Teoria współbieżności

Laboratorium 4
Java Concurrency Utilities

Albert Gierlach

31.10.2020

1. Zadanie

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
- Przeprowadzic porownanie wydajnosci vs. rozne parametry, zrobic wykresy i je skomentowac

2. Koncept rozwiązania

Rozwiązanie będzie opierać się na zaimplementowaniu obu metod oraz przeprowadzeniu testów dla różnych konfiguracji. Program w dużej mierze, będzie zależał od losowości, więc dla każdej konfiguracji zostanie przeprowadzone 200 iteracji, a wynikiem będzie średni czas działania programu dla zadanych parametrów. Każdy wątek będzie wykonywał 100 iteracji dodania i pobrania z bufora.

Pierwsza metoda będzie opierać się monitorach. Wątki producentów będą wstrzymywane, gdy w buforze będzie zbyt wiele elementów, a wątki konsumentów gdy będzie zbyt mało. Nad wykonaniem będzie czuwać nadzorca (executor), który zakończy pracę wątków, jeśli jedna ze stron zakończy swoją pracę.

Druga metoda opiera się o gotowe mechanizmy biblioteki Java Concurrent. Wykorzystana zostanie kolejka blokująca (*BlockingQueue*) oraz pula wątków. Podobnie jak w pierwszym wariancie - zostanie wykorzystany nadzorca.

Czasy zostaną zapisane do pliku i na ich podstawie zostaną narysowane wykresy.

3. Implementacja oraz wyniki

```
public interface IBuffer {
    void unregisterProducer();
    void unregisterConsumer();

    int maxSize();

    void put(int[] products) throws InterruptedException;

    void get(List<Integer> results, int howMany) throws InterruptedException;

    boolean isAnySideInterested();
}

public interface IExecutor {
    void submit(Thread t);

    void shutdownNow();

    void awaitTermination();
}


class Producer extends Thread {
    private final IBuffer buffer;
    private final Random random = new Random();
    private final int produceLimit;
    private final int iterations;
    private final IExecutor executor;

    public Producer(IExecutor ex, IBuffer buf, int iters) {
        iterations = iters;
        buffer = buf;
        produceLimit = buffer.maxSize() / 2;
        executor = ex;
    }

    public void run() {
        for(int i = 0; i < iterations; i++) {
            var howMany = Math.max(1, random.nextInt(produceLimit)-1);
            int[] data = new int[howMany];
```

```
            IntStream.range(0, howMany).forEach(j -> {
                data[j] = j;
            });

            try {
                buffer.put(data);
            } catch (InterruptedException exception) {
                break;
            }
        }
        buffer.unregisterProducer();
        if(!buffer.isAnySideInterested()){
            executor.shutdownNow();
        }
    }
}

class Consumer extends Thread {
    private final IBuffer buffer;
    private final int consumeLimit;
    private final int iterations;
    private final Random random = new Random();
    private final IExecutor executor;

    public Consumer(IExecutor ex, IBuffer buf, int iters) {
        iterations = iters;
        buffer = buf;
        consumeLimit = buffer.maxSize() / 2;
        executor = ex;
    }

    public void run() {
        for (int i = 0; i < iterations; i++) {
            var howMany = Math.max(1, random.nextInt(consumeLimit)-1);
            List<Integer> results = new LinkedList<>();
            try {
                buffer.get(results, howMany);
            } catch (InterruptedException exception) {
                break;
            }
        }

        buffer.unregisterConsumer();
```

```
            if (!buffer.isAnySideInterested()) {
                executor.shutdownNow();
            }
        }
    }

    public class MyExecutor implements IExecutor {
        private final List<Thread> workers = new LinkedList<>();

        @Override
        public void submit(Thread t){
            workers.add(t);
        }

        @Override
        public void shutdownNow(){
            workers.forEach(Thread::interrupt);
        }

        @Override
        public void awaitTermination() {
            workers.forEach(Thread::start);

            workers.forEach(t -> {
                try {
                    t.join();
                } catch (InterruptedException ignored) {
                }
            });
        }
    }

    public class ExecutorServiceWrapper implements IExecutor {
        private final ExecutorService executorService;
        private final List<Thread> workers = new LinkedList<>();

        public ExecutorServiceWrapper(int threadNum) {
            executorService = Executors.newFixedThreadPool(threadNum);
        }

        @Override
        public void submit(Thread t) {
            workers.add(t);
```

```
    }

    @Override
    public void shutdownNow() {
        executorService.shutdownNow();
    }

    @Override
    public void awaitTermination() {
        workers.forEach(executorService::submit);

        executorService.shutdown();
        try {
            executorService.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
        } catch (InterruptedException ignored) {
        }
    }
}

class Buffer implements IBuffer {
    private final LinkedList<Integer> buffer = new LinkedList<>();
    private final int maxBufferSize;
    private final Object lock = new Object();
    private Integer producersRunning;
    private Integer consumersRunning;

    public Buffer(int size, int m, int n) {
        maxBufferSize = size * 2;
        producersRunning = m;
        consumersRunning = n;
    }

    @Override
    public void unregisterProducer() {
        synchronized (lock) {
            producersRunning--;
        }
    }

    @Override
    public void unregisterConsumer() {
        synchronized (lock) {
            consumersRunning--;
```

```
    }
}

@Override
public boolean isAnySideInterested() {
    return producersRunning > 0 && consumersRunning > 0;
}

@Override
public int maxSize() {
    return maxBufferSize;
}

@Override
public synchronized void put(int[] products) throws InterruptedException {
    while (isAnySideInterested() &&
            (buffer.size() + products.length >= maxBufferSize)) {
        wait();
    }

    if (!isAnySideInterested()) {
        return;
    }

    buffer.addAll(Arrays.stream(products).boxed().collect(Collectors.toList()));
    notifyAll();
}

@Override
public synchronized void get(List<Integer> results,
                             int howMany) throws InterruptedException {
    while (isAnySideInterested() && buffer.size() < howMany) {
        wait();
    }

    if (!isAnySideInterested()) {
        return;
    }

    var sublist = buffer.subList(0, howMany);
    for(int i = 0; i < howMany; i++){
        var v = sublist.get(i);
        results.add(v);
```

```
            }
            sublist.clear();
            notifyAll();
        }
    }

    class Buffer2 implements IBuffer {
        private final int maxBufferSize;
        private final AtomicInteger producersRunning;
        private final AtomicInteger consumersRunning;
        private final BlockingQueue<Integer> buffer;

        public Buffer2(int size, int m, int n) {
            maxBufferSize = size * 2;
            producersRunning = new AtomicInteger(m);
            consumersRunning = new AtomicInteger(n);
            buffer = new ArrayBlockingQueue<>(maxBufferSize);
        }

        @Override
        public void unregisterProducer() {
            producersRunning.decrementAndGet();
        }

        @Override
        public void unregisterConsumer() {
            consumersRunning.decrementAndGet();
        }

        @Override
        public boolean isAnySideInterested() {
            return producersRunning.get() > 0 && consumersRunning.get() > 0;
        }

        @Override
        public int maxSize() {
            return maxBufferSize;
        }

        @Override
        public void put(int[] products) throws InterruptedException {
            for (var v : products) {
                if(!isAnySideInterested()){
```

```
                return;
            }
            buffer.put(v);
        }
    }

    @Override
    public void get(List<Integer> results, int howMany) throws InterruptedException {

        while (results.size() < howMany) {
            if(!isAnySideInterested()){
                return;
            }
            Integer v = buffer.take();
            results.add(v);
        }
    }
}

public class PKMain {
    enum MODE {
        MODE_MONITORS,
        MODE_CONCURRENT_LIB
    }

    public static void main(String[] args) {
        var producers = List.of(3, 10, 20);
        var consumers = List.of(3, 10, 20);
        var bufferSizes = List.of(5, 10, 50, 100);

        var results = new LinkedList<String>();

        for (var p : producers) {
            for (var c : consumers) {
                for (var bs : bufferSizes) {
                    var avg1 = testCaseWrapper(p, c, bs, MODE.MODE_MONITORS);
                    var avg2 = testCaseWrapper(p, c, bs, MODE.MODE_CONCURRENT_LIB);


                    results.add(p + ", " + c + ", " +
                            bs + ", " + round(avg1, 2) + ", " + round(avg2, 2));
                }
            }
```

```
    }

    Path out = Paths.get("results.csv");
    try {
        Files.write(out, results, Charset.defaultCharset());
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static double round(double v, int places){
    v = Math.round(v * Math.pow(10, places));
    v = v / Math.pow(10, places);
    return v;
}

public static double testCaseWrapper(int prodNum, int consNum,
            int halfBufSize, MODE mode) {
    var times = new LinkedList<Long>();

    IntStream.range(0, 200).forEach(i -> {
        long before = System.currentTimeMillis();

        if (mode == MODE.MODE_CONCURRENT_LIB) {
            testConcurrentLibrary(prodNum, consNum, halfBufSize);
        } else {
            testMonitors(prodNum, consNum, halfBufSize);
        }

        long after = System.currentTimeMillis();
        long diff = after - before;
        times.add(diff);
    });

    return times.stream().mapToLong(i -> i).average().getAsDouble();
}

public static void testConcurrentLibrary(int m, int n, int M) {
    var buffer = new Buffer2(M, m, n);
    var executor = new ExecutorServiceWrapper(n+m);

    runTasks(m, n, buffer, executor);
}
```

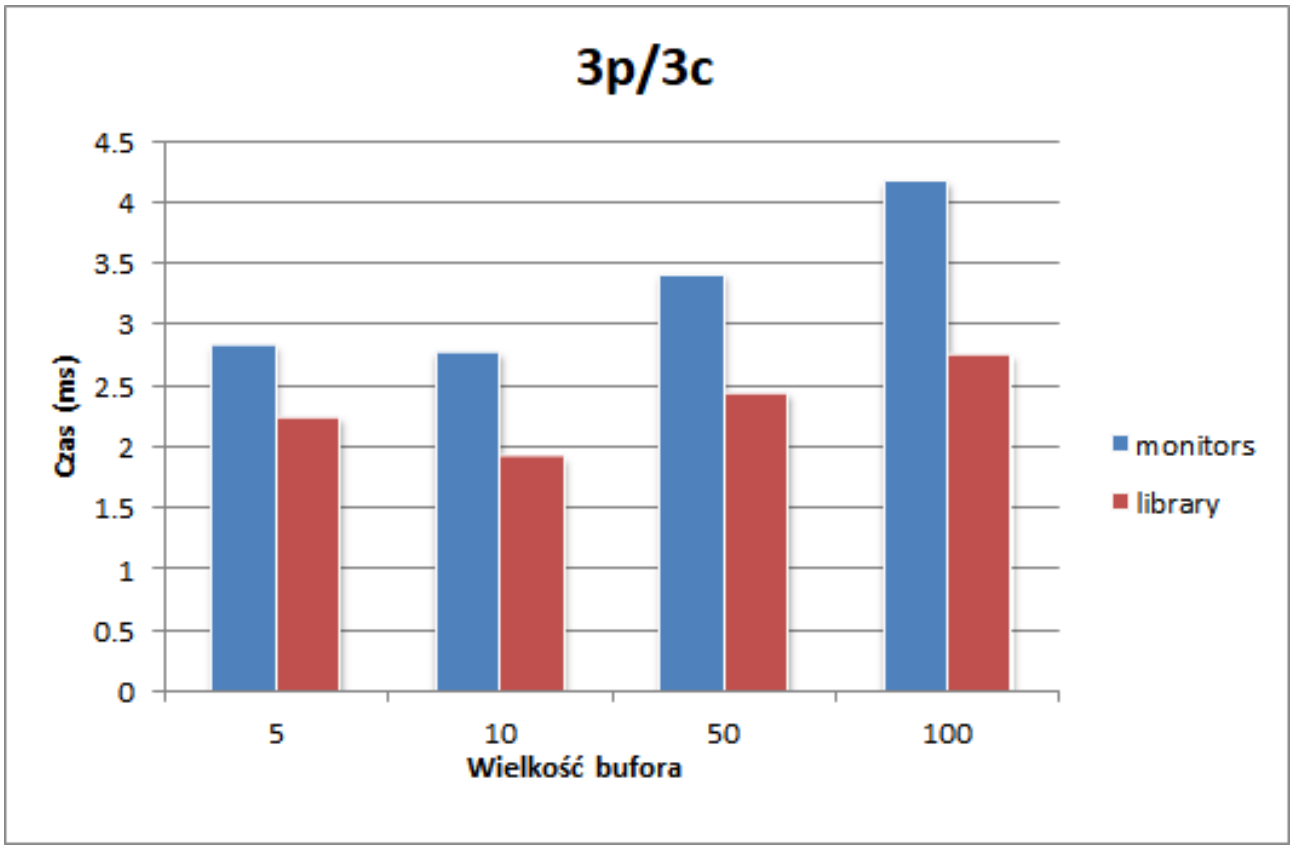
```
    public static void testMonitors(int m, int n, int M) {
        var buffer = new Buffer(M, m, n);
        var executor = new MyExecutor();

        runTasks(m, n, buffer, executor);
    }

    public static void runTasks(int m, int n, IBuffer buffer, IExecutor executor){
        IntStream.range(0, n).forEach(i ->
                executor.submit(
                        new Consumer(executor, buffer, 100)
                ));
        IntStream.range(0, m).forEach(i ->
                executor.submit(
                        new Producer(executor, buffer, 100)
                ));

        executor.awaitTermination();
    }
}
```

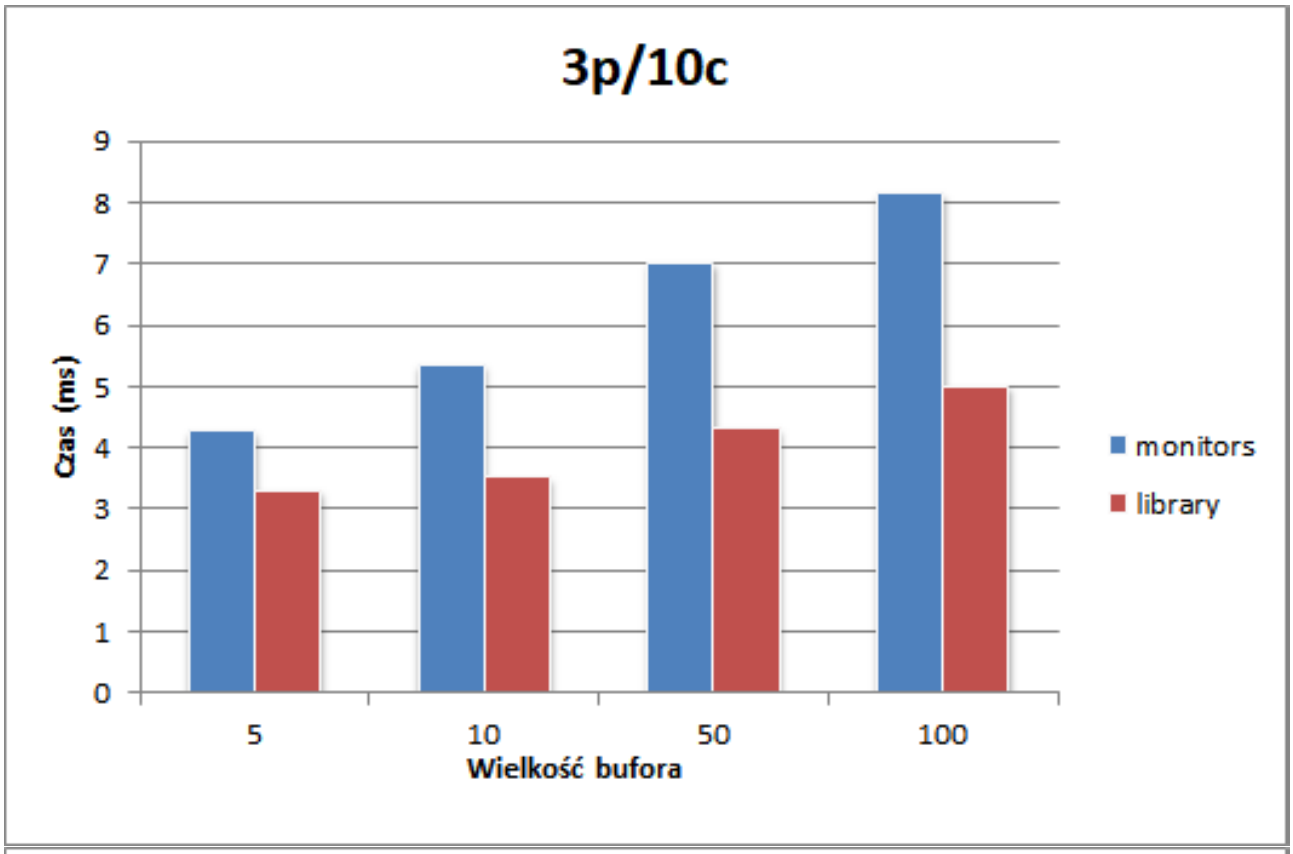
Po wykonaniu programu został stworzony plik results.csv. Wyniki prezentują się następująco:

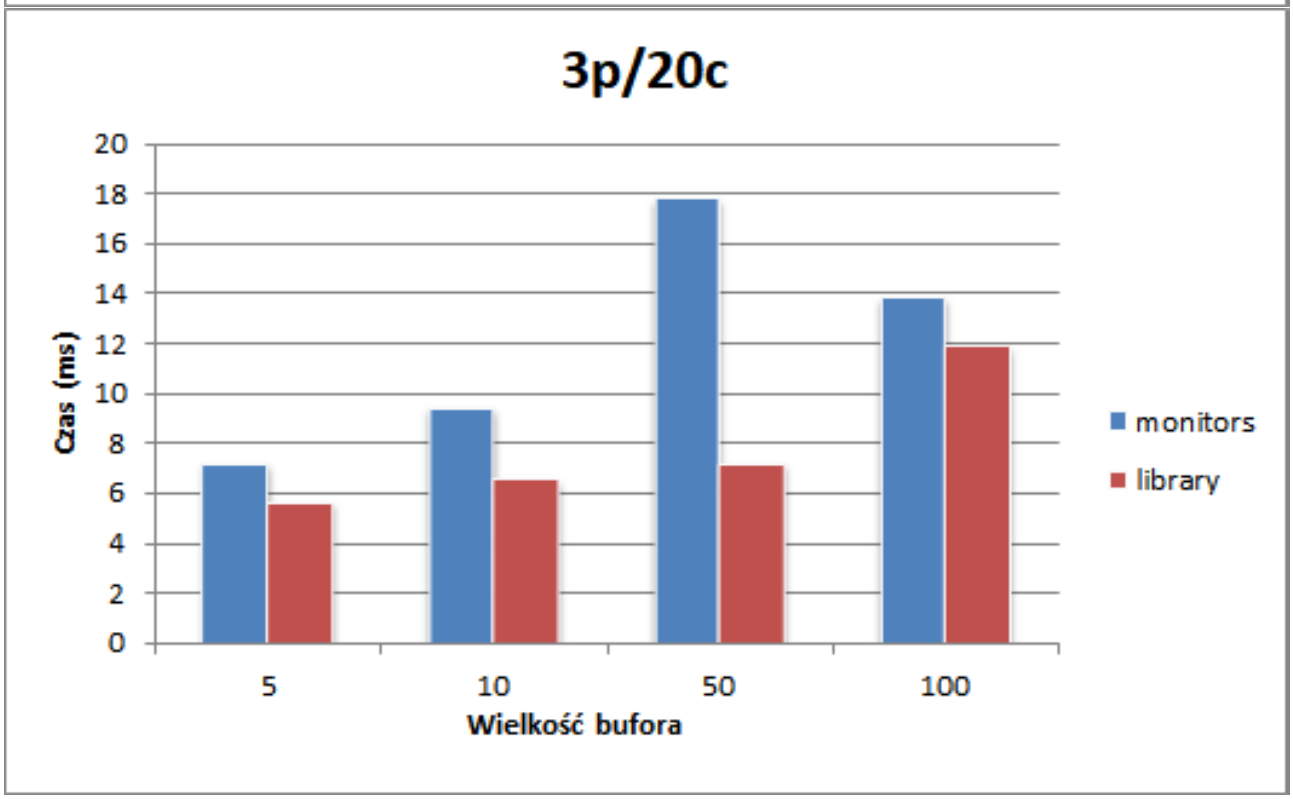
3	3	5	2.85	2.26
3	3	10	2.79	1.93
3	3	50	3.43	2.46
3	3	100	4.19	2.77
3	10	5	4.28	3.31
3	10	10	5.36	3.52
3	10	50	7.02	4.32
3	10	100	8.15	5.01
3	20	5	7.19	5.63
3	20	10	9.42	6.54
3	20	50	17.81	7.19
3	20	100	13.84	11.89
10	3	5	4.27	3.84
10	3	10	5.14	4.35
10	3	50	6.99	6.69
10	3	100	7.99	7.53
10	10	5	13.41	8.29
10	10	10	15.03	9.2
10	10	50	21.37	9.76
10	10	100	26.42	18.1
10	20	5	17.93	12.71
10	20	10	21.86	13.12

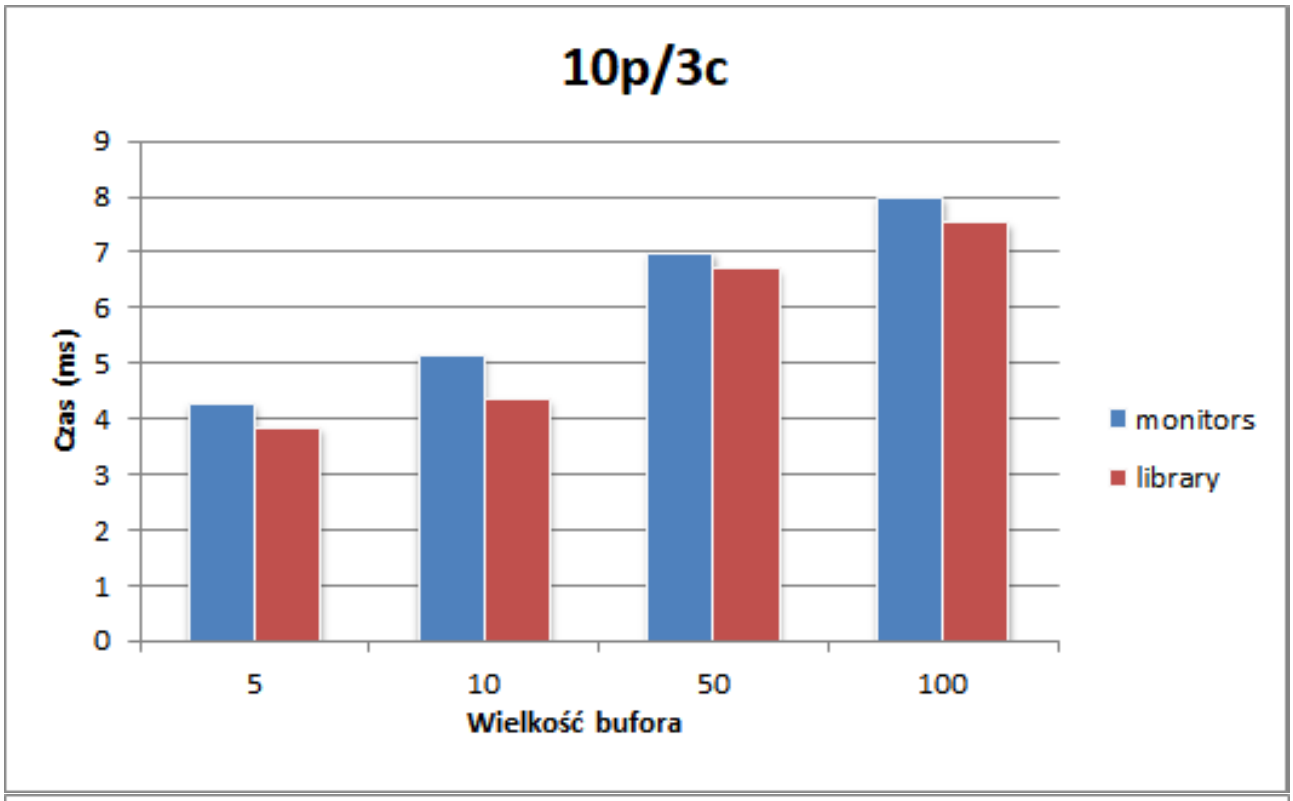
4. Wnioski

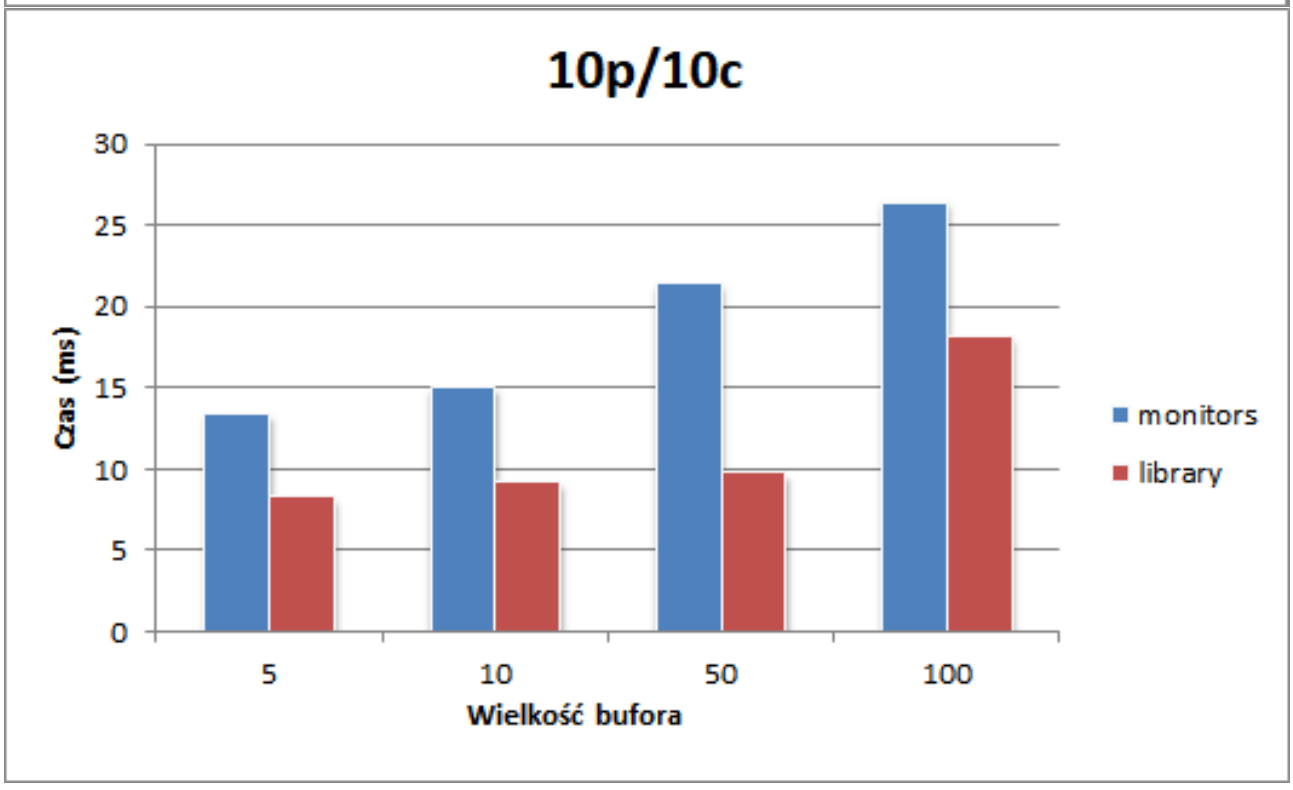
Na podstawie danych zostały wygenerowane wykresy przedstawiające czasy działania programu dla konkretnych konfiguracji. Tytuł wykresów mówi o liczbie producentów i konsumentów.

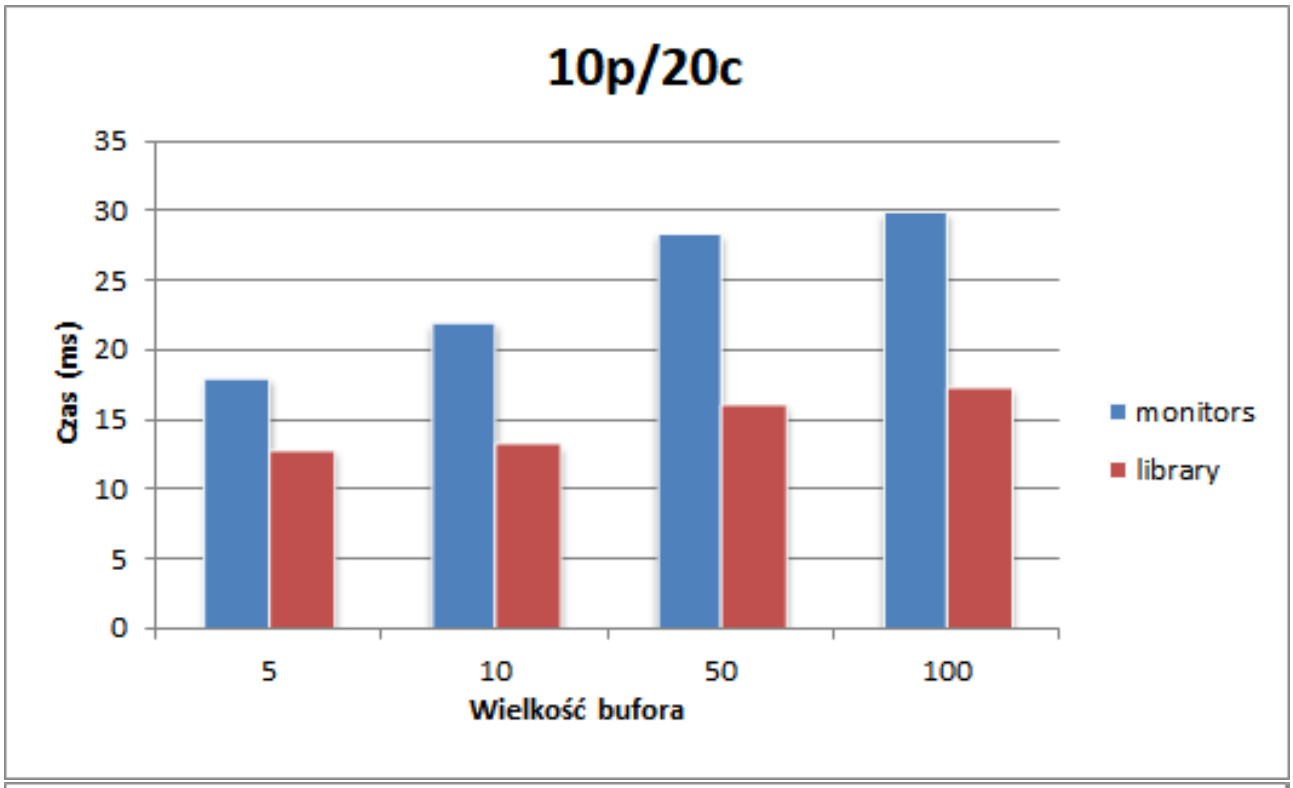


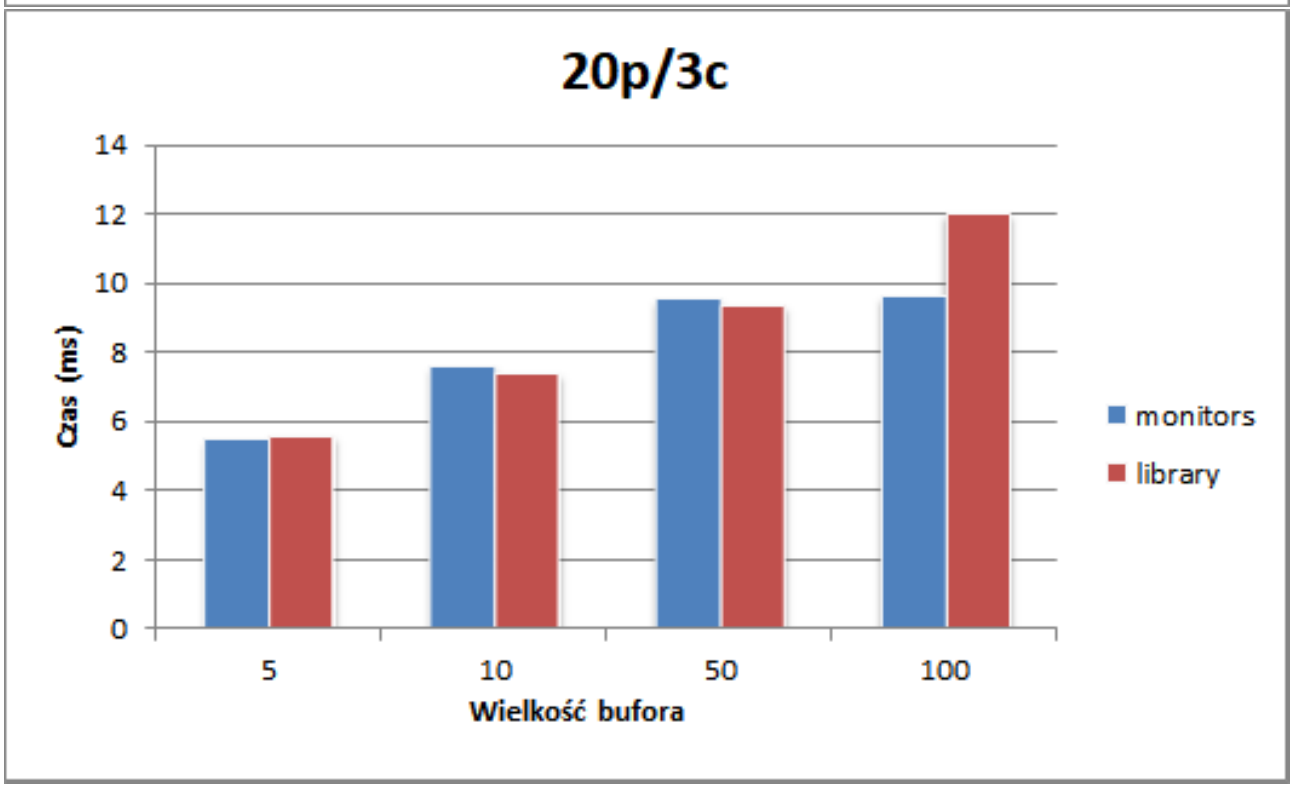

3p/10c
Czas (ms)
0
1
2
3
4
5
6
7
8
9
5
10
50
100
Wielkość bufora
monitors
library

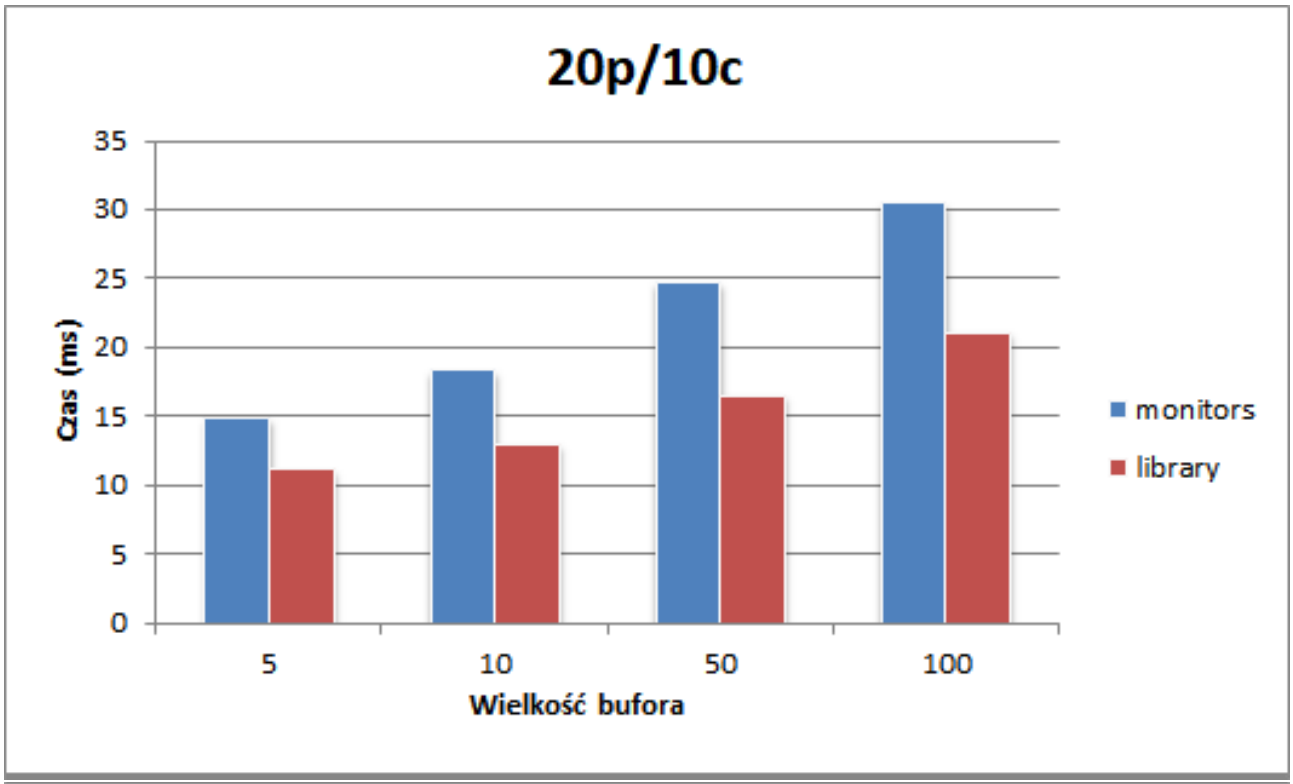

3p/20c
Czas (ms)
0
2
4
6
8
10
12
14
16
18
20
5
10
50
100
Wielkość bufora
monitors
library

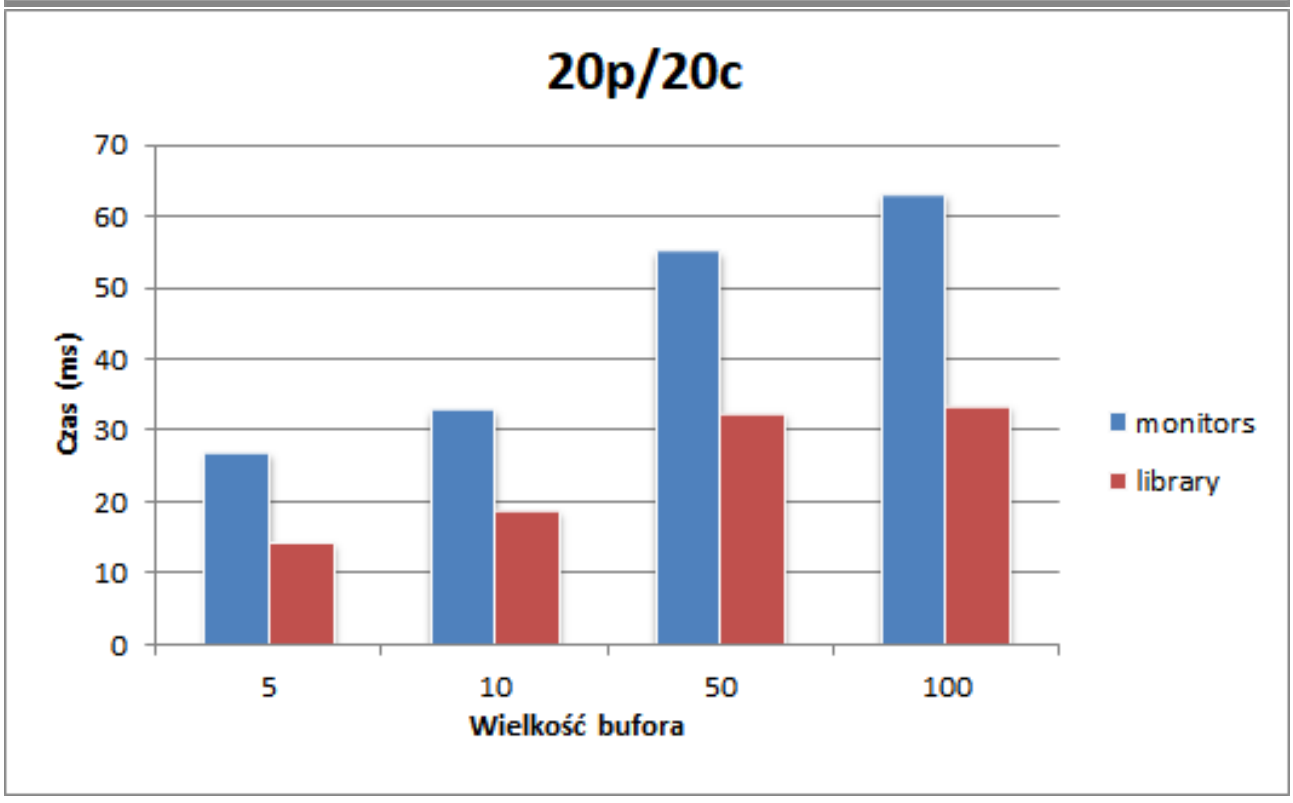

10p/3c
Czas (ms)
0
1
2
3
4
5
6
7
8
9
5
10
50
100
Wielkość bufora
monitors
library


10p/10c
Czas (ms)
0
5
10
15
20
25
30
5
10
50
100
Wielkość bufora
monitors
library


10p/20c
35
30
25
20
15
10
5
0
Czas (ms)
5
10
50
100
Wielkość bufora
monitors
library


20p/3c
14
12
10
8
6
4
2
0
Czas (ms)
5
10
50
100
Wielkość bufora
monitors
library


20p/10c
35
30
25
20
15
10
5
0
Czas (ms)
5
10
50
100
Wielkość bufora
monitors
library


20p/20c
70
60
50
40
30
20
10
0
Czas (ms)
5
10
50
100
Wielkość bufora
monitors
library

Na podstawie powyższych wykresów, można zauważyć, że w niemalże każdym przypadku, wersja problemu producentów-konsumentów zaimplementowana za pomocą monitorów jest wolniejsza od rozwiązania za pomocą Java Concurrent Library. Największe różnice w czasach widać, gdy liczba konsumentów jest większa niż liczba producentów. W przypadku gdy liczba producentów była znacznie większa niż liczba konsumentów wyniki są bardzo do siebie zbliżone.

Warto dodać, że biblioteka Java Concurrent umożliwia korzystanie z kontenerów, które mają synchronizowany dostęp (np. *BlockingQueue*), mechanizmów synchronizacji jak: *Sempahore*, *CyclicBarrier*, *CountdownLatch*, *Lock*, *Condition* oraz zmiennych atomowych (np. *AtomicInteger*). Używanie gotowych rozwiązań ogranicza także możliwość pomyłki podczas implementacji.