto result, **double**[][] ab) {

**for** (Element e : result.getElements()) {

ab[e.getWiersz()][e.getKolumna()] = e.getWartość();

}

}

**public** **static** **int** sizeOf(Object obj) **throws** IOException {

ByteArrayOutputStream byteOutputStream = **new** ByteArrayOutputStream();

ObjectOutputStream objectOutputStream = **new** ObjectOutputStream(

byteOutputStream);

objectOutputStream.writeObject(obj);

objectOutputStream.flush();

objectOutputStream.close();

**return** byteOutputStream.toByteArray().length;

}

**public** **static** String humanReadableByteCount(**long** bytes, **boolean** si) {

**int** unit = si ? 1000 : 1024;

**if** (bytes < unit)

**return** bytes + " B";

**int** exp = (**int**) (Math.*log*(bytes) / Math.*log*(unit));

String pre = (si ? "kMGTPE" : "KMGTPE").charAt(exp - 1)

+ (si ? "" : "");

**return** String.*format*("%.1f %sB", bytes / Math.*pow*(unit, exp), pre);

}

}