Laboratorium Nr 4
"Java Concurrency Utilities"
Radosław Kopeć
27.10.2020

I. Zadanie 1

1. Treść zadania

Producenci i konsumenci z losowa ilością pobieranych i wstawianych porcji

- Bufor o rozmiarze 2M
- Jest m producentów i n konsumentów
- Producent wstawia do bufora losowa liczbę elementów (nie więcej niż M)
- Konsument pobiera losowa liczbę elementów (nie więcej niż M)
- Zaimplementować przy pomocy monitorów Javy oraz mechanizmów Java Concurrency Utilities
- Przeprowadzić porównanie wydajności vs. różne parametry, zrobić wykresy i je skomentować

2. Koncepcja rozwiązania

Rozwiązanie problemu producentów i konsumentów przy użyciu Java Concurrency Utilities w programie nazwałem "library", wszystkie zmienne z przedrostkiem "lib" lub "library" dotyczą tego właśnie sposobu. Rozwiązanie z użyciem standardowej javy nazwałem "standard" i posiada ono analogiczny sposób nazewnictwa jak rozwiązanie library.

Klasy Producer,Consumer będą używały abstrakcji buffera, odwoływać się będą do metod interfejsu IBuffer implementowanego przez konkretne klasy LibBuffer, oraz StandardBuffer. Obydwie te klasy różnić się będą w użyciu tylko implementacją Semafora. LibBuffer korzystać będzie z gotowego z pakietu Java Concurrency Utilities, natomiast StandardBuffer z własnej implementacji przy użyciu notify/wait, oraz bloku synchronized.

Generalna koncepcja rozwiązania problemu producenta i konsumenta dla obu przypadków jest następująca. Używamy trzech semaforów liczących.

- writeSemaphore - semafor do zapisu, obrazuje ile jest wolnych pól do zapisu w buforze
- readSemaphore - semafor do odczytu, obrazuje ile jest pól gotowych do odczytu w buforze
- indexSemapgore - semafor pilnujący dostępu do tablicy canWrite. Tablica canWrite jest tablicą wartości logicznych o rozmiarze bufora. Gdy pole x jest ustawione na true, oznacza to, że w buforze na polu x, można dokonać zapisu. Natomiast gdy pole x jest ustawione na false, oznacza to, że w buforze na polu o indeksie x znajduje się wartosć do odczytania.

Początkowo wartość semafora writeSemaphore ustawiamy na rozmiar bufora, co oznacza, że można dokonać zapisu na każde pole. Wątki producentów otrzymują indeks do którego mogą coś zapisać przy użyciu funkcji findFieldToWrite(). Gdy producent zapisze coś do bufora dekrementuje wartość swojego semafora (nie zwalniając go), oraz inkrementuje wartość semafora readSemaphore tym samym odblokowuje jeden z wątków konsumera. Konsument działa analogicznie, wartość semafora readSemaphore jest początkowo ustawiona na 0, żaden konsument nie może przeczytać (bo na początku nic nie ma w tablicy), gdy któryś z producentów odblokuje konsumenta zwiększając wartość readSemaphore ten uzyskuje dostęp do czytanego pola przez metodę findFieldToRead(), nie zwalnia swojego semafora, oraz inkrementuje wartość semafora writeSemaphore. Może się zdarzyć że będą same pola do czytania i wszyscy producenci będą czekać, wtedy zwiększenie semafora writeSemaphore odblokuje jednego pisarza który znajdzie odpowiednie pole do zapisu. Semafor indexSemapfore służy do dostępu do tablicy canWrite poprzez metody findFieldToWrite(), oraz findFieldToRead().

Porównanie i wizualizacja:

Porównanie obu sposobów zostanie dokonane poprzez zmierzenie czasu wykonania programu dla różnych wątków. Będziemy mogli zmieniać liczbę producentów i konsumentów dla każdego wywołania, dlatego wywołamy program dla każdego $(p,c)\,\varepsilon\, M\, x\, M$ gdzie p to liczba producentów i c to liczba konsumentów. Liczba produkowanych dóbr przez producentów zostanie z góry ustalona na 10. Liczba dóbr absorbowanych przez konsumentów zostanie wyliczona na podstawie wszystkich dóbr produkowanych dla danego (p,c) według rozkładu równomiernego jak poniżej.

p = 5
c = 3
*liczba_dóbr = 5*10 = 50*
rozkład_dóbr_dla_konsumentów = [17,17,16]

Na koniec wyniki zostaną przedstawione na wykresie 3D przy użyciu biblioteki jzy3d.

Dodatkowo:

Zrezygnowałem z losowego wybierania liczby operacji dla producentów z dwóch powodów:

1. Konsumenci nie kończyli swojej pracy gdy wylosowali łącznie więcej operacji niż producenci.
2. Wyłączenie tej opcji daje mi większa kontrolne nad parametrami i przez to dokładniejsze wyniki.

3. **Implementacja i wyniki:**

Producer:

```
class Producer extends Thread {
    private IBuffer _buf;
    private int operations;

    public Producer(IBuffer buffer, int operations){
        super();
        this._buf = buffer;
        this.operations = operations;
    }

    public void run() {

//        Random random = new Random();
//        int operations = random.nextInt(this.operations);

        for (int i = 0; i < operations; i++) {
            _buf.put(i);
        }
    }
}
```

Zakomentowana część stanowi implementację losowania operacji.

Consumer:

```
class Consumer extends Thread {
    private final IBuffer _buf;
    private final int operations;
    public Consumer(IBuffer buffer, int operations){
        super();
        this._buf = buffer;
        this.operations = operations;
    }

    public void run() {

//      Random random = new Random();
//      int operations = random.nextInt(M);

      for (int i = 0; i < operations; i++) {
          _buf.get();
      }
    }
}
```

Zakomentowana część stanowi implementację losowania operacji.

Interfejs IBuffer:

```
public interface IBuffer {

    void put(int value);
    int get();

}
```

CountingSemapfore wykorzystywany w StandardBuffer:

```
public class CountingSemaphore {
    private int resources = 0;

    public CountingSemaphore(int resources) {
        this.resources = resources;
    }

    public synchronized void release() {
        this.resources++;
        this.notify();
    }

    public synchronized void acquire() throws InterruptedException{
        while(this.resources <= 0) wait();
        this.resources--;
    }

}
```

LibBuffer:

```java
public class LibBuffer implements IBuffer {

  private final int[] buffer;
  private final boolean[] canWrite;
  private final int size;
  private final Semaphore readSemaphore;
  private final Semaphore writeSemaphore;
  private final Semaphore indexSemaphore;

  public LibBuffer(int size){
    this.size = size;
    buffer = new int[size];
    canWrite = new boolean[size];

    for (int i =0;i < size; i++){
      buffer[i]= 0;
      canWrite[i] = true;
    }
    writeSemaphore = new Semaphore(size);
    indexSemaphore = new Semaphore( permits: 1);
    readSemaphore = new Semaphore( permits: 0);
  }
```

StandardBuffer:

```java
public class StandardBuffer implements IBuffer {

    private final int[] buffer;
    private final boolean[] canWrite;
    private final int size;
    private final CountingSemaphore readSemaphore;
    private final CountingSemaphore writeSemaphore;
    private final CountingSemaphore indexSemaphore;

    public StandardBuffer(int size){
        readSemaphore = new CountingSemaphore( resources: 0);
        writeSemaphore = new CountingSemaphore(size);
        indexSemaphore = new CountingSemaphore( resources: 1);

        this.size = size;
        buffer = new int[size];
        canWrite = new boolean[size];

        for (int i =0;i < size; i++){
            buffer[i]= 0;
            canWrite[i] = true;
        }
    }
```

Jak widać powyżej klasy LibBuffer i StandardBuffer różnią się wyłącznie semaforami. Poniższe funkcje są identyczne dla obu klas.

```
private int findFieldToWrite(){
  for(int i=0;i<size;i++){
      if(canWrite[i]){
        canWrite[i] = false;
        return i;
      }
  }
  throw new IllegalStateException("at least one field should be free to write");

}

private int findFieldToRead(){
  for(int i=0;i<size;i++){
    if(!canWrite[i]){
      canWrite[i] = true;
      return i;
    }
  }
  throw new IllegalStateException("at least one filed should be free to read");
}
```

```
public void put(int value) {

  try {
    writeSemaphore.acquire();
    indexSemaphore.acquire();

  } catch (InterruptedException e) {
    e.printStackTrace();
  }


    int index = findFieldToWrite();

    indexSemaphore.release();

    buffer[index] = value;


  readSemaphore.release();
```

```
public int get(){

  try {
    readSemaphore.acquire();
    indexSemaphore.acquire();

  } catch (InterruptedException e) {
    e.printStackTrace();
  }

    int index = findFieldToRead();


    indexSemaphore.release();
    int result = buffer[index];
    writeSemaphore.release();

    return result;

}
```

Klasa testująca:

Wyniki w nanosekundach znajdują się w tablicy long[M][M]. Pierwsza zmienna odpowiada liczbie producentów, druga liczbie konsumentów. M - rozmiar bufora.

Funkcja obliczająca średni czas:

```
private static void calculateAverage(long[][] tab, int M, String info){
    long sum = Arrays.stream(tab).flatMapToLong(Arrays::stream).reduce( identity: 0, Long::sum);
    System.out.println("Average for" + info + ": " + (double) sum/(1000000*M) + "ms");
}
```

Funkcja mierząca czas:

```
private static long measureTime(ArrayList<Thread> threads){
    long startTime = System.nanoTime();

    for( Thread thread: threads){
        thread.start();
    }

    for( Thread thread: threads){
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    return System.nanoTime() - startTime;
}
```

Funkcja rysująca wykres:

```
    private static void create3dPlot(long[][] table,int M,String surfaceName){

// Define a function to plot
        Mapper mapper = (x, y) -> { return (double) table[(int) x][ (int) y]/1000000; };

// Define range and precision for the function to plot
        Range range = new Range(0, M-1);
        int steps = 50;

// Create a surface drawing that function
        Shape surface = Builder.buildOrthonormal(new OrthonormalGrid(range, steps), mapper);
        surface.setFaceDisplayed(true);
        surface.setColorMapper(new ColorMapper(new ColorMapRainbow(), surface.getBounds().getZmin(), surface.getBounds().getZmax(), new Color( r: 1,  g: 1,  b: 1,  a: .5f)));
        surface.setWireframeDisplayed(false);
        surface.setWireframeColor(Color.BLACK);

// Create a chart and add the surface
        Chart chart = new AWTChart(Quality.Advanced);
        chart.add(surface);
        chart.open(surfaceName,  width: 1000,  height: 1000);
    }
```

Funkcja main programu:
M - rozmiar bufora
produceNumber - liczba dóbr produkowanych przez producenta
long[][] … - tablice przechowujące wyniki.Pierwsza zmienna odpowiada liczbie producentów, druga liczbie konsumentów, pole tablicy to wartość w nanosekundach czasu wykonania.

```
public static void main(String[] args) {
    System.out.println("Compare java monitors vs concurrency utilities");
    final int M = 30;
    final int produceNumber = 10;
    long[][] libTimes = new long[M][M];
    long[][] standardTimes = new long[M][M];
```

Wypełniamy tabele wynikami:

```
for(int producers = 1; producers <= M ; producers++){
    for(int consumers = 1; consumers <= M; consumers++){

        ArrayList<Thread> libThreads = new ArrayList<~>();
        ArrayList<Thread> standardThreads = new ArrayList<~>();

        LibBuffer libBuffer = new LibBuffer(M);
        StandardBuffer standardBuffer = new StandardBuffer(M);

          Spread operations between consumers
        int[] countOfConsumersOperations = new int[consumers];
        for(int i =0;i<consumers;i++){
            countOfConsumersOperations[i] = 0;
        }
        int poolOfOperations = producers*produceNumber;

        while (poolOfOperations != 0){
            countOfConsumersOperations[poolOfOperations%consumers]++;
            poolOfOperations --;
        }
```

```
//              add threads into list
            for(int i=0;i<producers ;i++){
                libThreads.add(new Producer(libBuffer, produceNumber));
                standardThreads.add(new Producer(standardBuffer, produceNumber));
            }

            for(int i=0;i<consumers ;i++){
                libThreads.add(new Consumer(libBuffer, countOfConsumersOperations[i]));
                standardThreads.add(new Consumer(standardBuffer, countOfConsumersOperations[i]));
            }

//          calculate time

            libTimes[producers-1][consumers-1] = measureTime(libThreads);
            standardTimes[producers-1][consumers-1] = measureTime(standardThreads);
        }
    }
```

Wypisanie wyników:

```
        printResult(standardTimes,M);
        System.out.println();
        printResult(libTimes,M);

        create3dPlot(libTimes,M, surfaceName: "Library Plot");
        create3dPlot(standardTimes,M, surfaceName: "Standard Java  Plot");

        long[][] diffTable = new long[M][M];
        for(int i= 0;i<M;i++){
            for(int j=0;j<M;j++){
                diffTable[i][j] =  standardTimes[i][j] - libTimes[i][j];
            }
        }
        create3dPlot(diffTable,M, surfaceName: "Difference");

        calculateAverage(standardTimes,M, info: "Standard Java");
        calculateAverage(libTimes,M, info: "Library");
    }
```

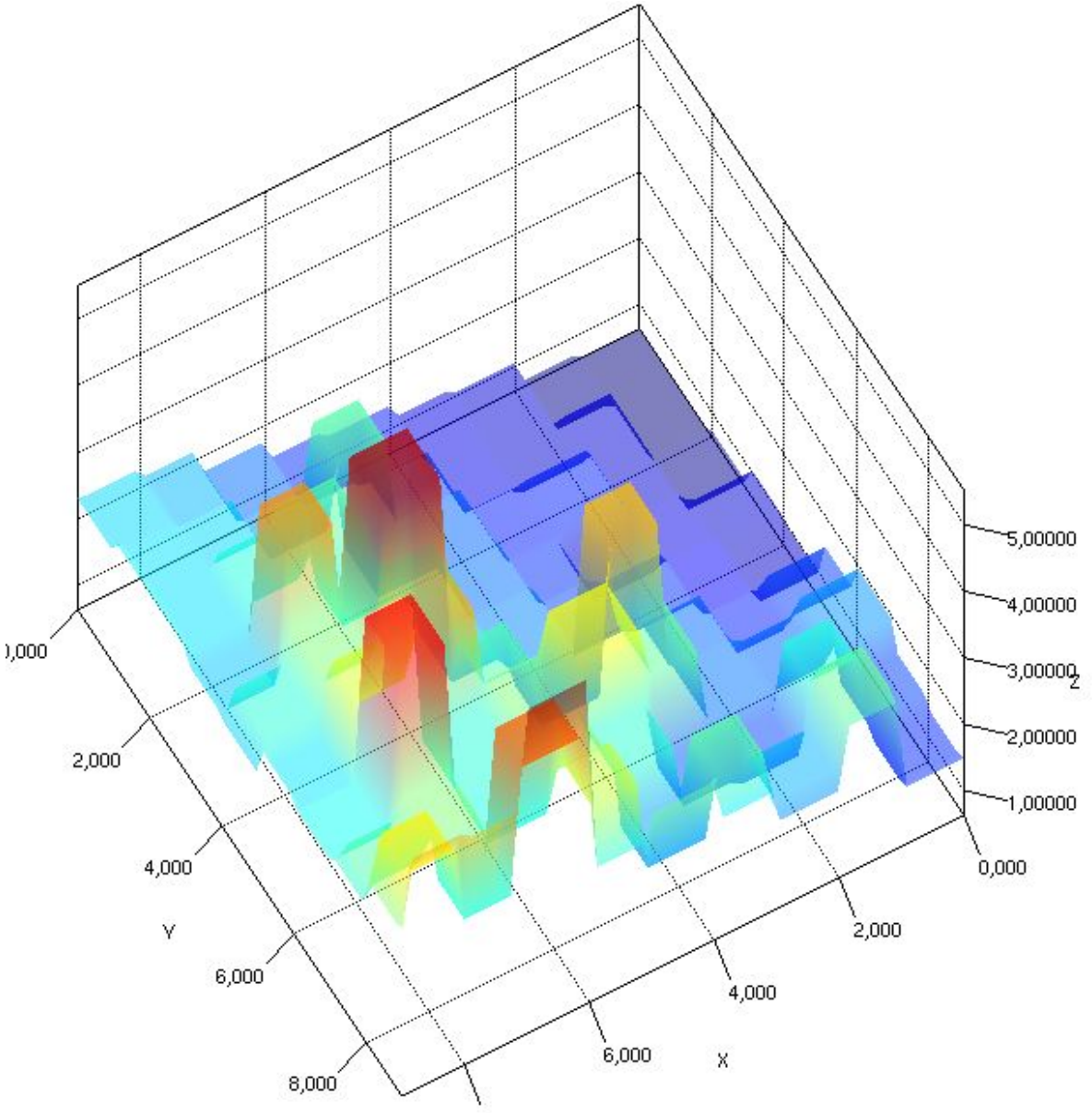
Funkcja pomocnicza do wypisywania tabeli:

```
private static void printResult(long[][] table, int M){
    for(int i=0;i<M;i++){
        System.out.println();
        for(int j =0;j<M;j++){
            System.out.print(table[i][j]);
            System.out.print(" ");
        }
    }
}
```

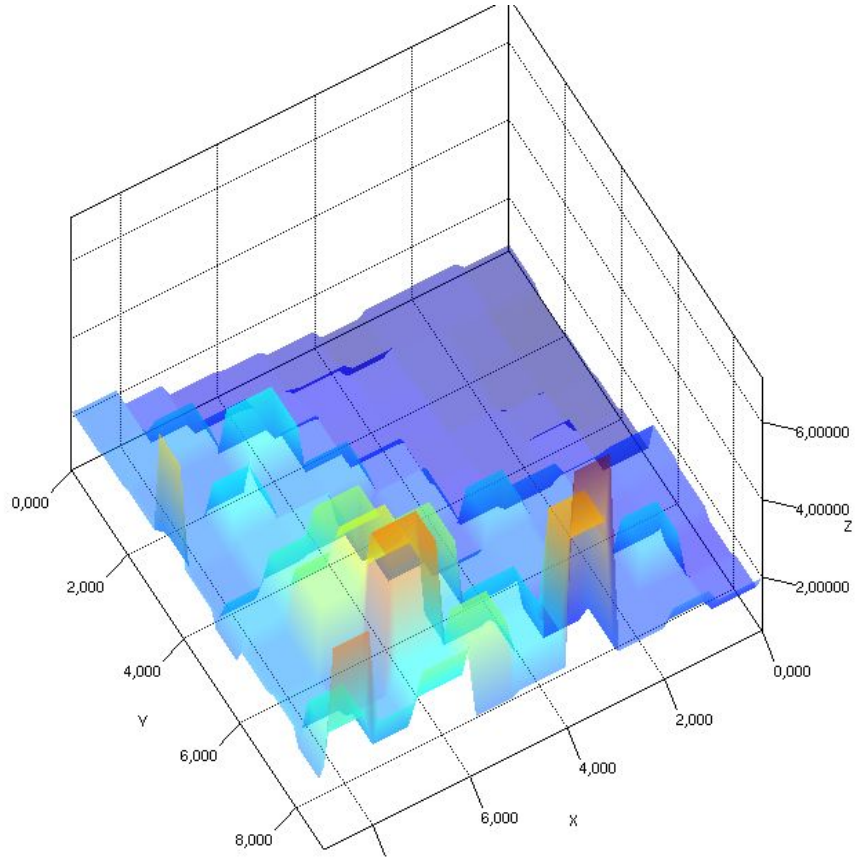
4. Wyniki

X - liczba producentów
Y -liczba konsumentów
Z - czas w ms
Dla M = 10

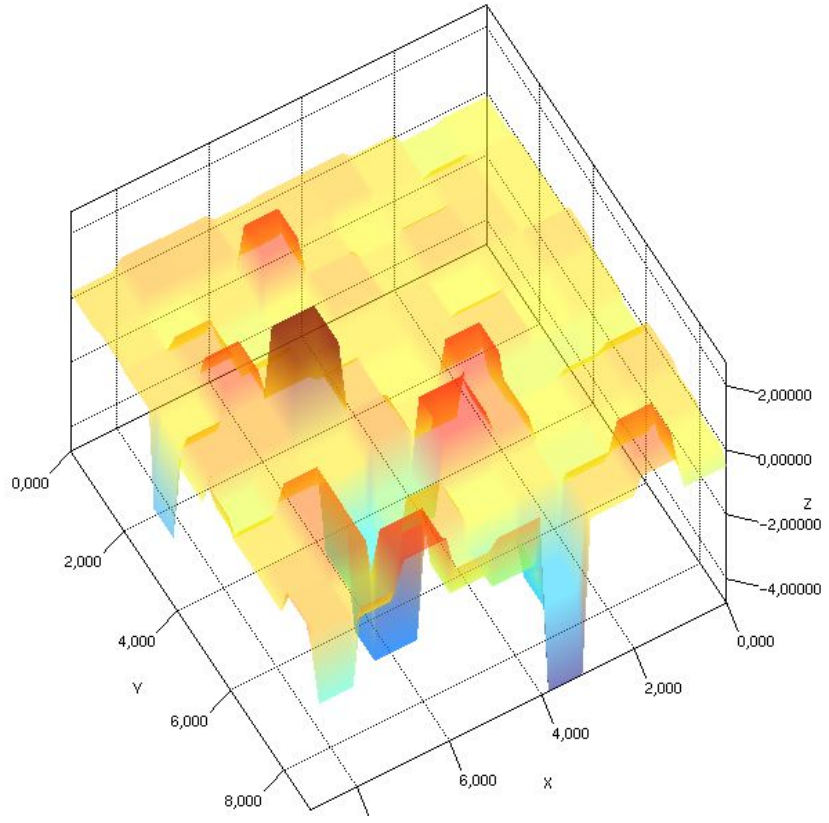
Standard:



Library:

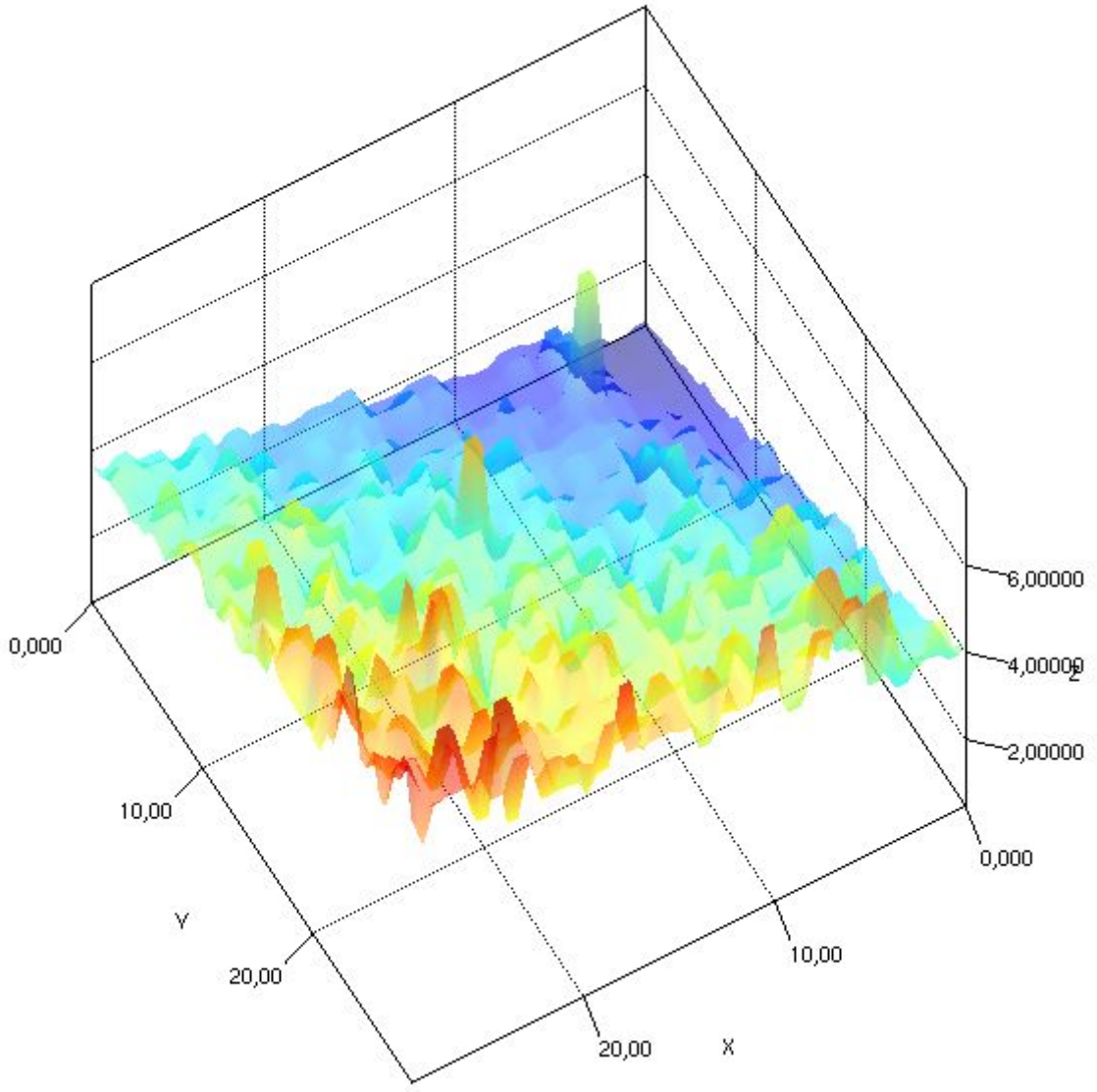


Różnica Standard - Lib:

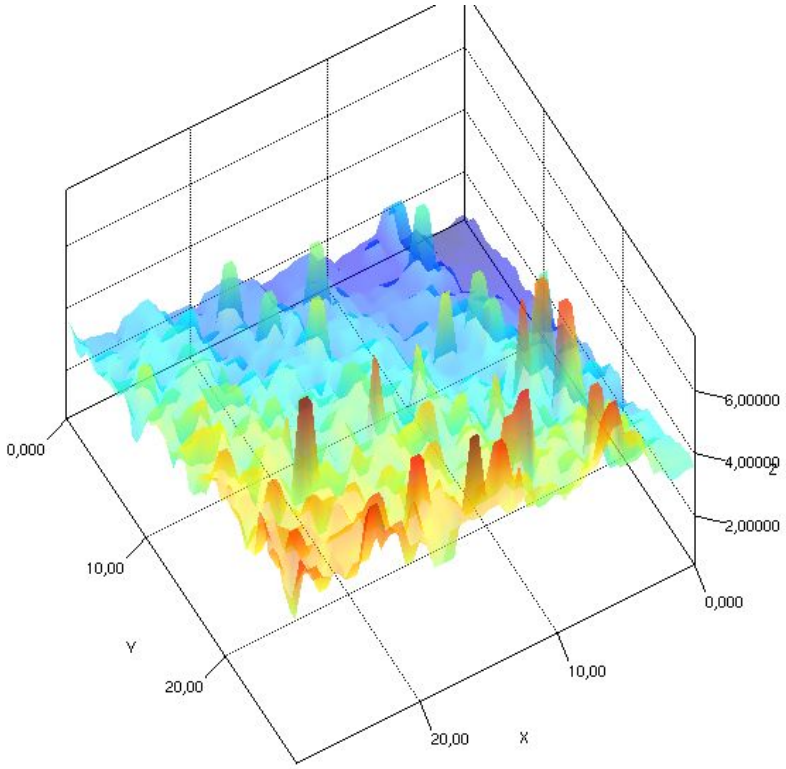


Average forStandard Java: 16.56617ms
Average forLibrary: 16.4074ms

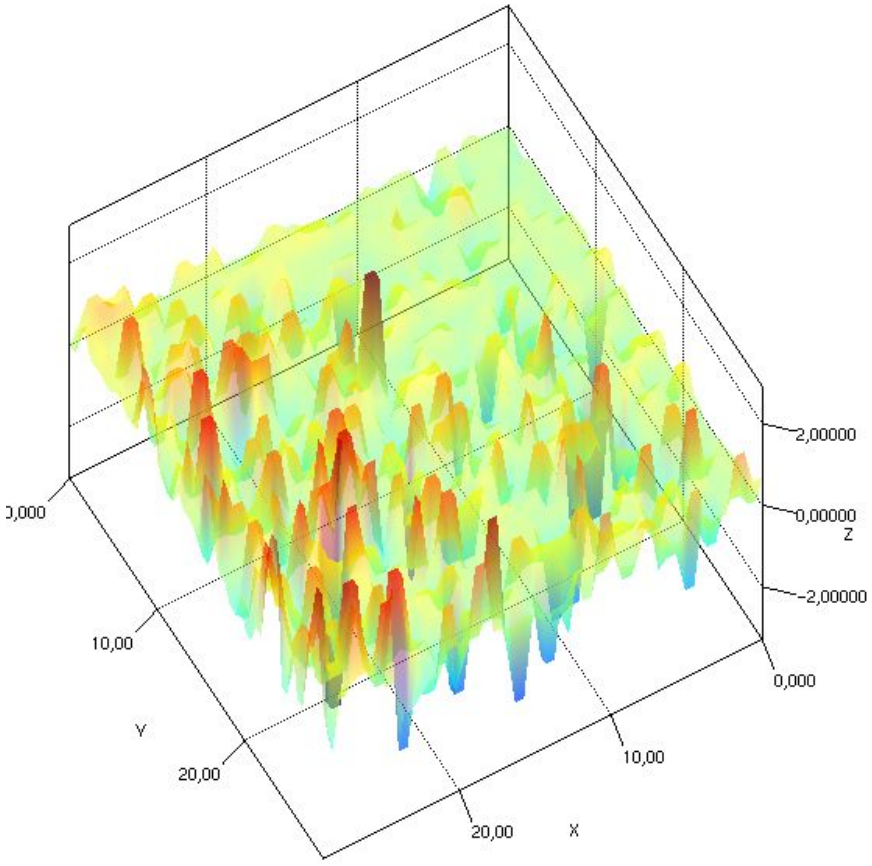
Dla M=30
Standard:



Library:



Różnica standard - library:

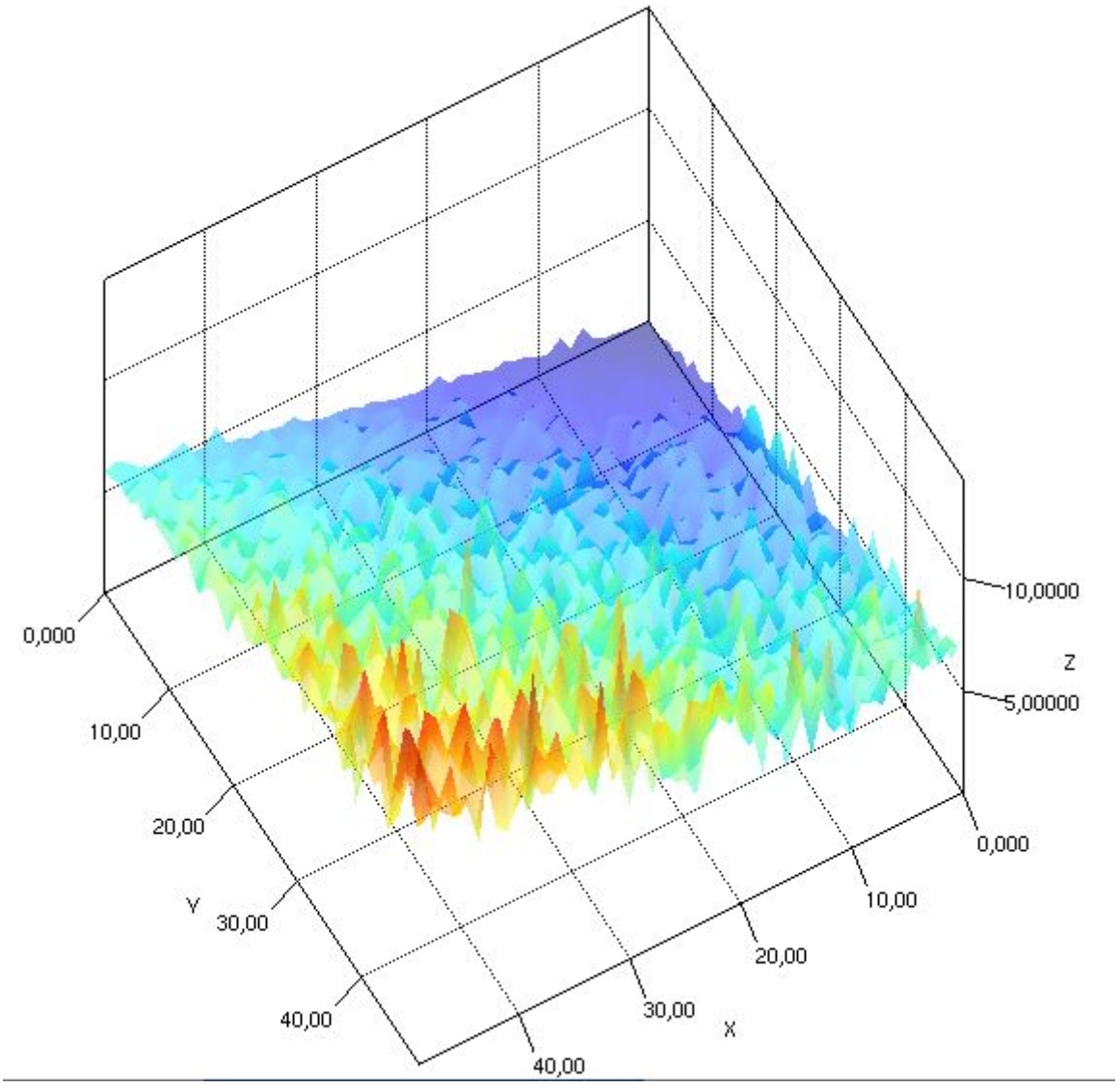


Average forStandard Java: 114.19957ms
Average forLibrary: 108.33309333333334ms

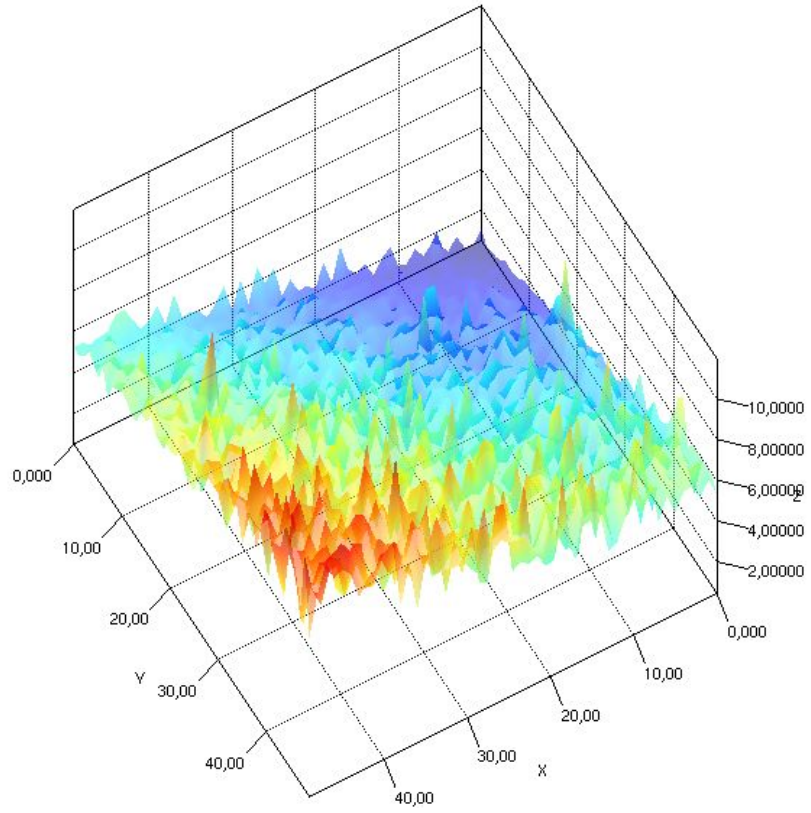
Dla M=50

```
Average forStandard Java: 290.943868ms
Average forLibrary: 273.236534ms
```

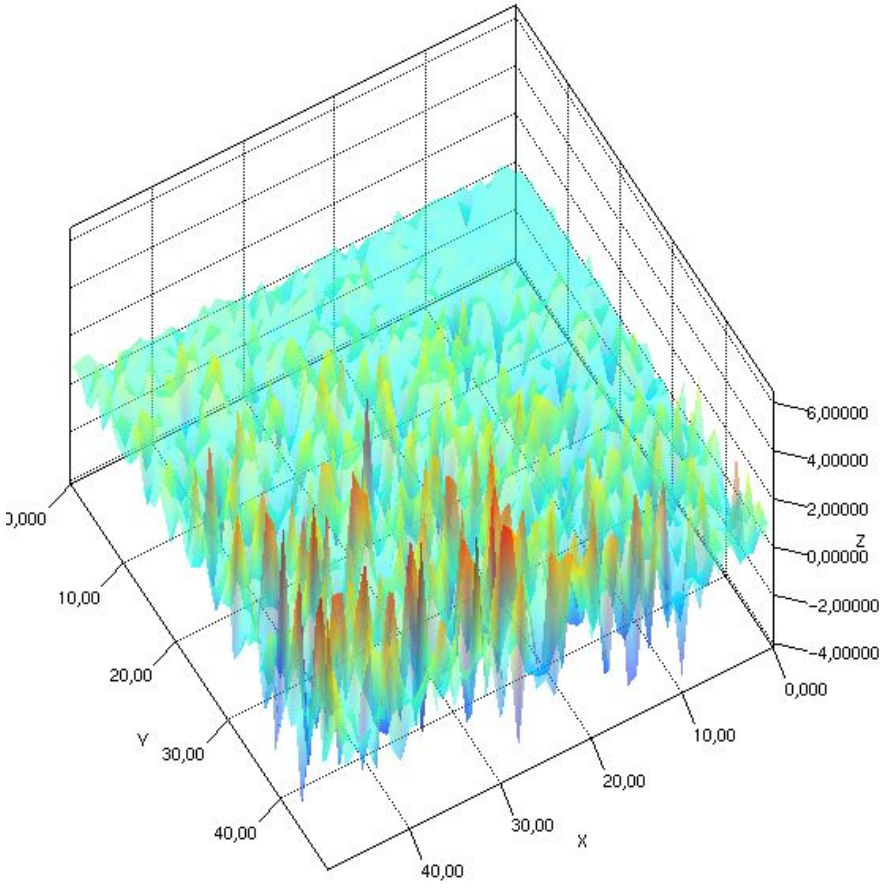
Standard:



Library:



Difference Standard - Library:

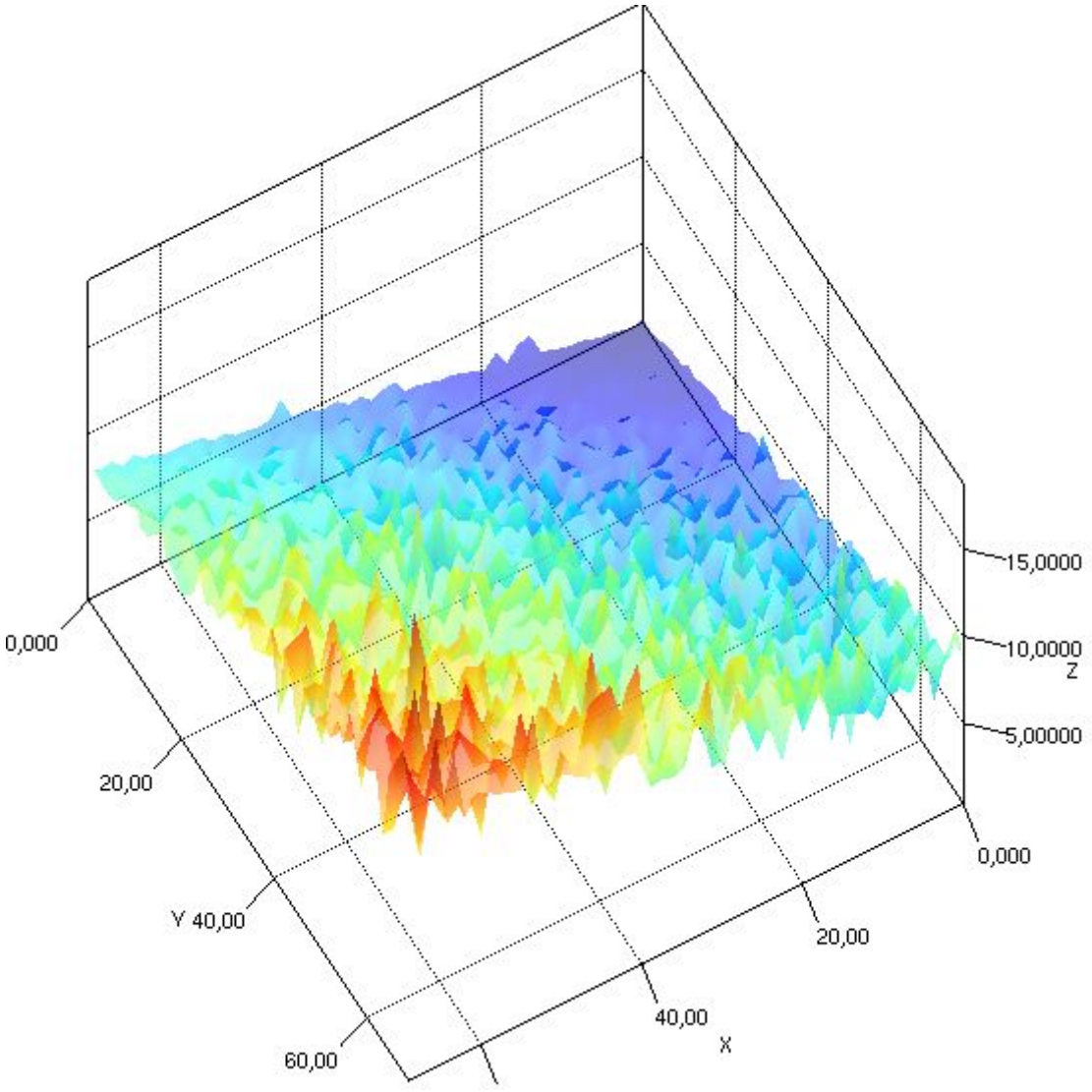


Average forStandard Java: 290.943868ms
Average forLibrary: 273.236534ms

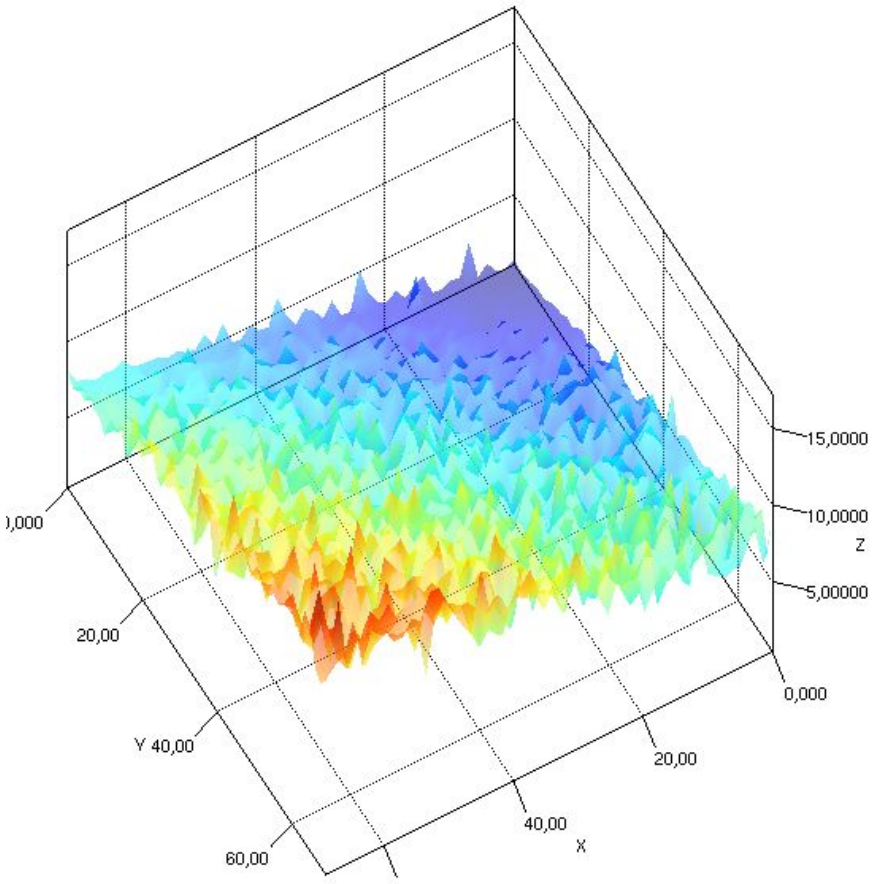
Dla M=70

```
Average forStandard Java: 562.8692614285715ms
Average forLibrary: 521.4832085714286ms
```

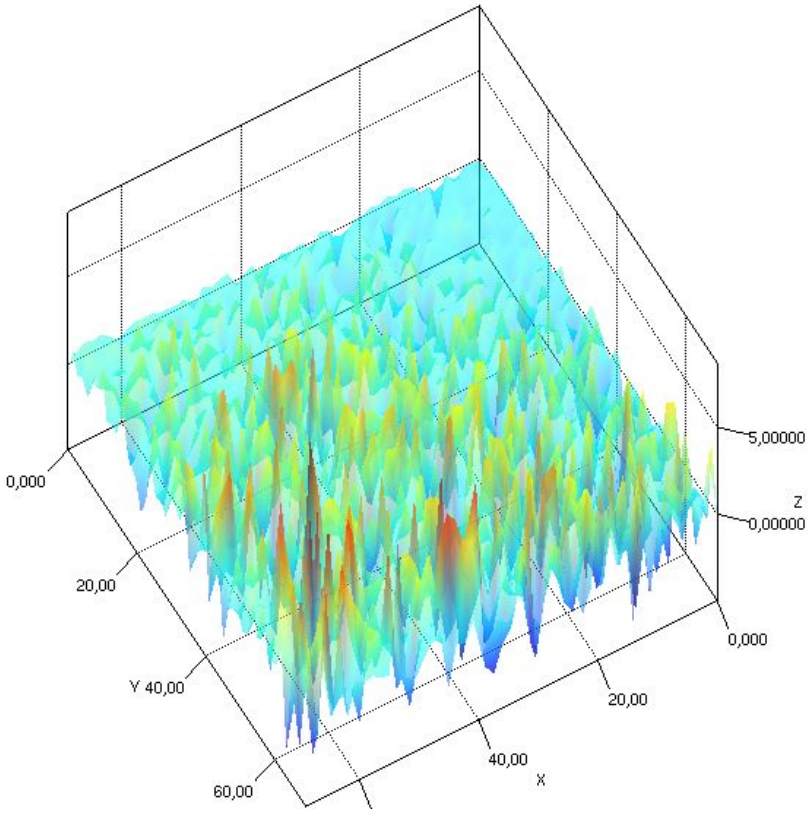
Standard:



Library:

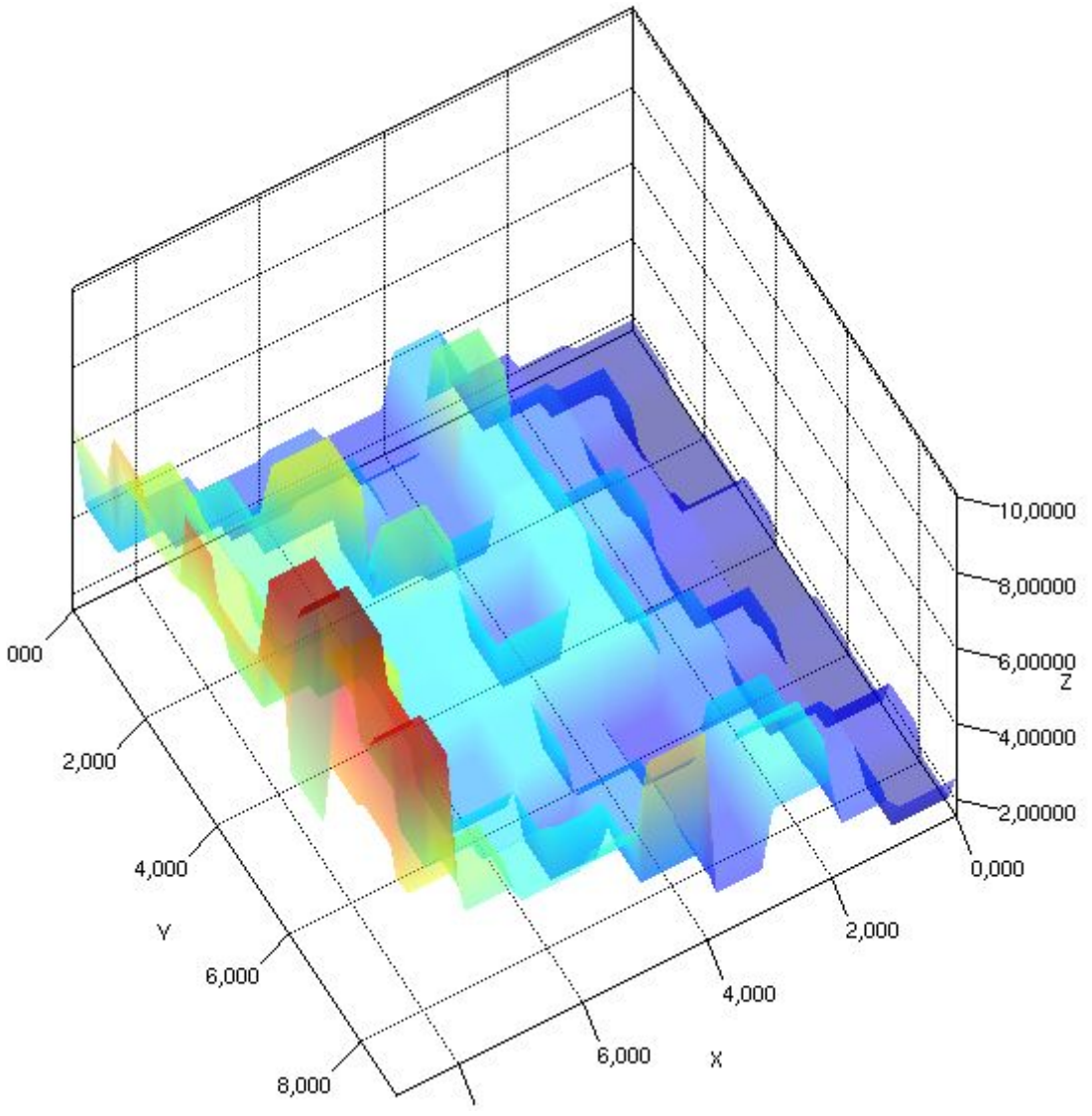


Różnica Standard - Library:

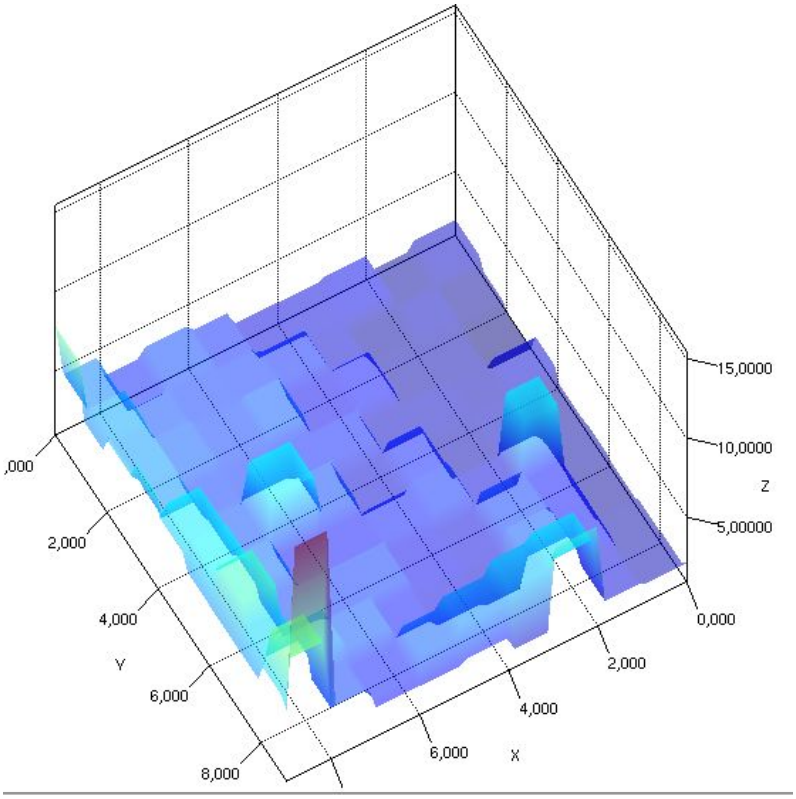


Trochę ciekawsze wyniki obserwujemy po zwiększeniu liczby produkowanych produktów przez producenta z 10 na 100:

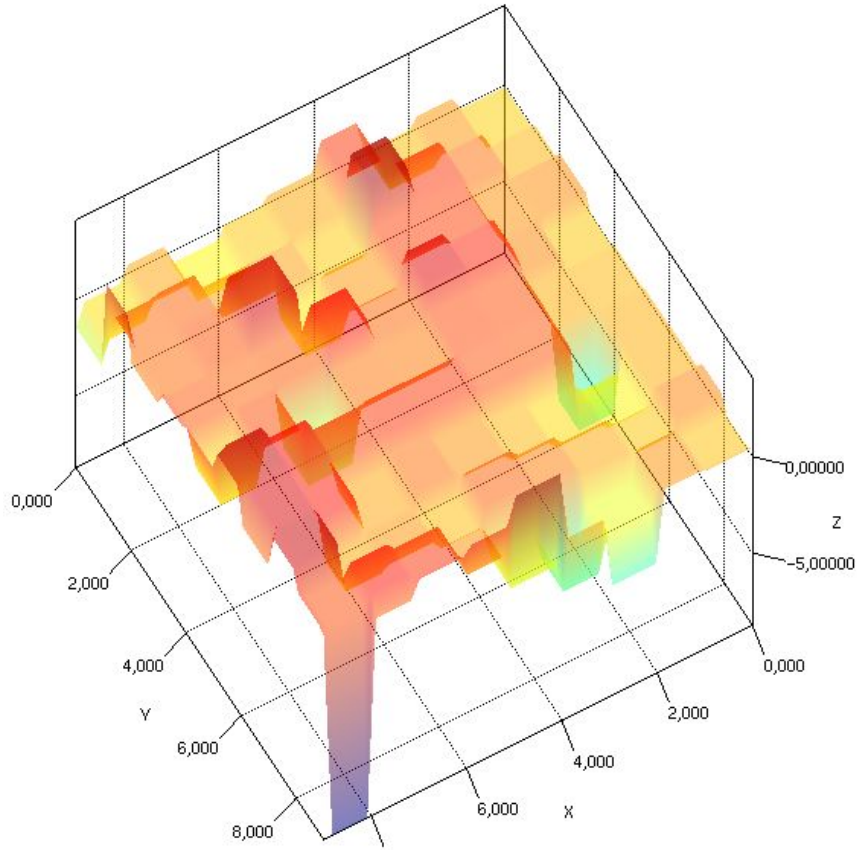
M=10
Average forStandard Java: 45.47908ms
Average forLibrary: 36.40097ms
Standard:



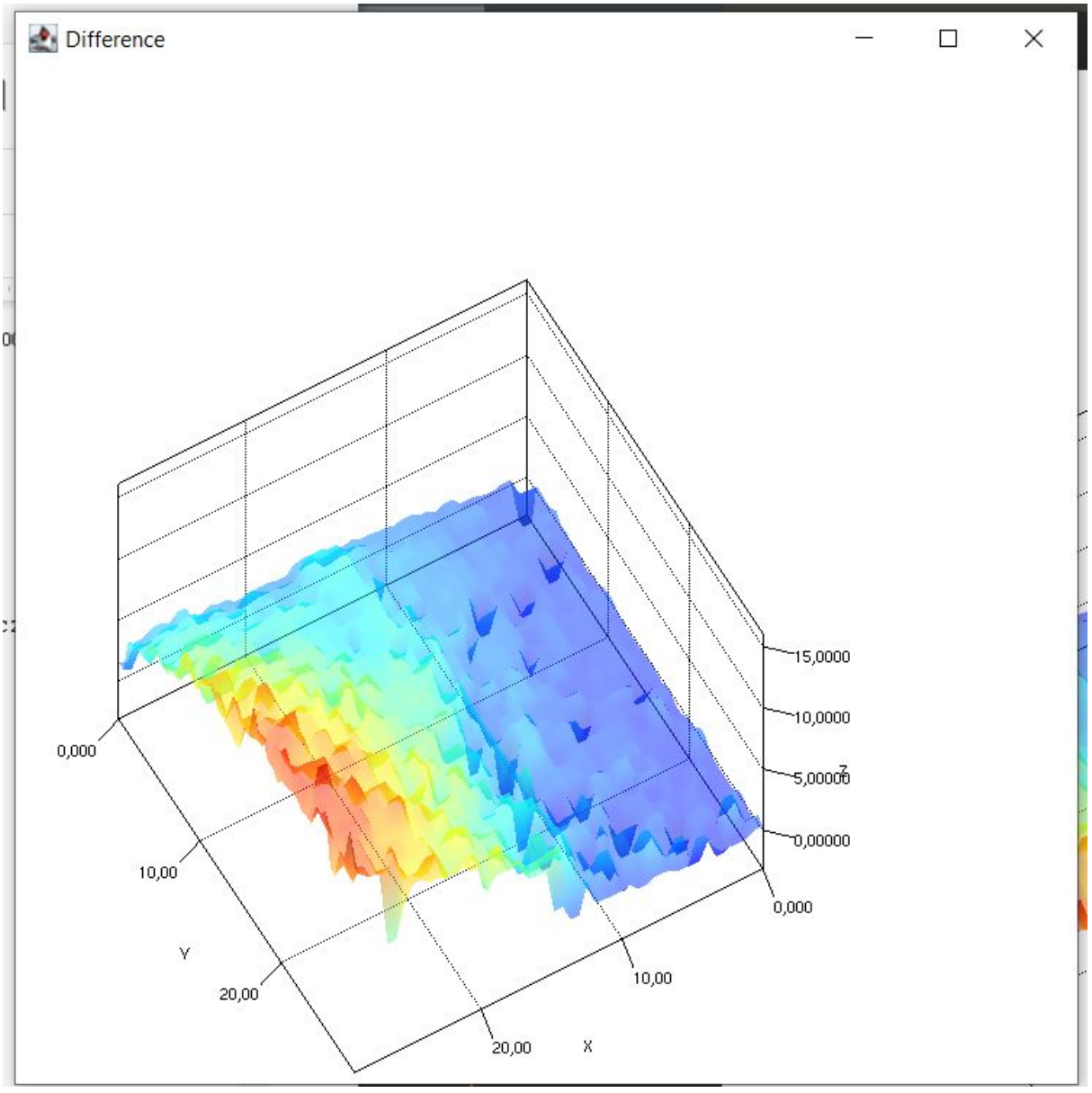
Library:



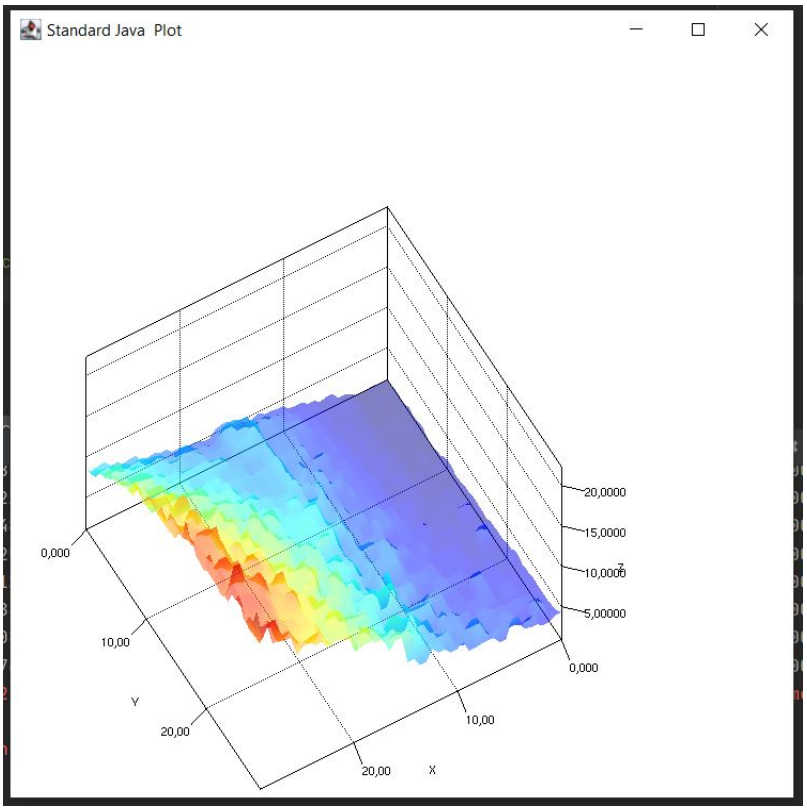
Różnica: Standard - Library:



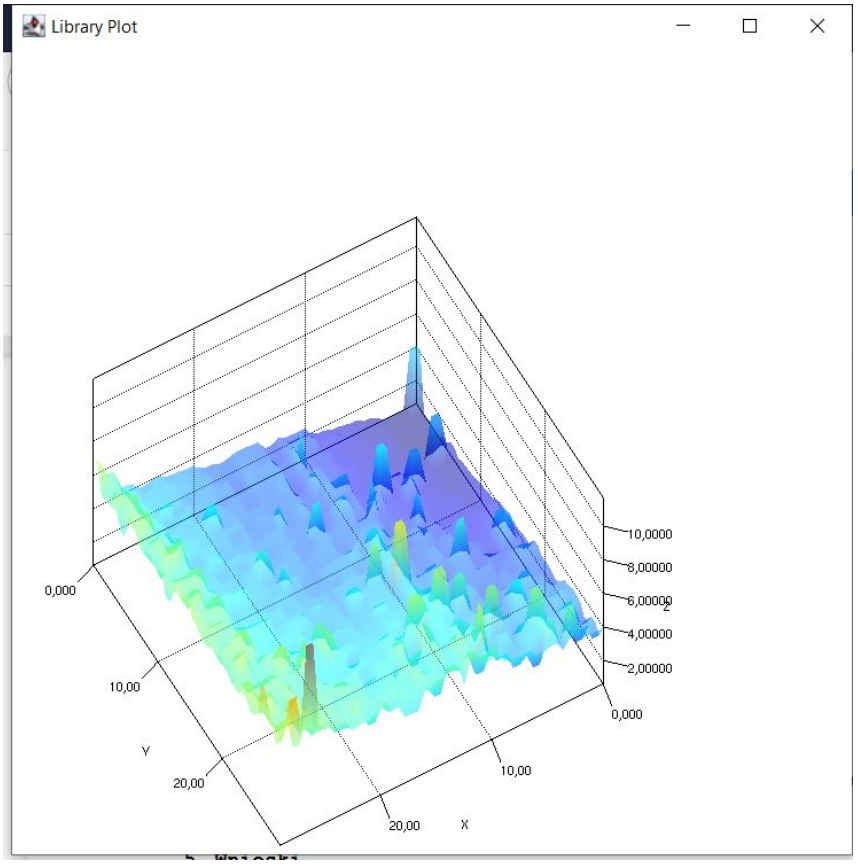
M = 30
Average forStandard Java: 247.98583333333335ms
Average forLibrary: 119.1812ms
Różnica:



Standard



Library:

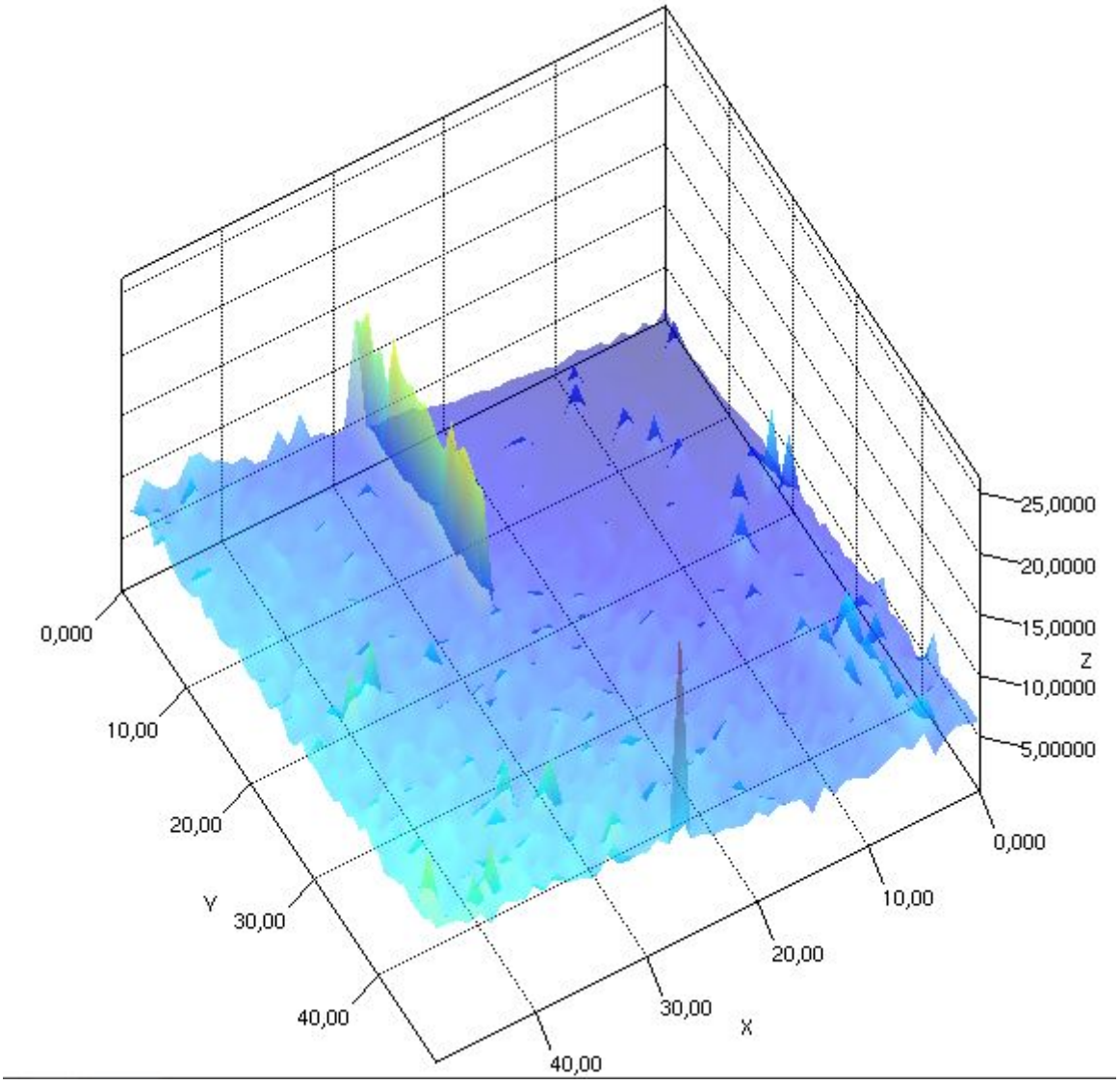


M=50

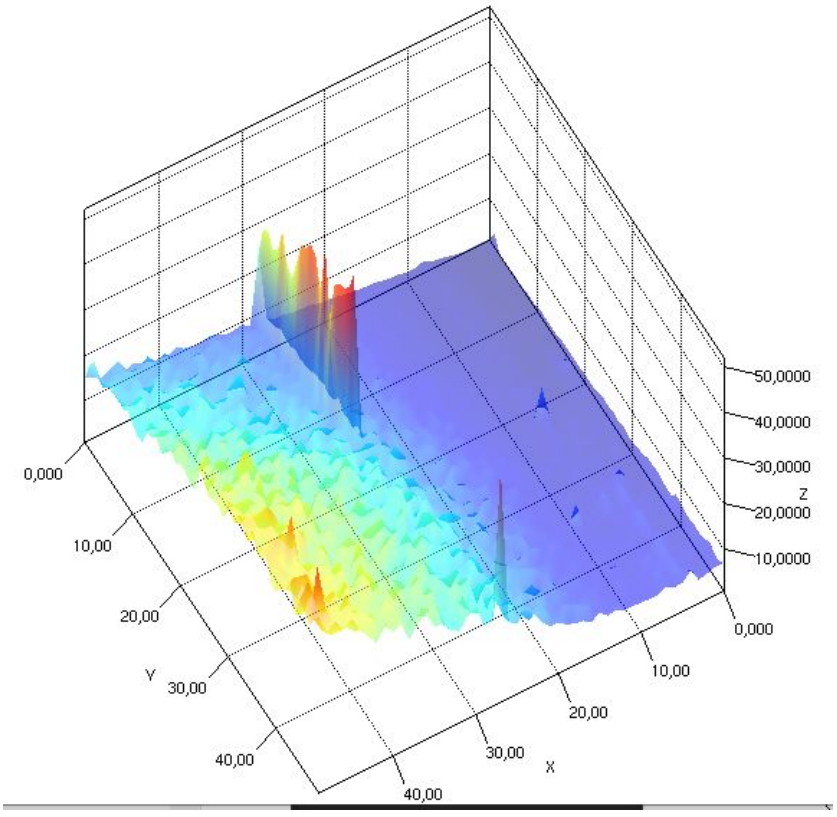
```
Average forStandard Java: 712.593646ms
Average forLibrary: 305.050012ms
```

Average forStandard Java: 712.593646ms
Average forLibrary: 305.050012ms

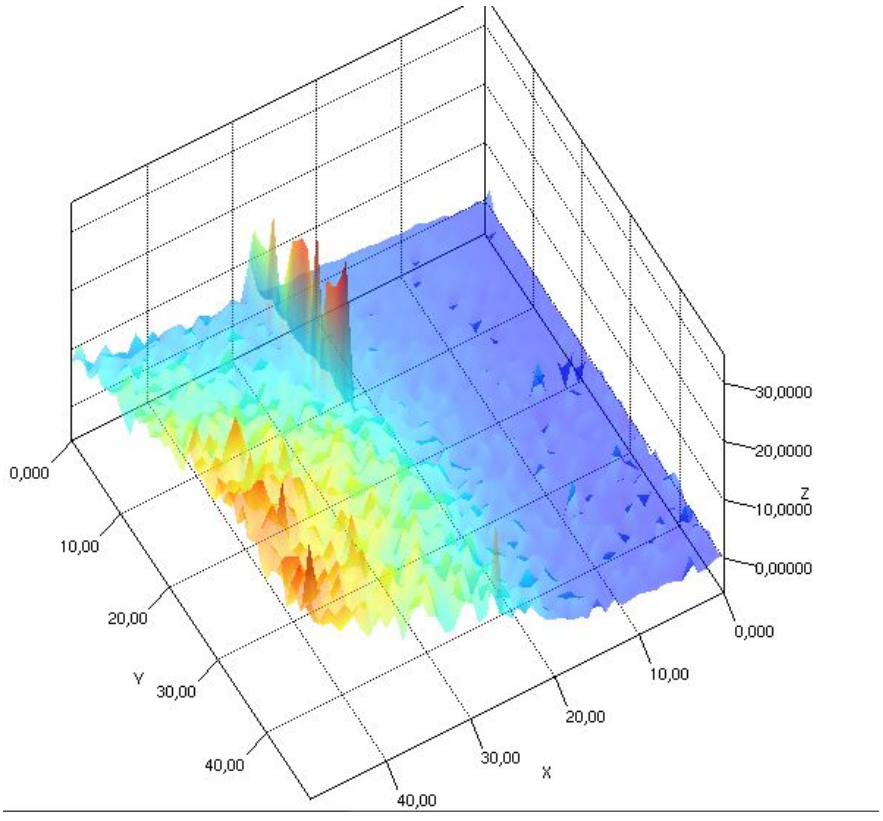
Library:



Standard:



Difference:

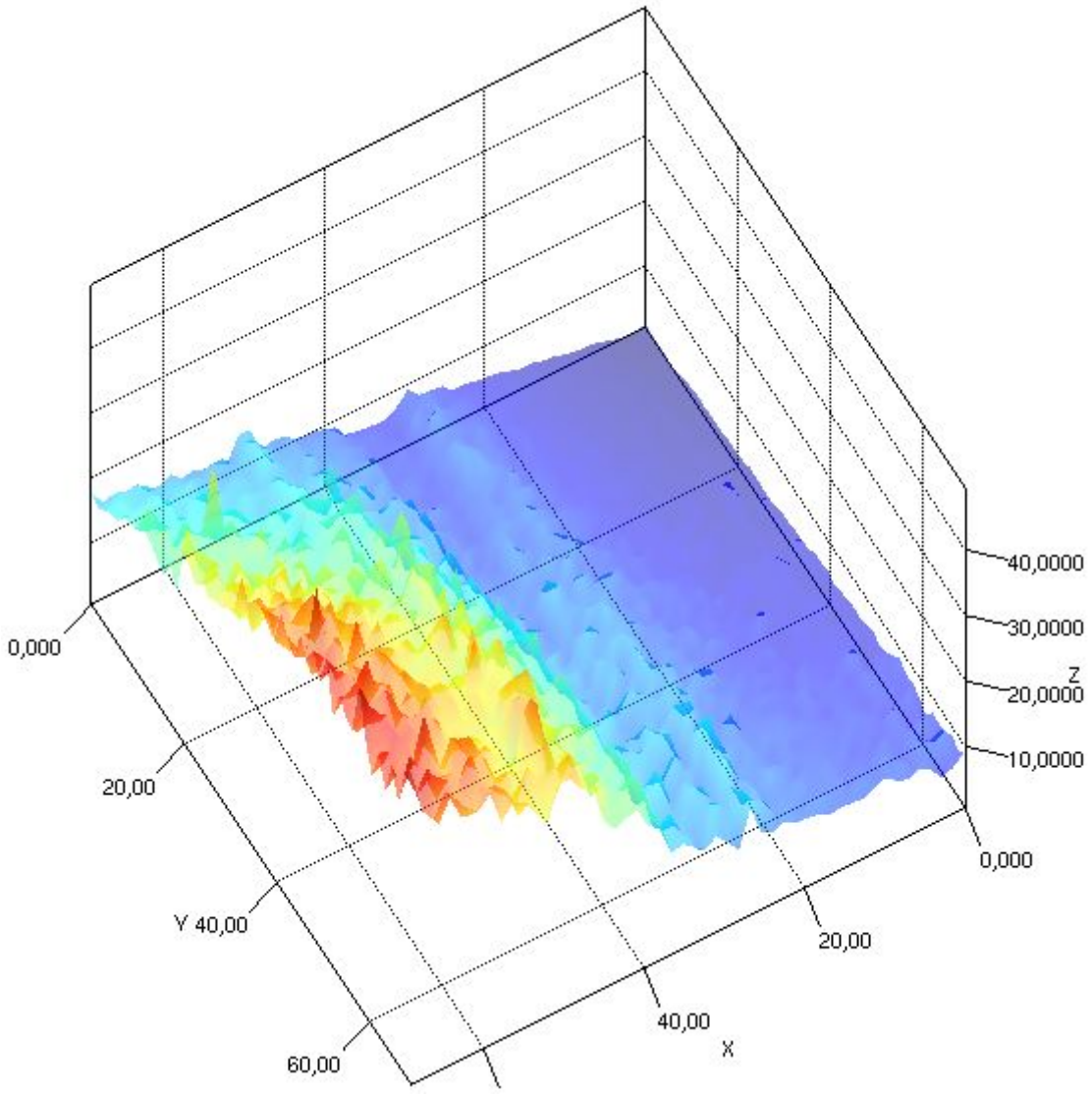


M = 70:

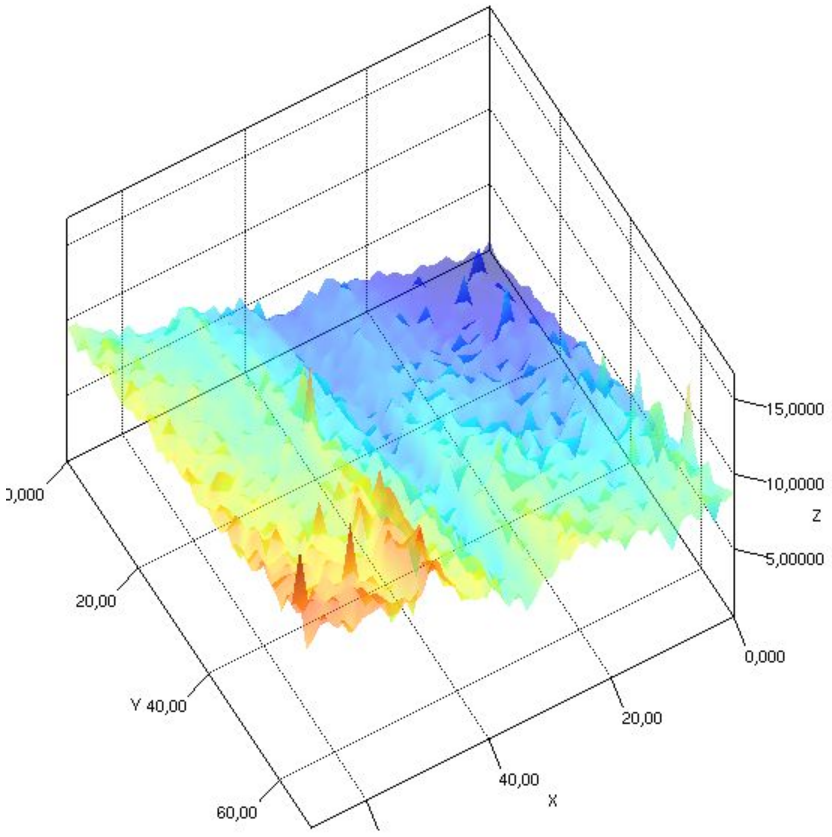
```
Average forStandard Java: 1184.0495071428572ms
Average forLibrary: 517.5778542857142ms
```

Average forStandard Java: 1184.0495071428572ms
Average forLibrary: 517.5778542857142ms

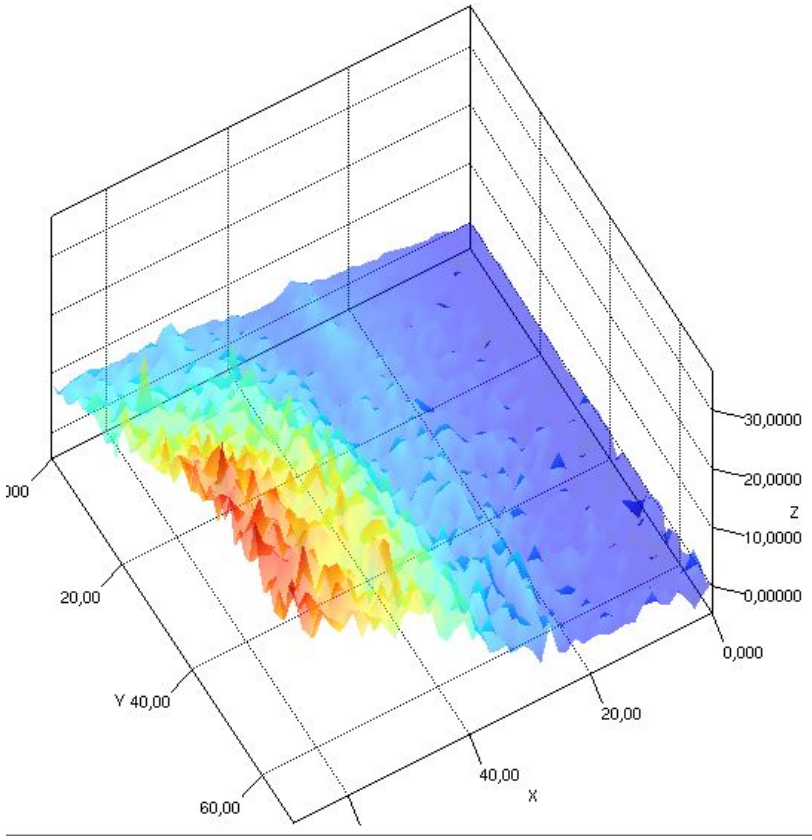
Standard:



Library:



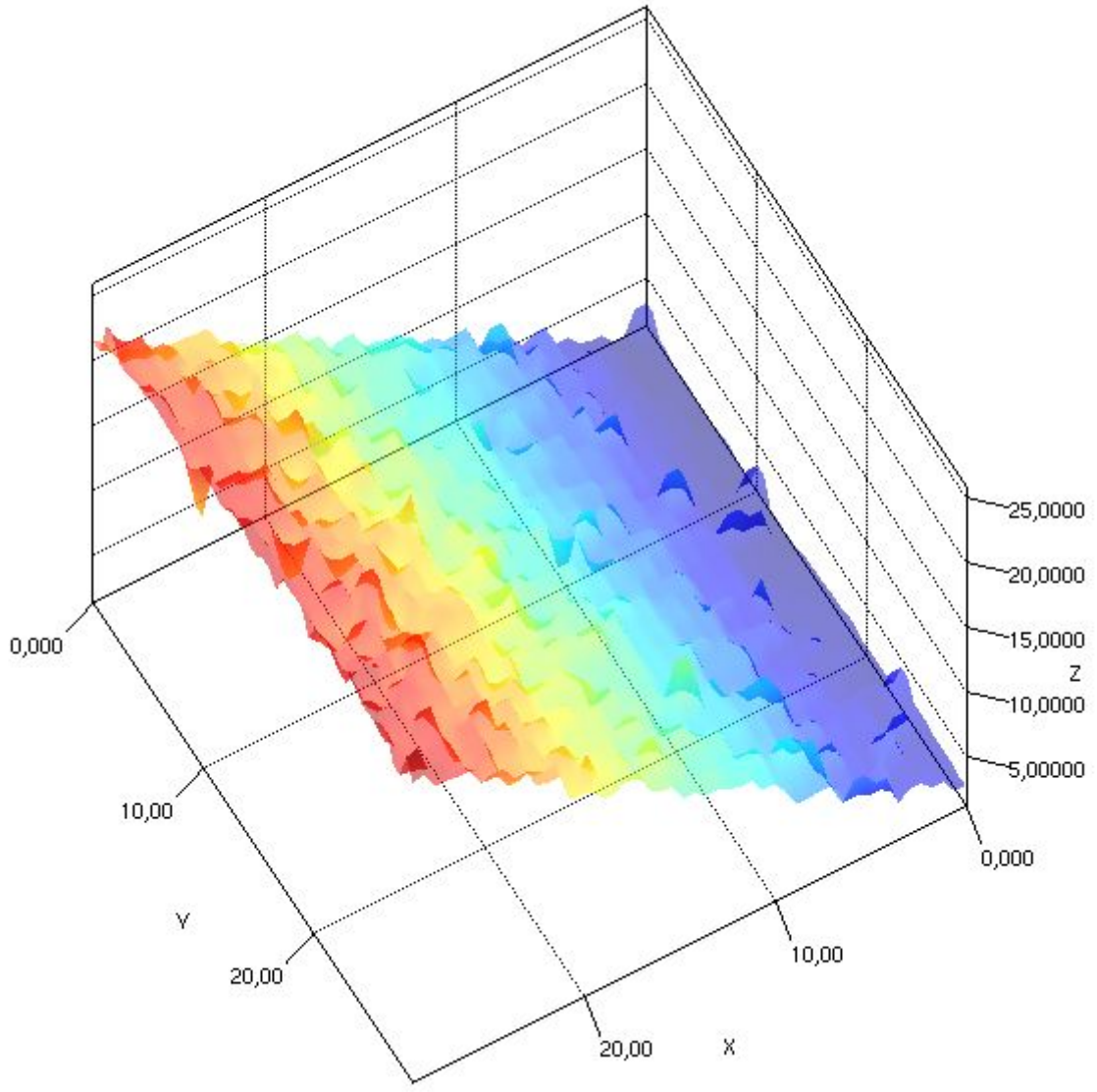
Difference:



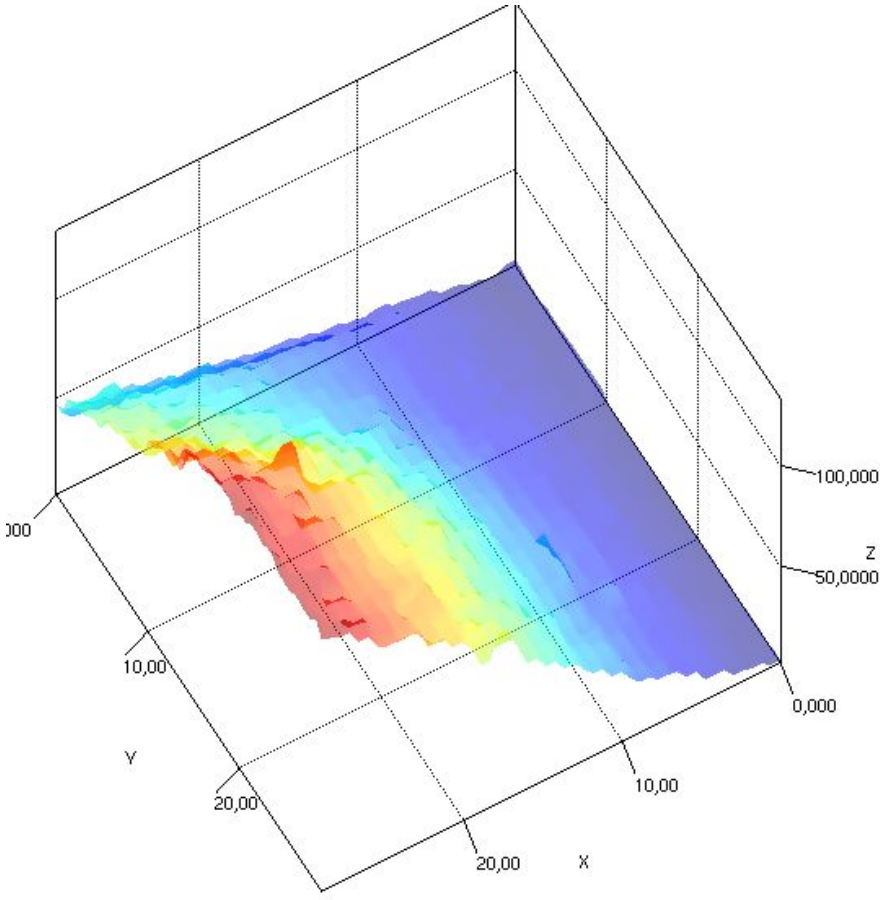
Po zwiększeniu liczby produkowanych produktów przez producenta z 100 na 1000:

M=30
Average forStandard Java: 1542.0396733333334ms
Average forLibrary: 380.50951ms

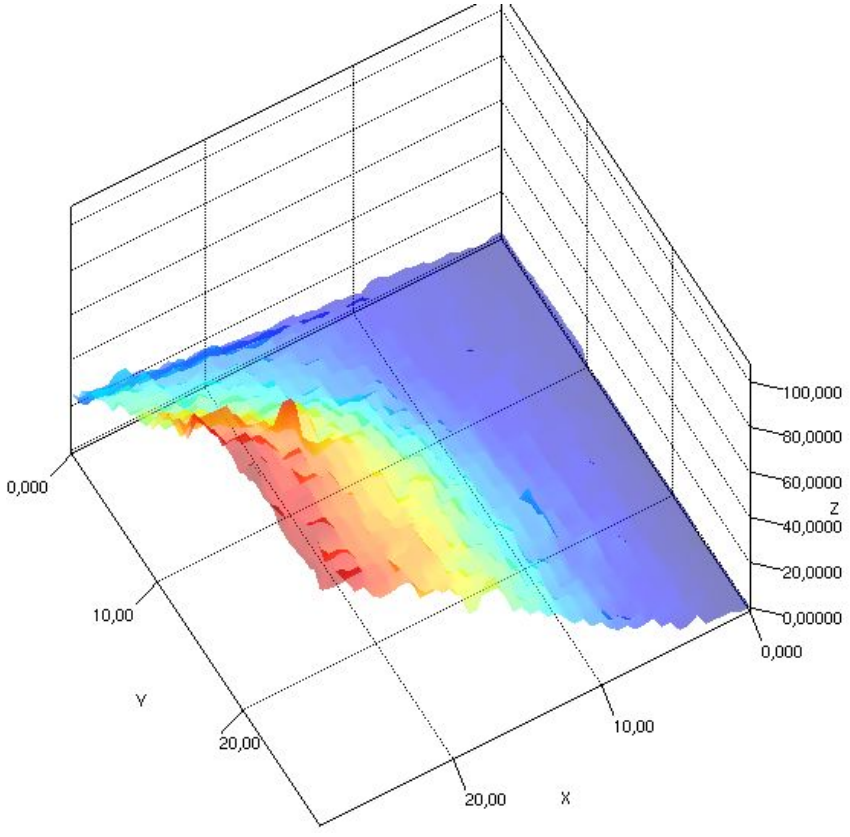
Library:



Standard:



Difference:

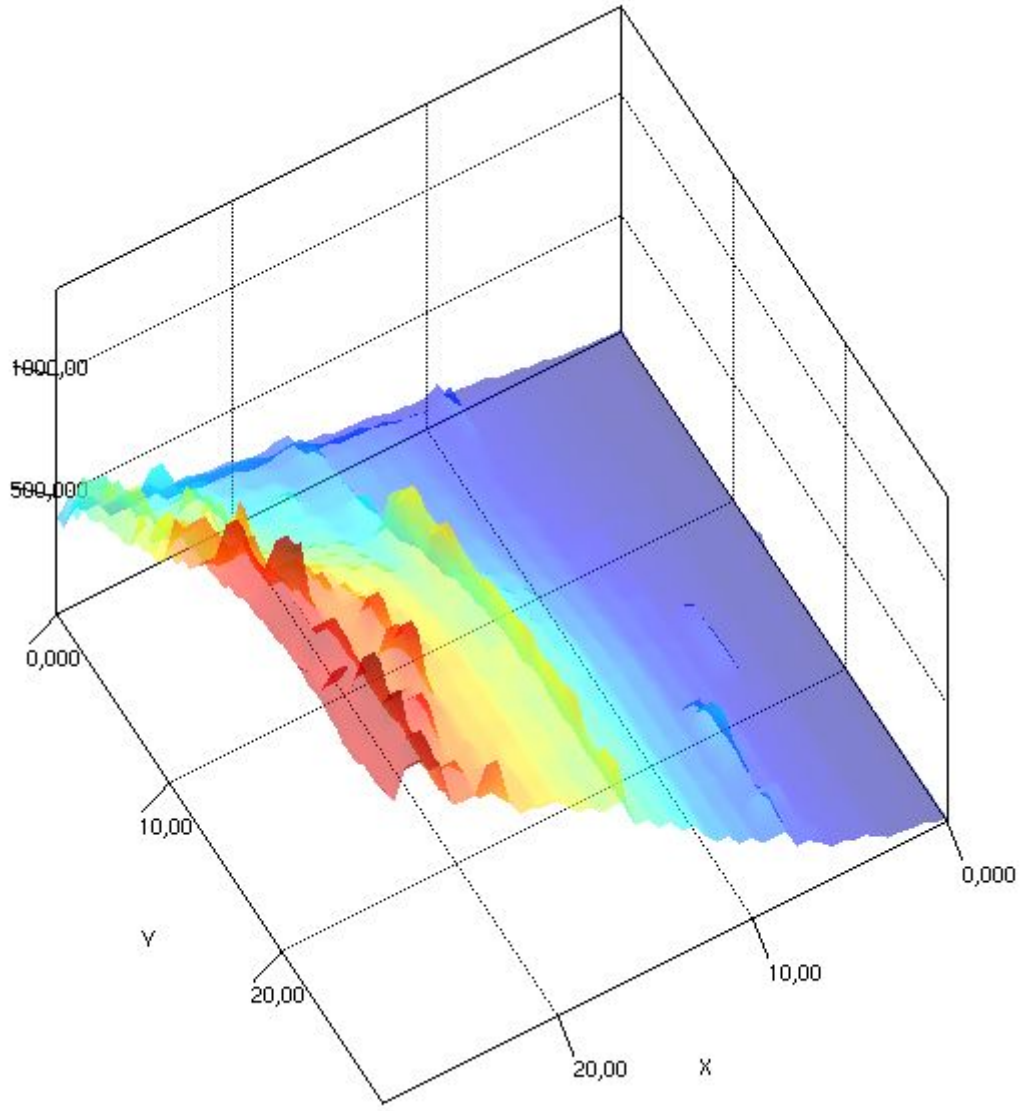


Zwiększenie liczby produkowanych produktów przez producenta z 1000 na 10000:

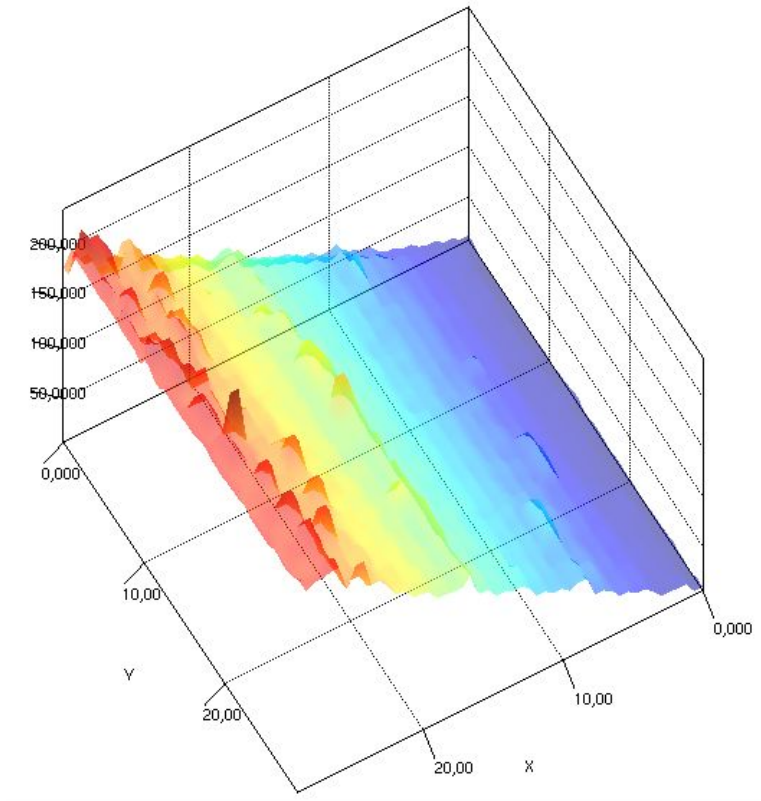
M=30
Average forStandard Java: 15377.509013333334ms
Average forLibrary: 3300.0351233333336ms

```
Average forStandard Java: 15377.509013333334ms
Average forLibrary: 3300.0351233333336ms
```

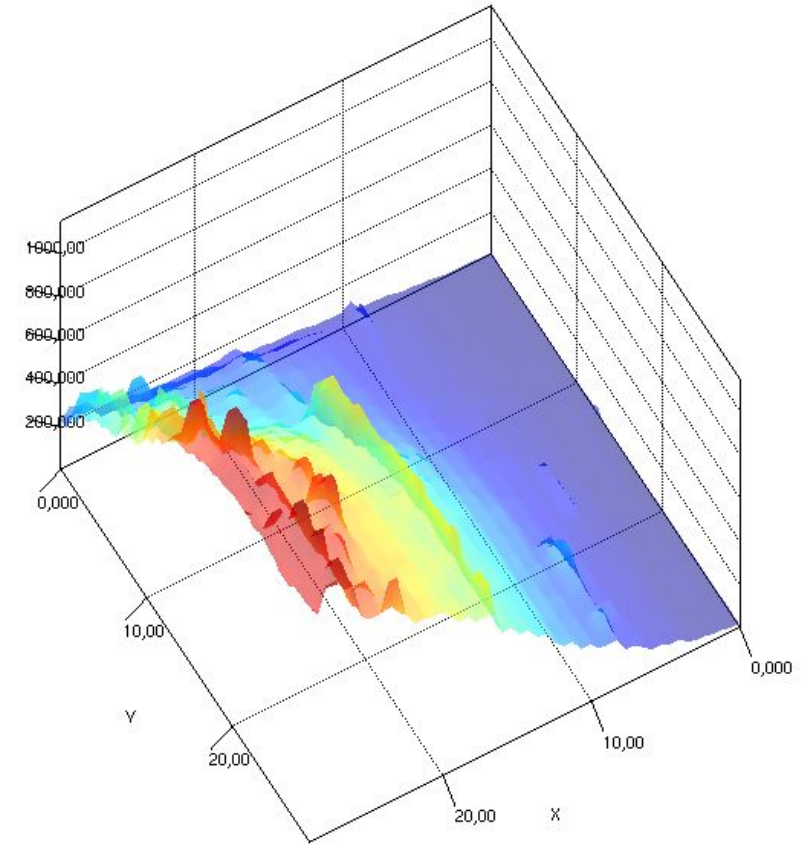
Standard:



Library:



Difference:



5. Wnioski

rozmiar bufora	liczba operacji dla jednego producenta	sredni wynik dla standard w ms	średni wynik dla library w ms
10	10	16.56	16.40
30	10	114.19	108.33
50	10	290.94	273.23
70	10	562.86	521.48
10	100	45.47	36.40
30	100	247.98	119.18
50	100	712.59	305.05
70	100	1184.04	517.57
30	1000	1542.03	380.50
30	10000	15377.50	3300.03

W testach liczba operacji jakie wykonają łącznie producenci jest równomiernie rozkładana na wszystkich konsumentów.

Kolory we wnioskach odnoszą się do wierszy w tabeli.

- Dla małej liczby operacji dla jednego producenta czasy wykonania dla obu implementacji są podobne. Mała liczba operacji oznacza krótki czas życia producentów i konsumentów. W takim przypadku sposoby praktycznie się nie różnią.

- Gdy zwiększamy liczbę operacji dla jednego producenta (a co za tym idzie i dla konsumenta) czasy już znacząco się różnią. Przy dziesięciokrotnym wzroście operacji mamy dwukrotny wzrost czasu dla implementacji standardowej w stosunku do implementacji z pakietu java utils.
- Przy dużo większej liczbie operacji widać znaczący spadek wydajności implementacji standardowej. Duża liczba operacji powoduje, że producenci i konsumerzy żyją długo i często dochodzi do rywalizacji o zasób. Gdy do tego dochodzi jesteśmy w stanie wykryć różnice między naszymi

dwiema implementacjami. Dla tego przypadku implementacja standard jest około 4 razy wolniejsza od library.

- Przy liczbie operacji 10000 dla jednego producenta uzyskujemy podobny wynik jak przy 1000 operacjach.

Interpretacja wykresów:

Po obserwacji wykresów można stwierdzić, że liczba producentów i konsumentów znacząco wpływa na szybkość działania bufora. Gdy jest ich zbyt dużo często dochodzi do wyścigu co źle wpływa na pracę bufora. Liczba producentów w badanych przykładach wpływa liniowo na całość wysyłanych dóbr, zatem czas rośnie liniowo wraz ze wzrostem producentów. Dla małej liczby dóbr dobór sposobu rozwiązywania w zasadzie nie ma znaczenia bo czasy działania są podobne. Przy częstym zjawisku "wyścigu" wersja library wypada lepiej od standard.

Użycie mechanizmów z pakietu Java Concurrency Utilities okazało się lepsze od własnej implementacji z użyciem notify()/wait() oraz bloku synchronized.

6. Bibliografia

http://home.agh.edu.pl/~funika/tw/lab4/
http://www.jzy3d.org/

Radosław Kopeć