Sprawozdanie porównania rozwiązań problemu producentów oraz konsumentów z losowością - bez zagłodzenia –

Adam Górka, grupa nr 13.

1. Implementacja rozwiązań.

a) Rozwiązanie na 3 Lockach składa się z :

- Trzech Reentrant Locków:

```
2 usages
private final ReentrantLock producerLock = new ReentrantLock();
2 usages
private final ReentrantLock consumerLock = new ReentrantLock();
5 usages
private final ReentrantLock commonLock = new ReentrantLock();
```

producerLock – blokada dla producentów – przebywa w nim jeden producent
consumerLock – blokada dla konsumentów – przebywa w nim jeden konsument
producerLock – blokada wspólna – producenci i konsumenci rywalizują, by się w nim znaleźć

- commonCondition:

```
private final Condition condition = commonLock.newCondition();
```

Warunek, który kontroluje przepływ producentów i konsumentów

- bufora:

```
public int value = 0;
2 usages
public int full = 0;
1 usage
public int empty = 0;
```

Jest to miejsce, które jest zwiększane przez producentów i zmniejszane przez konsumentów. Może się mieścić pomiędzy empty i full. Full ustawiane jest za pomocą funkcji setFullVal

- producenta:

```
@Override
public void producent(int randomInt, int id) throws InterruptedException {

        producerLock.lock();
        commonLock.lock();
        if (!myTimer.isEnd()){
            while (value + randomInt > full) {
                condition.await();
                if (myTimer.isEnd()){
                    condition.signalAll();
                    break;
                }
            }
            if (!myTimer.isEnd()){
                myTimer.addOperation();
                value += randomInt;
            }
        }
        condition.signal();
        producerLock.unlock();
        commonLock.unlock();
}
```

Producent dostaje się do producerLocka i commonLocka. Gdy nie może obecnie wyprodukować, czeka na commonCondition. Gdy dostanie sygnał, i dostanie się do locka, produkuje (jeśli już może), wysyła sygnał do commonLocka i zwalnia producerLocka i commonLocka

- konsumenta:

```
@Override
public void consument(int randomInt, int id) throws InterruptedException {
        consumerLock.lock();
        commonLock.lock();
        if (!myTimer.isEnd()) {
            while (value - randomInt < empty) {
                condition.await();
                if (myTimer.isEnd()){
                    condition.signalAll();
                    break;
                }
            }
            if (!myTimer.isEnd()) {
                myTimer.addOperation();
                value -= randomInt;
            }
        }
        condition.signal();
        consumerLock.unlock();
        commonLock.unlock();

}
```

Konsument dostaje się do consumerLocka i commonLocka. Gdy nie może obecnie konsumować, czeka na commonCondition. Gdy dostanie sygnał, i dostanie się do locka, konsumuje (jeśli już może), wysyła sygnał do commonLocka i zwalnia consumerLocka i commonLocka

- kontrolera ilości wykonanych operacji:

```
11 usages
public MyTimer myTimer = new MyTimer();
```

Po każdej wykonanej operacji produkcji lub konsumpcji wartość w nim jest zwiększana, a gdy dojdzie do maksymalnego punktu, konsumenci i producenci przestają walczyć o wykonywanie operacji. Wszyscy zostają uwolnieni i program kończy działanie.

b) Rozwiązanie na 4 Condition składa się z :

- Jednego ReentrantLocka

```
private final ReentrantLock lock = new ReentrantLock();
```

Odpowiada on za chronienie zasobów przed dostępem innych wątków. Dopuszcza wyłącznie jeden wątek do bufora w danym momencie.

- Czterech Condition:

```
4 usages
private final Condition otherProducerCondition = lock.newCondition();
4 usages
private final Condition otherConsumerCondition = lock.newCondition();
4 usages
private final Condition firstProducerCondition = lock.newCondition();
4 usages
private final Condition firstConsumerCondition = lock.newCondition();
```

firstProducerCondition – warunek, na którym producent oczekuje na wejście do locka
firstConsumerCondition - warunek, na którym konsument oczekuje na wejście do locka
otherProducerCondition - warunek, na którym reszta producentów oczekuje, gdy firstProducerCondition jest już zajęty
otherConsumerCondition - warunek, na którym reszta konsumentów oczekuje, gdy firstConsumerCondition jest już zajęty

- producenta:

```java
@Override
public void producent(int randomInt, int id) throws InterruptedException {
    lock.lock();
    if (!myTimer.isEnd()) {
        while (isFirstElementQueueProducerFull) {
            otherProducerCondition.await();
            if (myTimer.isEnd()) {
                lock.unlock();
                return;
            }
        }
        while (value + randomInt > full) {
            isFirstElementQueueProducerFull = true;
            firstProducerCondition.await();
            if (myTimer.isEnd()) {
                lock.unlock();
                return;
            }
        }
        myTimer.addOperation();
        value += randomInt;
        isFirstElementQueueProducerFull = false;
        otherProducerCondition.signal();
        firstConsumerCondition.signal();
    }
    if (myTimer.isEnd()) {
        isFirstElementQueueProducerFull = false;
        otherProducerCondition.signalAll();
        firstConsumerCondition.signalAll();
        otherConsumerCondition.signalAll();
        firstProducerCondition.signalAll();
    }
    lock.unlock();
}
```

Po wejściu do locka, sprawdza czy ktoś już czeka na firstProducerCondition, jeśli tak, to czeka na otherProducerCondition. Jeśli nikt nie czeka na firstProducerCondition, to sprawdza, czy może produkować. Jeśli nie może, to oczekuje na firstProducerCondition, jeśli może, to produkuje i wysyła sygnał do otherProducerCondition i firstConsumerCondition.

- consumer:

```
@Override
public void consument(int randomInt, int id) throws InterruptedException {
    lock.lock();
    if (!myTimer.isEnd()) {
        while (isFirstElementQueueConsumerFull) {
            otherConsumerCondition.await();
            if (myTimer.isEnd()) {
                lock.unlock();
                return;
            }
        }
        while (value - randomInt < empty) {
            isFirstElementQueueConsumerFull = true;
            firstConsumerCondition.await();
            if (myTimer.isEnd()) {
                lock.unlock();
                return;
            }
        }
        myTimer.addOperation();
        value -= randomInt;
        isFirstElementQueueConsumerFull = false;
        otherConsumerCondition.signal();
        firstProducerCondition.signal();
    }
    if (myTimer.isEnd()) {
        isFirstElementQueueConsumerFull = false;
        otherConsumerCondition.signalAll();
        firstProducerCondition.signalAll();
        otherProducerCondition.signalAll();
        firstConsumerCondition.signalAll();
    }
    lock.unlock();
}
```

  Po wejściu do locka, sprawdza czy ktoś już czeka na firstConsumerCondition, jeśli tak, to czeka na otherConsumerCondition. Jeśli nikt nie czeka na firstConsumerCondition, to sprawdza, czy może konsumować. Jeśli nie może, to oczekuje na firstConsumerCondition, jeśli może, to produkuje i wysyła sygnał do otherConsumerCondition i firstProducerCondition.
- Bufora i kontrolera ilości wykonanych operacji, które działają na takiej samej zadzie jak w poprzednim rozwiązaniu

2. Porównanie rozwiązań.

W celu porównania rozwiązań wykonane zostały pomiary czasów rzeczywistych i czasów CPU dla obu rozwiązań. Aby więcej można było wyciągnąć z porównania, niektóre parametry zostawały zmieniane, a testy były powtarzane wiele razy, po czym zostawała z nich wyciągana średnia.
Parametry używane do testów:

- Ilość wątków producentów i konsumentów – 5 lub 20
- Ilość powtórzeń każdego testu – 30
- Rozmiar bufora – 800
- Maksymalny rozmiar bufora – 50, 100, 150, 200, 250, 300, 350, 400
- Liczba wykonanych operacji – 10000, 100000, 1000000, 10000000

Najpierw ustalano wartości odpowiednich parametrów. Następnie tworzono wątki producentów i konsumentów. W kolejnym kroku wątki pracowały. Po zatrzymaniu wątków, zliczano czasy do sumy, a po osiągnięciu maksymalnej ilości testów, brano z nich średnią arytmetyczną i wpisywano do pliku dane.csv. Jednostka dla czasów to milisekunda.

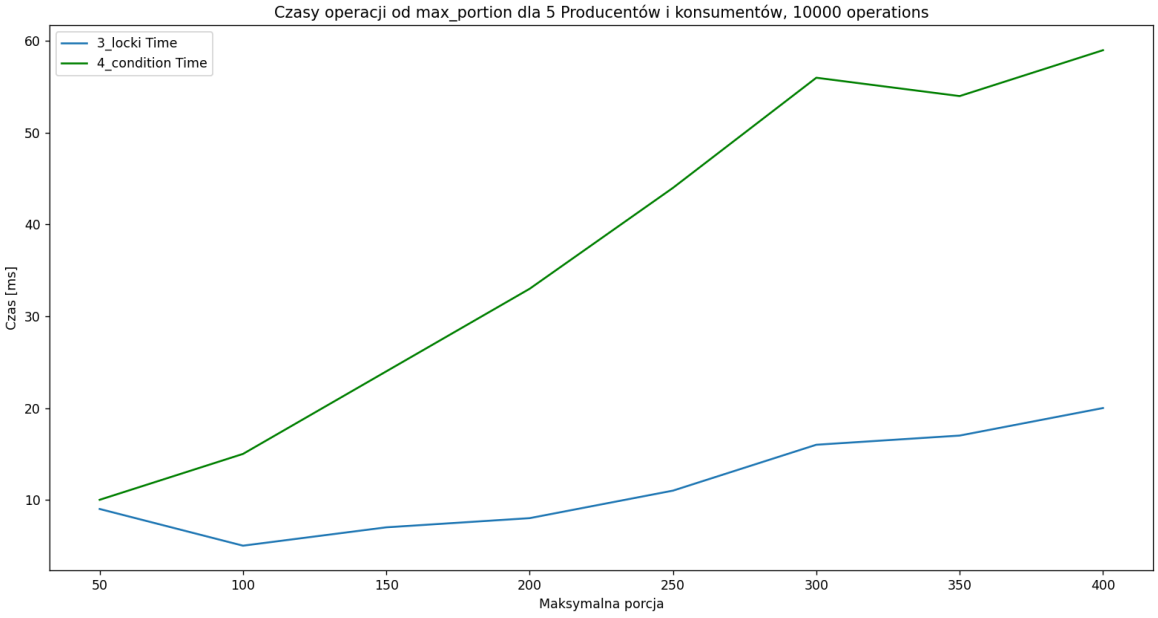
3. Specyfikacja urządzenia.

Pomiary wykonano na procesorze AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx, 2100 MHz, Rdzenie: 4, Procesory logiczne: 8. Nazwa systemu operacyjnego - Microsoft Windows 10 Home. Testy uruchamiane były w środowisku Intellij IDEA z Java 17

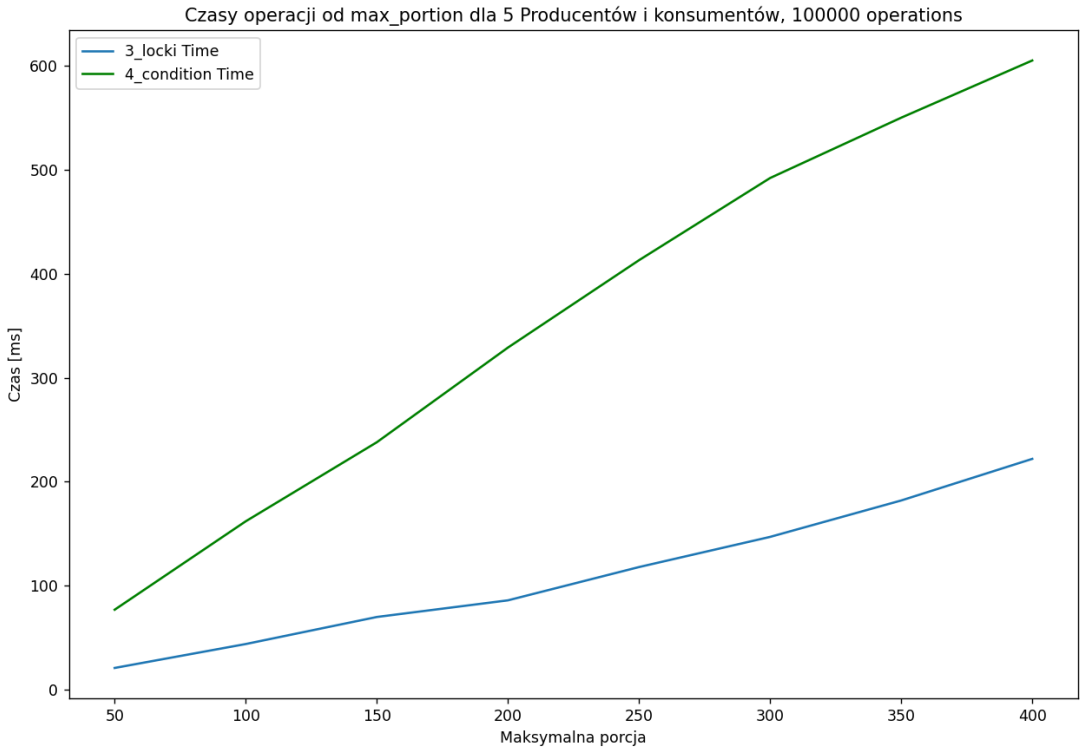
4. Testy.

Poniżej znajdują się wykresy otrzymane za pomocą programu w języku Python (MakePlots.py). Dane pochodzą z pliku dane.csv

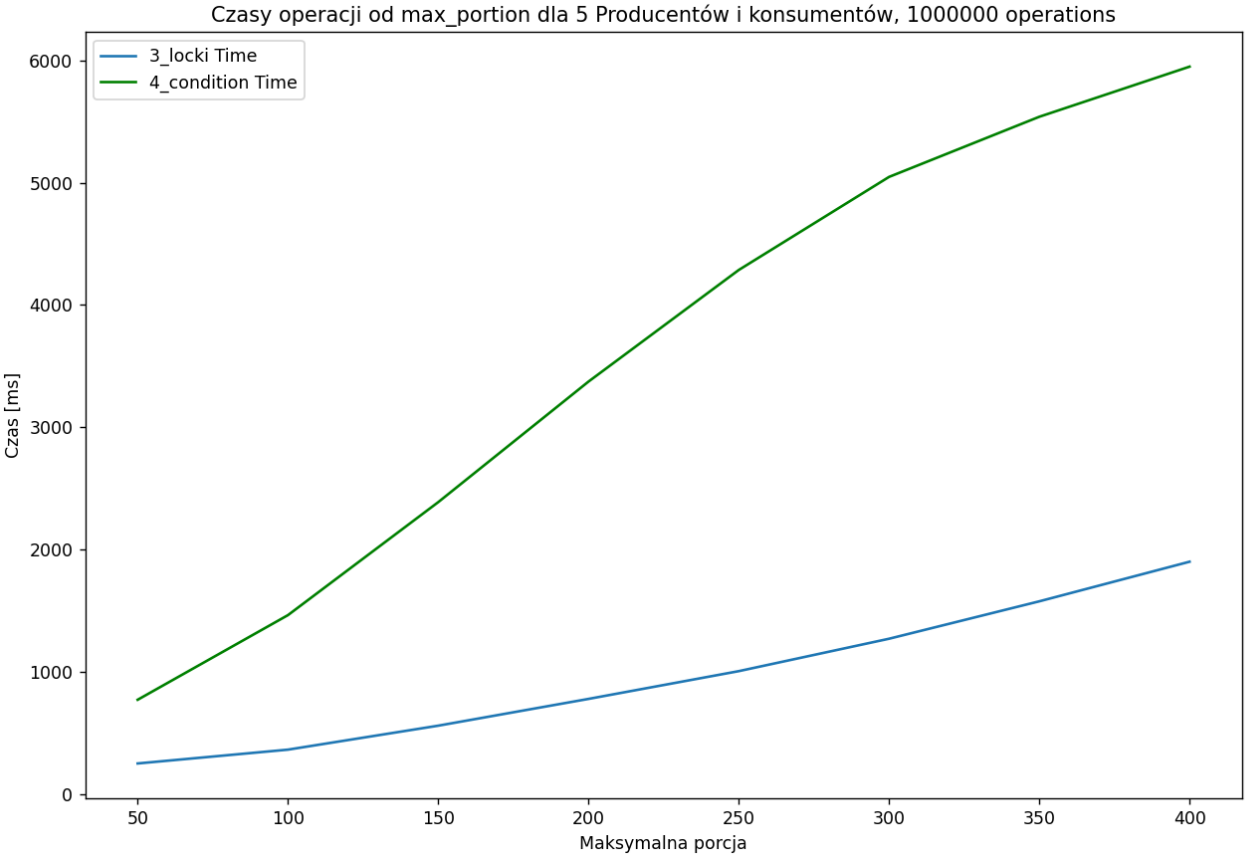
Czasy dla bufora – 800, 5 producentów i konsumentów, oraz 10000 operacji



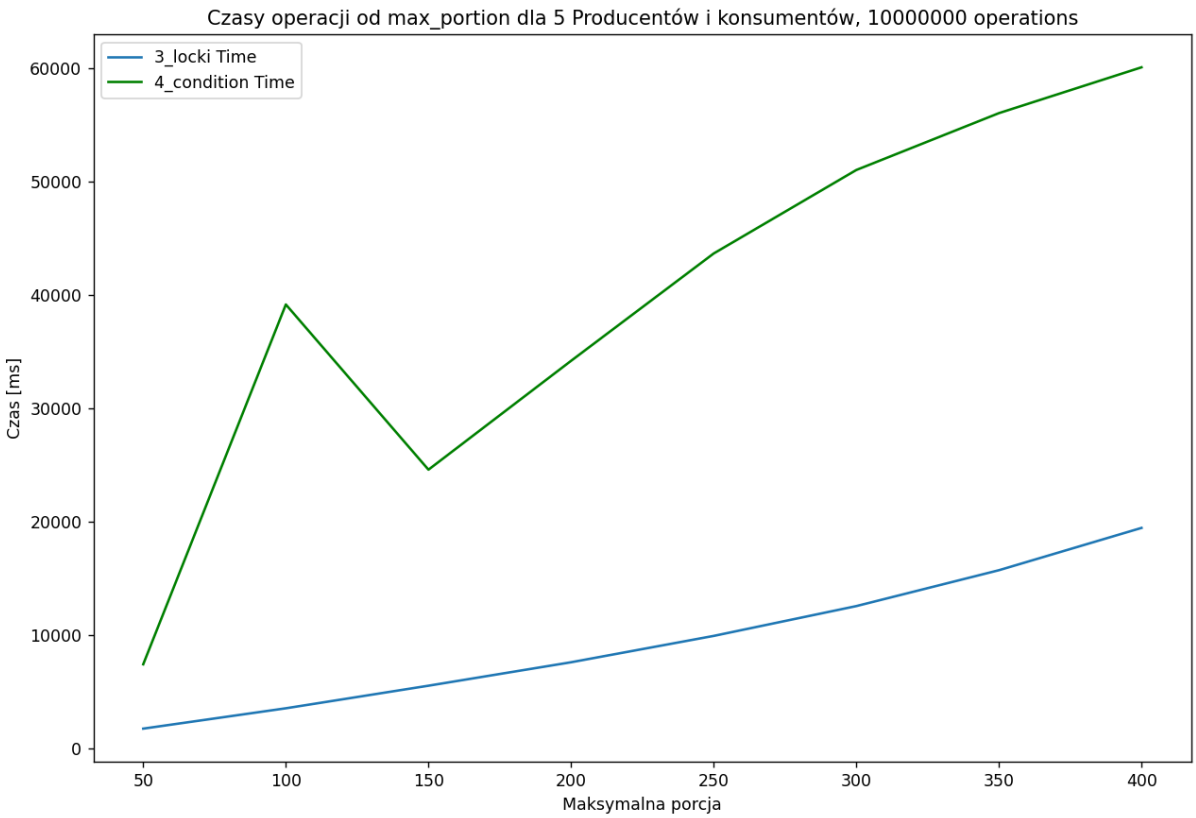
Czasy dla bufora – 800, 5 producentów i konsumentów, oraz 100000 operacji



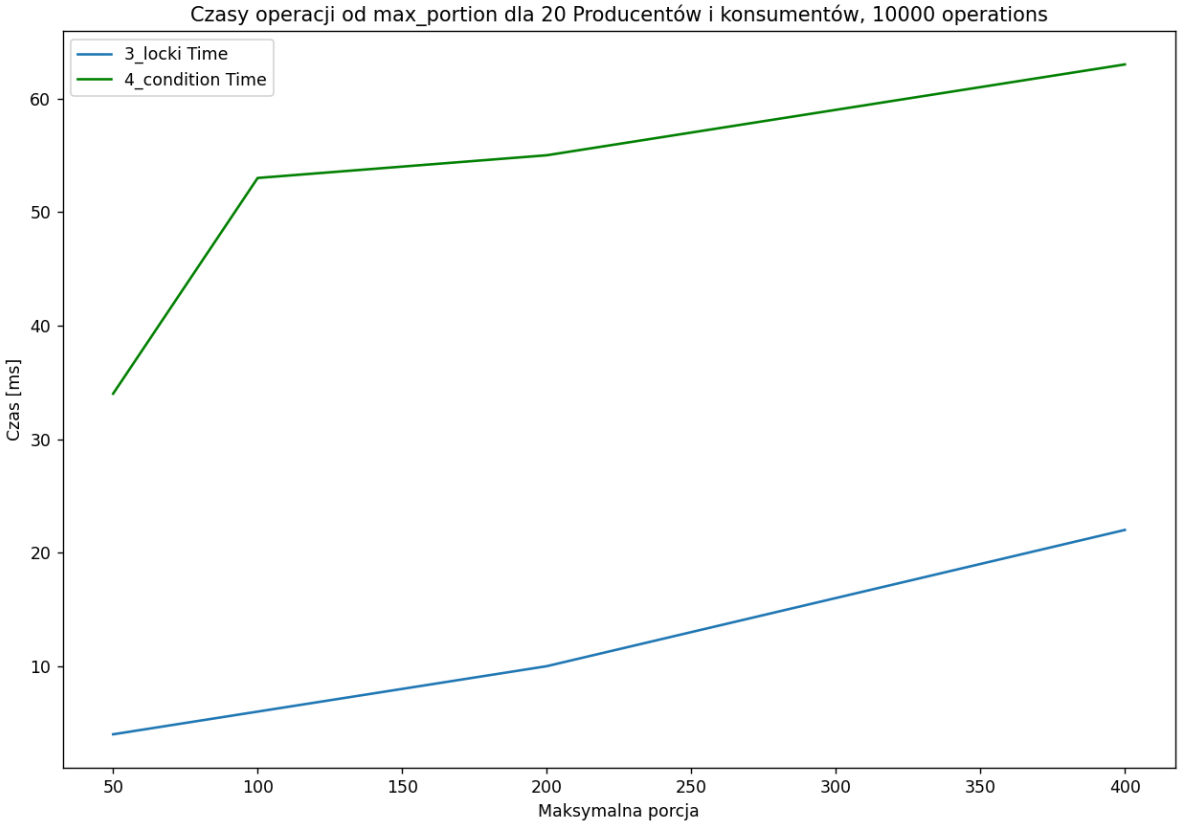
Czasy dla bufora – 800, 5 producentów i konsumentów, oraz 1000000 operacji



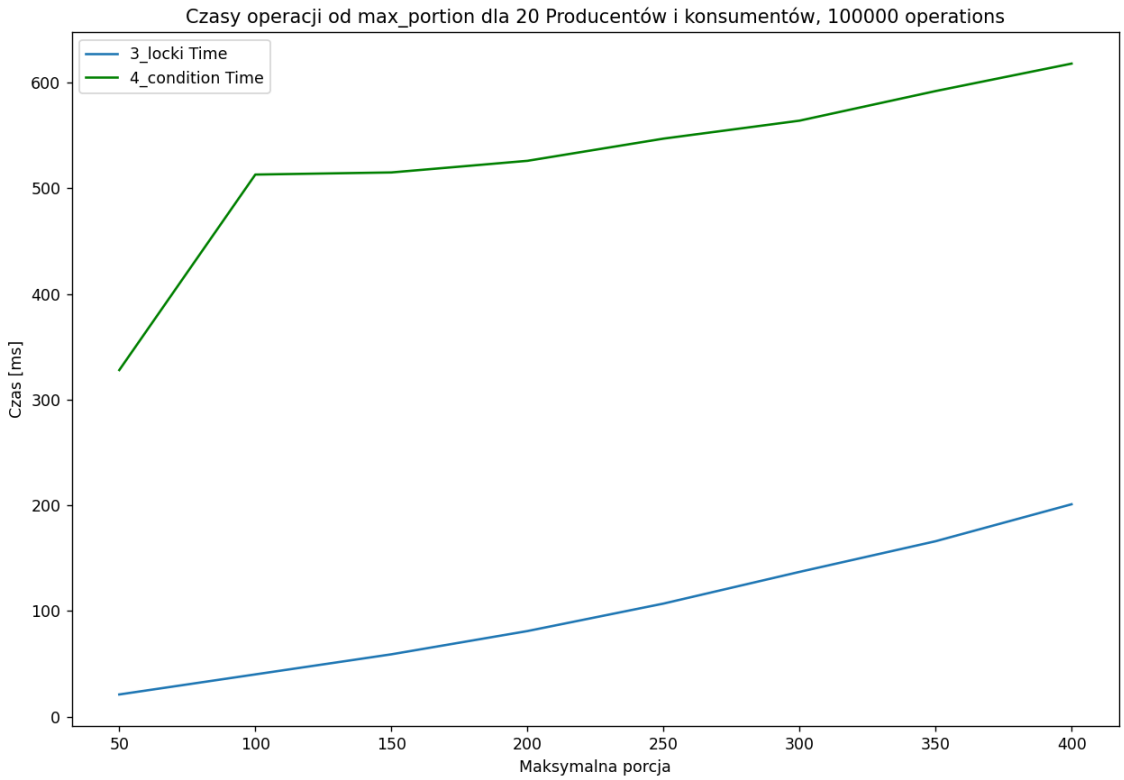
Czasy dla bufora – 800, 5 producentów i konsumentów, oraz 10000000 operacji



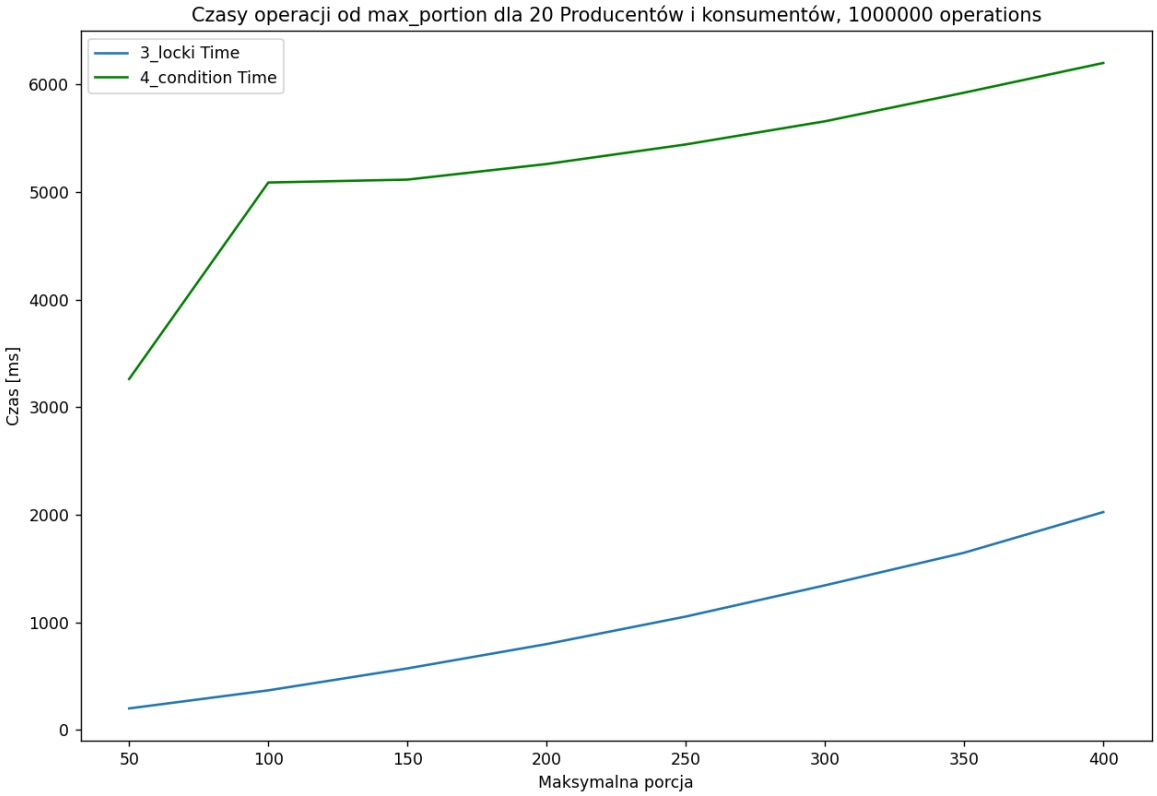
Czasy dla bufora – 800, 20 producentów i konsumentów, oraz 10000 operacji



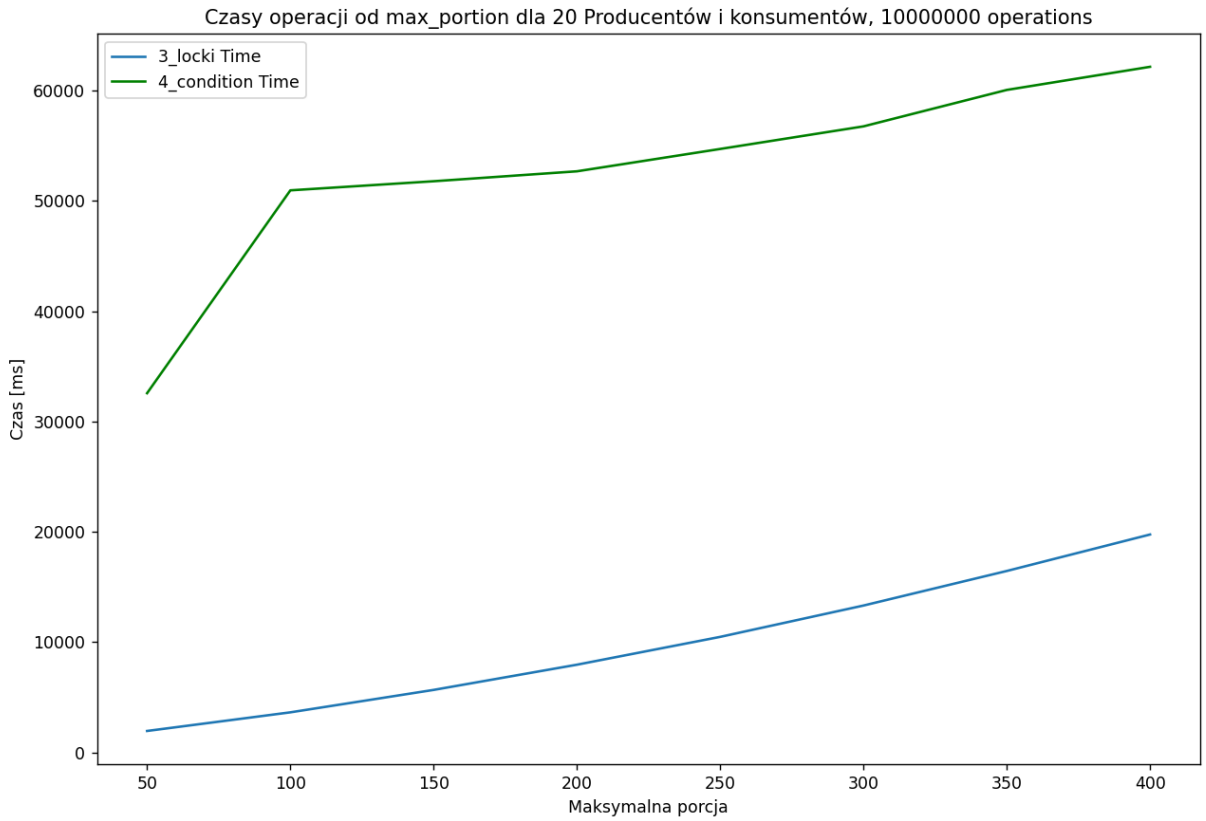
Czasy dla bufora – 800, 20 producentów i konsumentów, oraz 100000 operacji



Czasy dla bufora – 800, 20 producentów i konsumentów, oraz 1000000 operacji



Czasy dla bufora – 800, 20 producentów i konsumentów, oraz 10000000 operacji



5. Wnioski.

- Łatwo możemy zauważyć na podstawie powyższych wyników, że choć oba rozwiązania są poprawne i nie powodują zakleszczeń, to rozwiązanie na 3 Lockach działa znacznie szybciej dla każdych parametrów.
- Czasy dla rozwiązania na 4 Condition zazwyczaj rosną szybciej.
- Kolejnym spostrzeżeniem jest, że każdy wykres jest rosnący. Oznacza to, że im większą mamy możliwą porcję do losowania, tym czasy wykonania danej ilości operacji są większe.
- Przy ustawieniu 20 producentów i konsumentów szczególnie zauważalna jest różnica między pomiarami czasów dla maksymalnego bufora 50, a maksymalnego bufora 100. Może to być spowodowane tym, że bufor w podanym przypadku zwiększa się aż dwukrotnie.
- Dla konkretnej ilości producentów i konsumentów, wykresy mają podobny kształt, niezależnie o ilości wykonywanych operacji.