Teoria Współbieżności Lab6 - sprawozdanie

Autor: Michał Flak

Blokowanie drobnoziarniste

Zamek (lock) jest przydatny wtedy, gdy operacje zamykania/otwierania nie mogą być umieszczone w jednej metodzie lub bloku synchronized. Przykładem jest zakładanie blokady (lock) na elementy struktury danych, np. listy. Podczas przeglądania listy stosujemy następujący algorytm:

1. zamknij zamek na pierwszym elemencie listy
2. zamknij zamek na drugim elemencie
3. otwórz zamek na pierwszym elemencie
4. zamknij zamek na trzecim elemencie
5. otwórz zamek na drugim elemencie
6. powtarzaj dla kolejnych elementów

Dzięki temu unikamy konieczności blokowania całej listy i wiele wątków może równocześnie przeglądać i modyfikować różne jej fragmenty.

Ćwiczenie

1. Proszę zaimplementować listę, w której każdy węzeł składa się z wartości typu Object, referencji do następnego węzła oraz zamka (lock). Lista nie może przechowywać wartości null.
2. Proszę zastosować metodę drobnoziarnistego blokowania do następujących metod listy:

```
boolean contains(Object o); //czy lista zawiera element o
boolean remove(Object o); //usuwa pierwsze wystąpienie elementu o
boolean add(Object o); //dodaje element o na koncu listy
```

Proszę porównać wydajność tego rozwiązania w stosunku do listy z jednym zamkiem blokującym dostęp do całości. Należy założyć, że koszt czasowy operacji na elemencie listy (porównanie, wstawianie obiektu) może być duży - proszę wykonać pomiary dla różnych wartości tego kosztu. Proszę sprawdzić jaka będzie wydajność tego rozwiązania dla różnej ilości procesów uzyskujących dostęp do zasobów. Ilość procesów od 10 do 100.

Zaproponowany test

Utworzono metodę `ListPerformanceTester.Test(int threadCount, int elemCount, IList list, int cost)`, która na podanej liście wykonuje po sobie trzy następujące testy, mierząc czas każdego z nich:

- `randomAdd(threadCount,elemCount,list)` Wstawia do listy `elemCount` losowo wybranych elementów z przedziału 0 - `elemCount`.
- `randomContains(threadCount,elemCount,list)` Sprawdza `elemCount` razy czy lista zawiera losowo wybraną wartość z zakresu 0 - `elemCount`.
- `randomRemove(threadCount,elemCount,list)` Próbuje usuwać `elemCount` razy losowe elementy z zakresu 0 - `elemCount`.

Każda z tych metod przed podjęciem działania sprawdza, czy argument `threadId` zgadza się z wartością `currentAllowedUser` odpowiadającą polu. Jeśli nie - czeka, aż inny wątek o mniejszym ID zakończy pracę nad polem i inkrementuje `currentAllowedUser`.

Kod klasy:

```
public class ListPerformanceTester {
    public void Test(int threadCount, int elemCount, IList list, int cost){
        var randomAddTime = randomAdd(threadCount,elemCount,list);
        var randomContainsTime = randomContains(threadCount,elemCount,list);
```

```
        var randomRemoveTime = randomRemove(threadCount,elemCount,list);
        System.out.println(
                "list: "+
                        list.getClass().getSimpleName() +
                        ", elements: "+
                        elemCount+
                        ", threadCount: "+
                        threadCount +
                        ", cost[ms]: "+
                        cost +
                        ", randomAdd[ms]: " +
                        randomAddTime +
                        ", randomContains[ms]: " +
                        randomContainsTime +
                        ", randomRemove[ms]: " +
                        randomRemoveTime
        );
    }

    private long randomAdd(int threadCount, int elemCount, IList list){
        Random random = new Random();
        Thread[] threads = new Thread[threadCount];

        var beforeAdd = System.currentTimeMillis();
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(() -> {
                for(int j = 0; j < elemCount/threadCount; j++) {
                    list.add(random.nextInt(elemCount));
                }
            });
        }
        for(var t : threads) t.start();
        for(var t : threads) {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        var afterAdd = System.currentTimeMillis();
        return afterAdd-beforeAdd;
    }
    private long randomContains(int threadCount, int elemCount, IList list){
        Random random = new Random();
        Thread[] threads = new Thread[threadCount];

        var before = System.currentTimeMillis();
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(() -> {
                for(int j = 0; j < elemCount/threadCount; j++) {
                    list.contains(random.nextInt(elemCount));
                }
```

```
            });
        }
        for(var t : threads) t.start();
        for(var t : threads) {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        var after = System.currentTimeMillis();
        return after-before;
    }
    private long randomRemove(int threadCount, int elemCount, IList list){
        Random random = new Random();
        Thread[] threads = new Thread[threadCount];

        var before = System.currentTimeMillis();
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(() -> {
                for(int j = 0; j < elemCount/threadCount; j++) {
                    list.remove(random.nextInt(elemCount));
                }
            });
        }
        for(var t : threads) t.start();
        for(var t : threads) {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        var after = System.currentTimeMillis();
        return after-before;
    }
}
```

Main

Tworzymy 200-elementowe listy obu typów i wykonujemy wyżej opisany test dla kosztów: [0,1,2,3,4,5] oraz ilości wątków [10,20,30,40,50,60,70,80,90,100].

```
public class Main {

    public static void main(String[] args) {

        int elemCount = 200;
        var tester = new ListPerformanceTester();
        for(int cost = 0; cost <= 5; cost++) {
            for (int threads = 10; threads <= 100; threads += 10) {
                tester.Test(threads, elemCount, new GranularLockingList(cost), cost);
```

```
                tester.Test(threads, elemCount, new WholeLockingList(cost), cost);
            }
        }
    }
}
```

Listy

Zdefiniowano interfejs listy:

```
public interface IList {
    public boolean contains(Object o);
    public boolean remove(Object o);
    public boolean add(Object o);
}
```

Dodatkowo każdy z konstruktorów implementacji przyjmuje w argumencie `int singleOperationTimeCost`, który aplikowany jest w `contains()` oraz `remove()` jako koszt porównania. W przypadku `add()` nie musimy dokonywać operacji na wartościach elementów listy, dlatego zdecydowałem się nie aplikować kosztu w tym przypadku.

Prostszym przypadkiem jest `WholeLockingList` w której blokujemy całą listę przy wykonaniu dowolnej z powyższych operacji oznaczając te metody jako `synchronized`.

```
public class WholeLockingList implements IList {

    private Node first;
    private int singleOperationTimeCost;

    public WholeLockingList(int singleOperationTimeCost) {
        this.singleOperationTimeCost = singleOperationTimeCost;
    }

    private void applyTimeCost(){
        try {
            Thread.sleep(singleOperationTimeCost);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public synchronized boolean contains(Object o) {
        if (first == null) return false;

        //Search the list
        Node current = first;
        do {
            applyTimeCost();
            if (current.getValue() == o) {
                return true;
            }
            current = current.getNext();
        } while (current != null);
        return false;
```

```
    }

    @Override
    public synchronized boolean remove(Object o) {
        if(o == null) {
            return false;
        }

        //List empty? return
        if(first == null) {
            return false;
        }

        //First element should be removed?
        if(first.getValue() == o) {
            applyTimeCost();
            this.first = first.next;
            return true;
        }

        //search the rest
        Node previous = first;
        Node current = first.getNext();
        while(current != null) {
            applyTimeCost();
            if(current.getValue() == o) {
                previous.setNext(current.getNext());
                return true;
            }
            previous = current;
            current = current.getNext();
        }
        return false;
    }

    @Override
    public synchronized boolean add(Object o) {
        if(o == null) {
            return false;
        }

        Node newNode = new Node(o);

        //List empty? Insert as first element
        if(first == null) {
            first = newNode;
            return true;
        }

        //Search for the tail
        Node current = first;
        while(current.getNext() != null){
            current = current.getNext();
```

```
        }

        //We have the tail of the list, now add new element
        applyTimeCost();
        current.setNext(newNode);
        return true;
    }

    private class Node {
        private Object value;
        private Node next;

        public Node(Object value) {
            this.value = value;
        }

        public Object getValue(){
            return value;
        }

        public void setNext(Node nextNode){
            this.next = nextNode;
        }

        public Node getNext() {
            return next;
        }
    }
}
```

Drugą listą jest `GranularLockingList`, w której blokowane są naraz jedynie dwa następujące po sobie elementy przez jeden wątek (zgodnie z zadanym algorytmem). Każdy z elementów listy posiada własny lock.

```
public class GranularLockingList implements IList {
    private Node first;
    private int singleOperationTimeCost;
    private ReentrantLock firstElementLock;

    public GranularLockingList(int singleOperationTimeCost) {
        this.singleOperationTimeCost = singleOperationTimeCost;
        firstElementLock = new ReentrantLock();
    }

    private void applyTimeCost(){
        try {
            Thread.sleep(singleOperationTimeCost);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public boolean contains(Object o) {
        //handle case of empty list
        if(first == null) return false;
```

```
        //Search the list
        Node current = first;
        current.lock();
        do {
            applyTimeCost();
            if(current.getValue() == o) {
                current.unlock();
                return true;
            }
            if(current.getNext() != null) current.getNext().lock();
            Node temp = current;
            current = current.getNext();
            temp.unlock();
        } while(current != null);

        return false;
    }
```

W tym miejscu potrzebny jest komentarz. Napotkałem pewien problem blokując wyłącznie elementy listy, ponieważ w czasie, kiedy jeden z wątków operował na pierwszym elemencie, inny próbował usunąć go z listy, tym samym zmieniając jej głowę. Żeby temu zaradzić dodałem blok `synchronized(this)` w przypadku operacji na pierwszym elemencie listy.

```
    public boolean remove(Object o) {
        Node previous = null, current = null;
        if(o == null) return false;

        //lock access to first element to prevent other threads
        //from removing the head
        synchronized (this) {

            //List empty? return
            if (first == null) {
                //System.out.println("List empty: " + o);
                return false;
            }

            //First element should be removed?
            first.lock();
            if (first.getValue() == o) {
                //System.out.println("Removing first element: " + o);
                applyTimeCost();
                var temp = first;
                this.first = temp.getNext();
                temp.unlock();
                return true;
            }

            //list 1-element long?
            if (first.getNext() == null) {
                first.unlock();
                //System.out.println("List is 1-element long: " + o);
                return false;
```

```
            }

            //2 locks acquired, go with the rest
            previous = first;
            current = first.getNext();

            //end lock on first element
        }

        while(current != null) {
            current.lock();
            applyTimeCost();
            if(current.getValue() == o) {
                //System.out.println("Removing nth element: " + o);
                previous.setNext(current.getNext());
                previous.unlock();
                current.unlock();
                return true;
            }
            previous.unlock();
            previous = current;
            current = current.getNext();
        }
        previous.unlock();

        return false;
    }

    public boolean add(Object o) {
        if(o == null) return false;

        Node newNode = new Node(o);

        //List empty? Insert as first element
        if(first == null) {
            first = newNode;
            return true;
        }

        //Search for the tail
        Node current = first;
        current.lock();
        while(current.getNext() != null){
            if(current.getNext() != null) current.getNext().lock();
            Node temp = current;
            current = current.getNext();
            temp.unlock();
        }
        //We have the tail of the list, now add new element
        applyTimeCost();
        current.setNext(newNode);
        //Release lock on the old tail
        current.unlock();
```

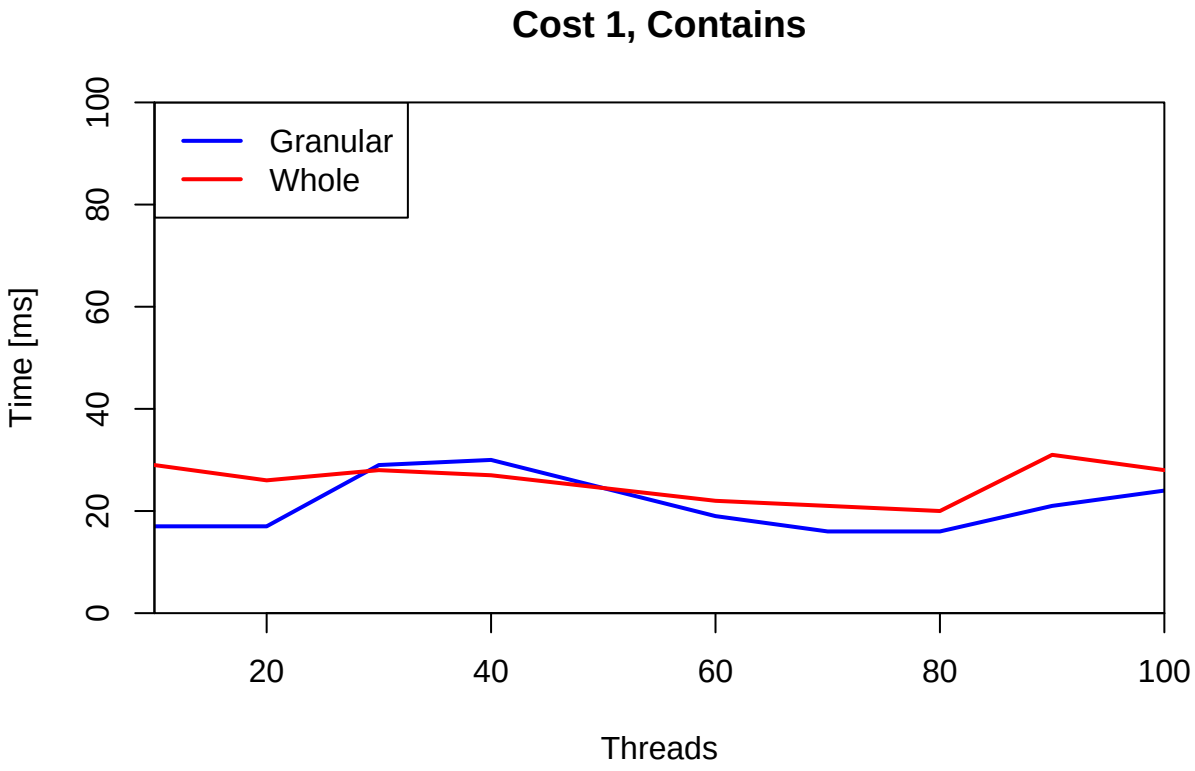
```
        return true;
    }

    private class Node {
        private Object value;
        private Node next;
        private ReentrantLock lock;

        public Node(Object value) {
            this.value = value;
            this.lock = new ReentrantLock();
        }

        public void lock(){
            this.lock.lock();
        }

        public void unlock(){
            this.lock.unlock();
        }

        public Object getValue(){
            return value;
        }

        public void setNext(Node nextNode){
            this.next = nextNode;
        }

        public Node getNext(){
            return next;
        }
    }
}
```

Wyniki

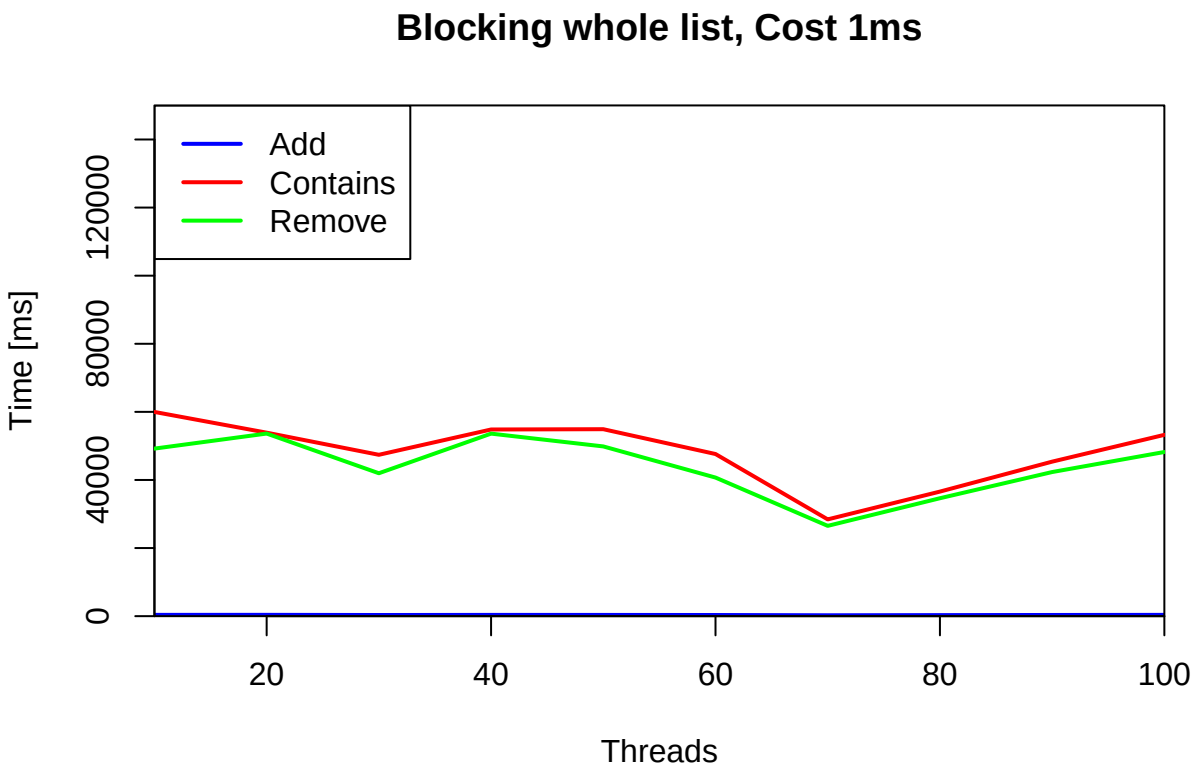
Program generuje wyniki w następującym formacie:

```
list: GranularLockingList, elements: 200, threadCount: 10, cost[ms]: 0, randomAdd[ms]: 14, randomContai
list: WholeLockingList, elements: 200, threadCount: 10, cost[ms]: 0, randomAdd[ms]: 5, randomContains[m
list: GranularLockingList, elements: 200, threadCount: 20, cost[ms]: 0, randomAdd[ms]: 7, randomContain
list: WholeLockingList, elements: 200, threadCount: 20, cost[ms]: 0, randomAdd[ms]: 3, randomContains[m
...
```

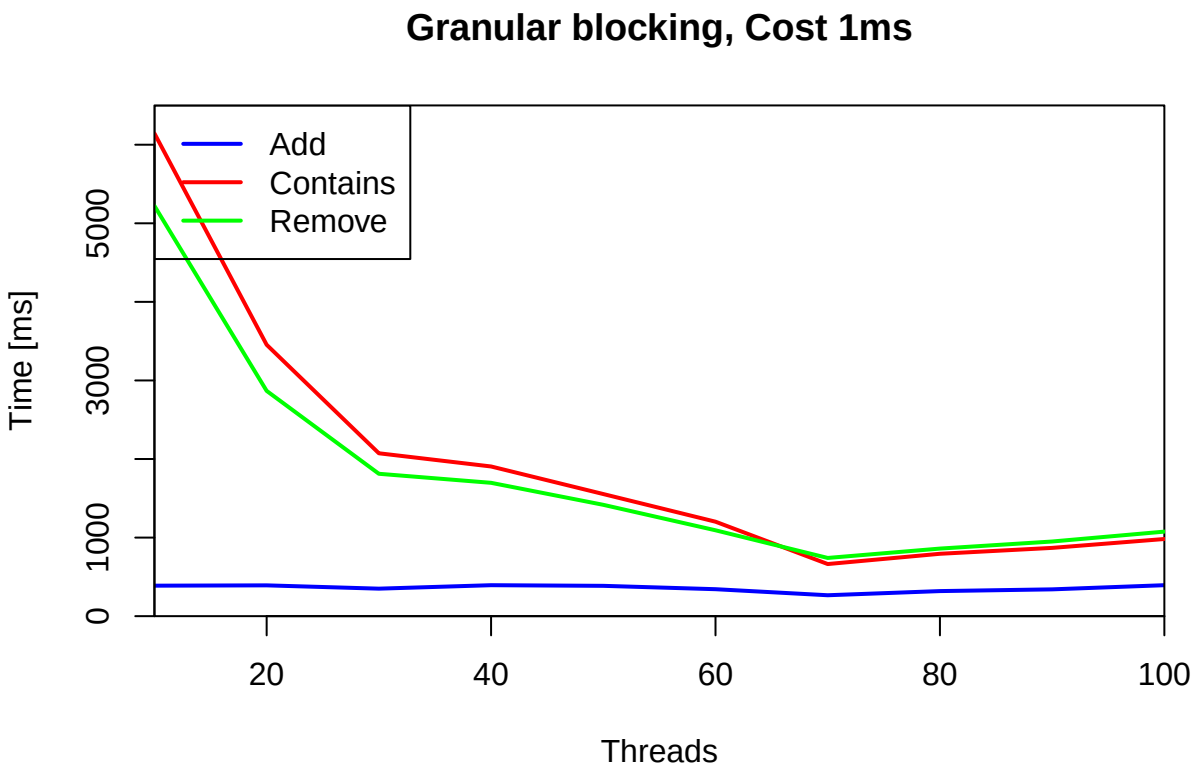
Przedstawię je na wykresach, gdzie osią Y będzie czas działania, a osią X ilość wątków.



Sprawdźmy różnice w czasie wykonania poszczególnych operacji (add, contains, remove) dla blokowania całej listy:

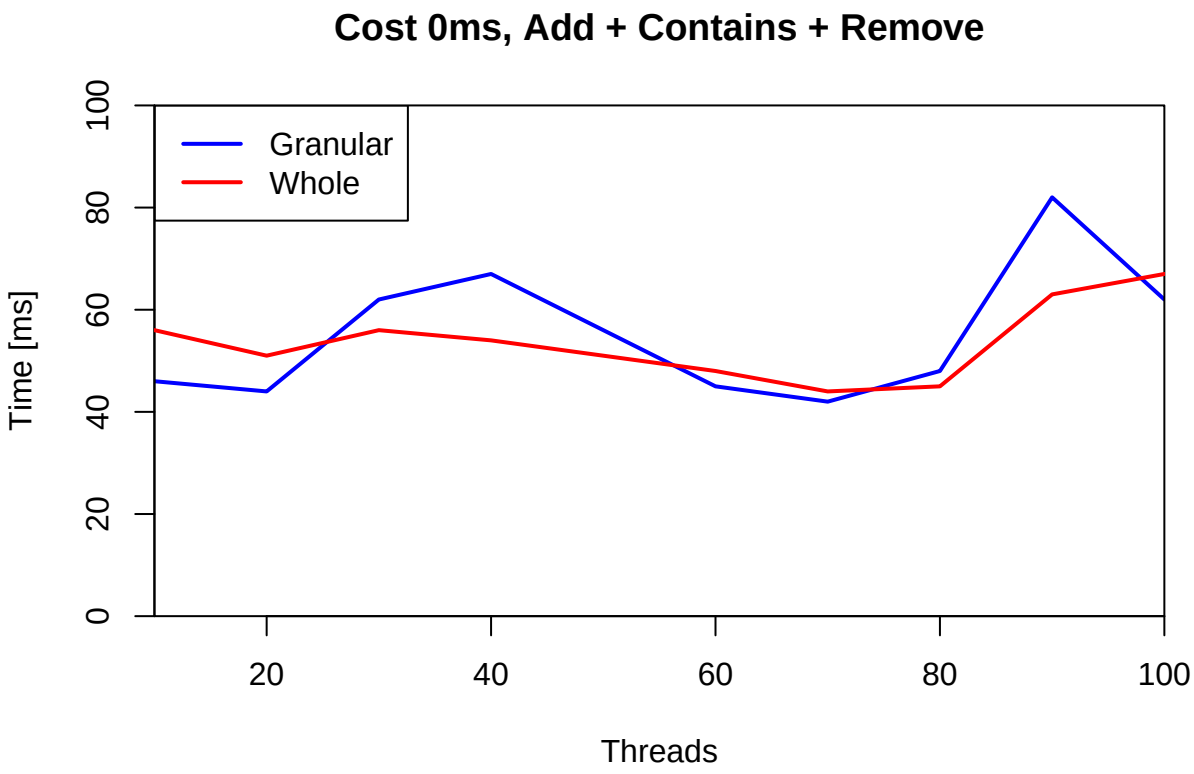


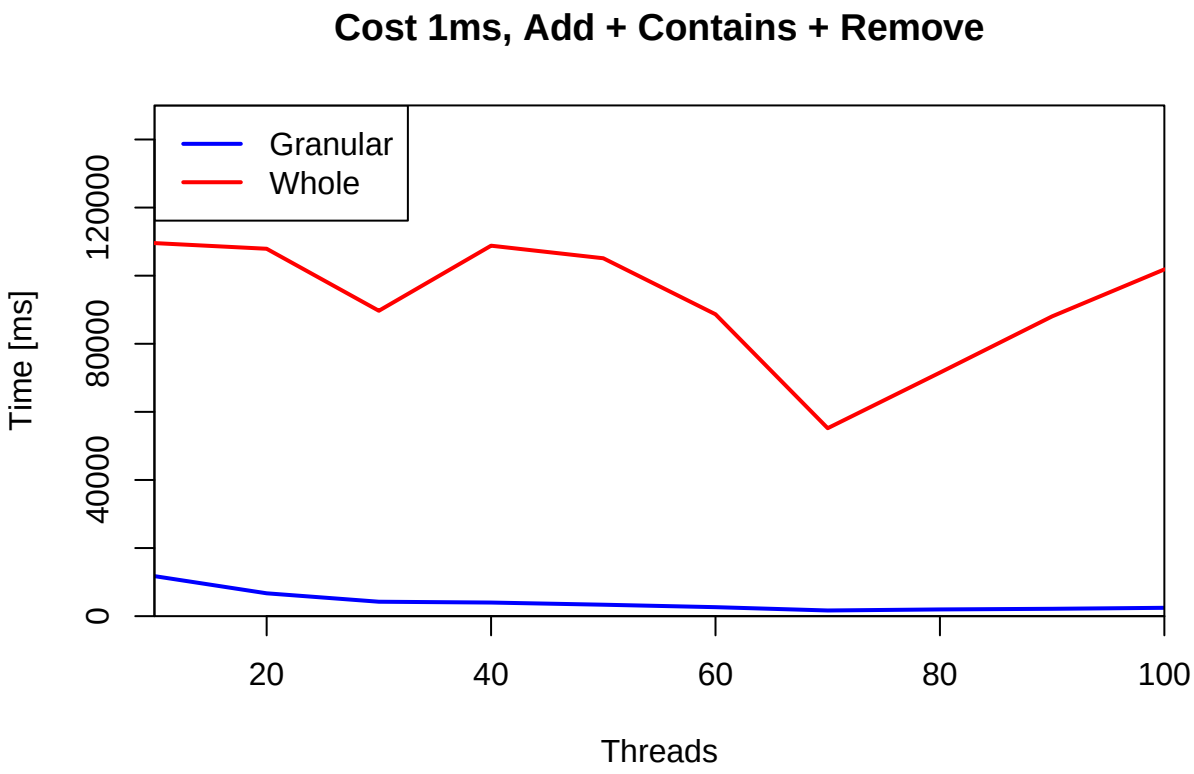
Oraz dla blokowania drobnoziarnistego:

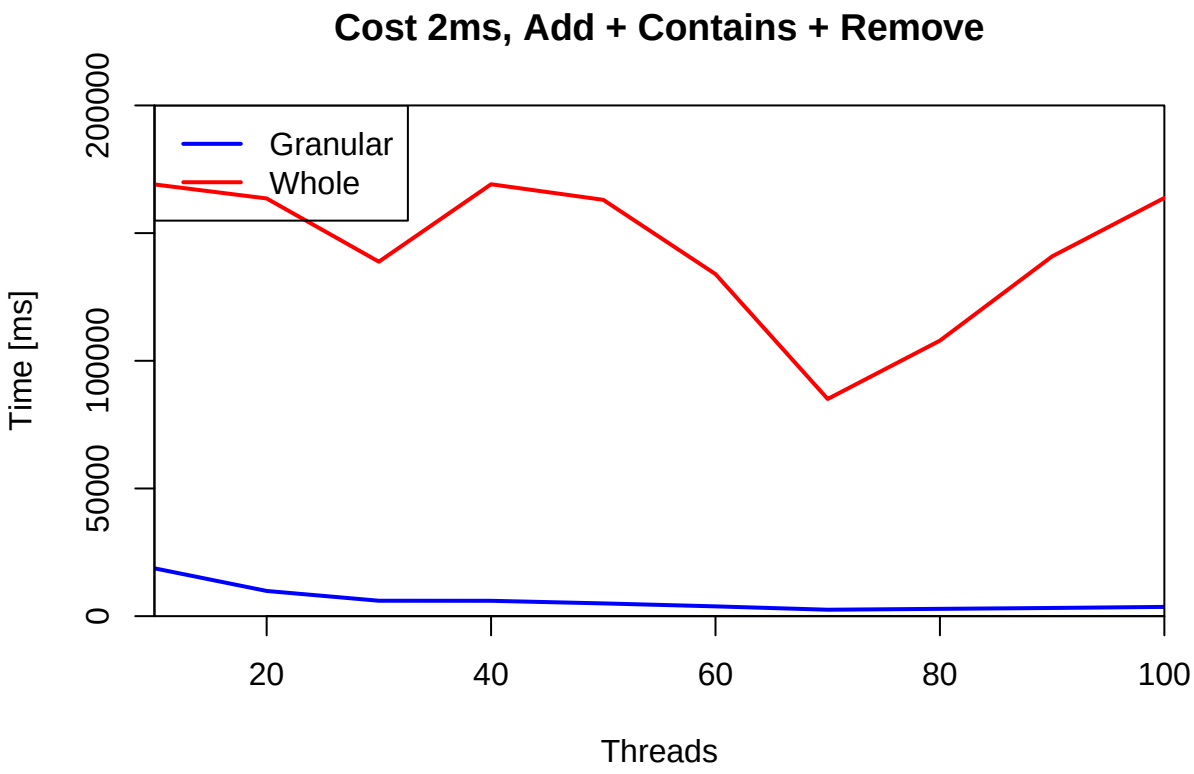


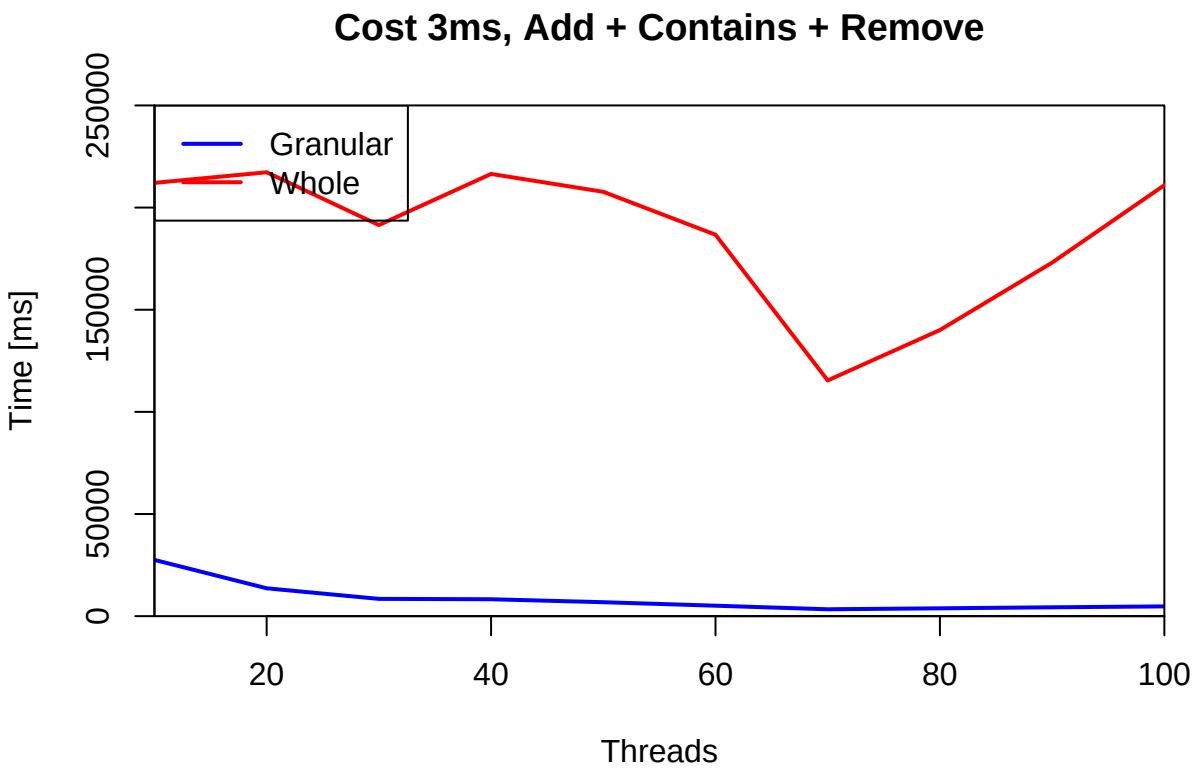
Remove i Contains są bardzo zbliżone, Add natomiast trwa sporo krócej. Dzieje sie tak dlatego że tylko w dwóch pierwszych wykonujemy operacje na wartościach elementów listy (porównanie), a więc tylko tam aplikowany jest koszt. Widzimy również że w obu przypadkach minimum czasu jest osiągane przy około 65 wątkach - przy kolejnych nie uzyskujemy już żadnej korzyści.

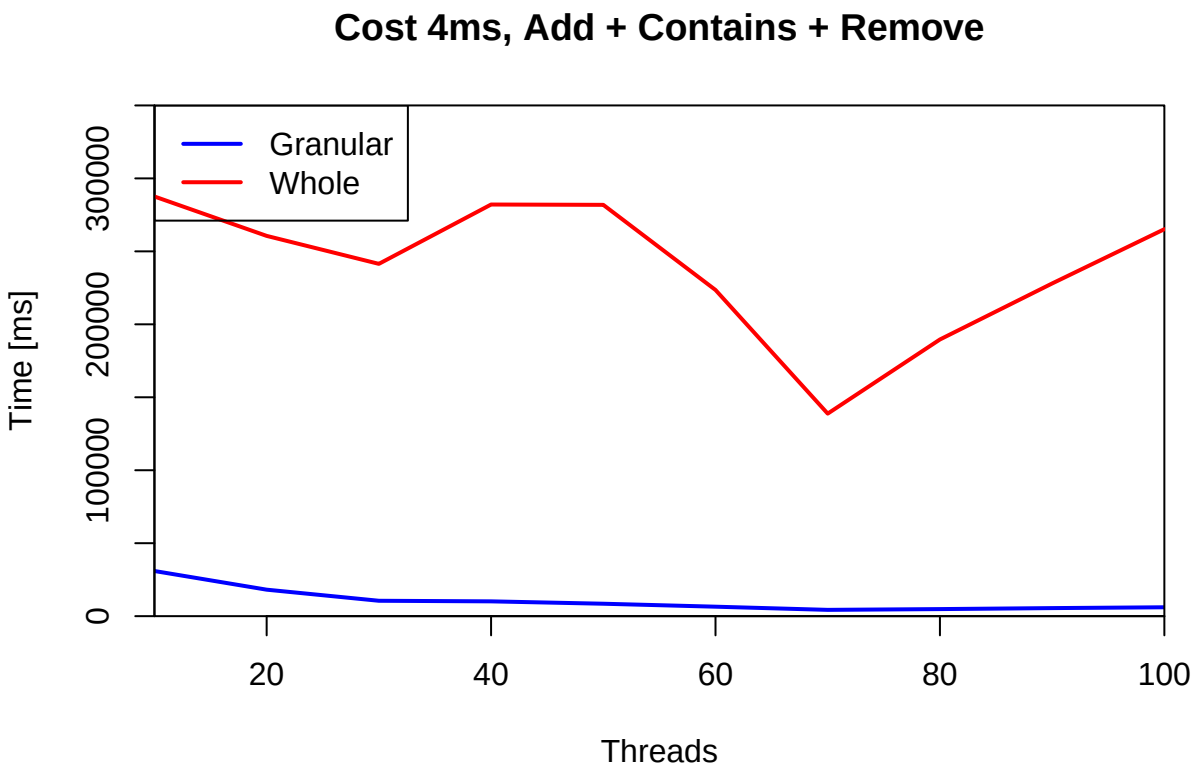
W następnej części będziemy dodawać czasy tych trzech operacji i porównywać na tej podstawie metody blokowania. Wyniki przy różnych kosztach:

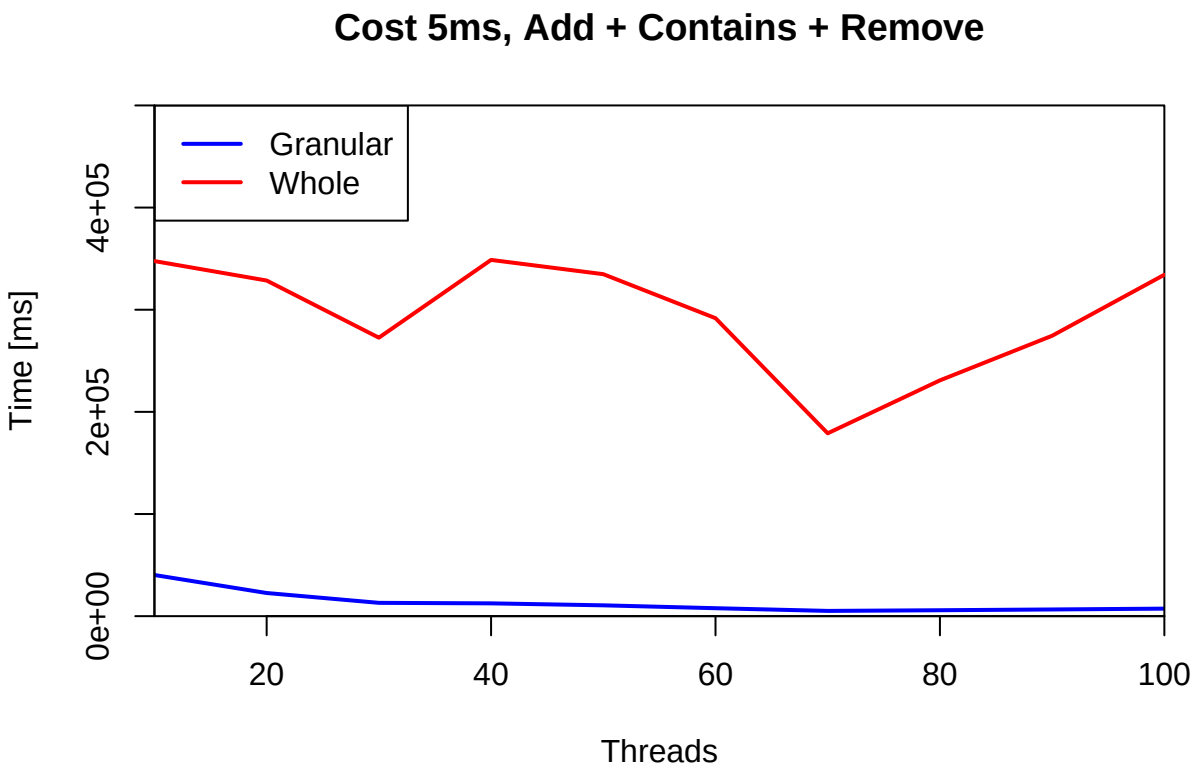

Cost 0ms, Add + Contains + Remove
Granular
Whole
Time [ms]
0
20
40
60
80
100
20
40
60
80
100
Threads


Cost 1ms, Add + Contains + Remove
Granular
Whole
Time [ms]
0
40000
80000
120000
20
40
60
80
100
Threads

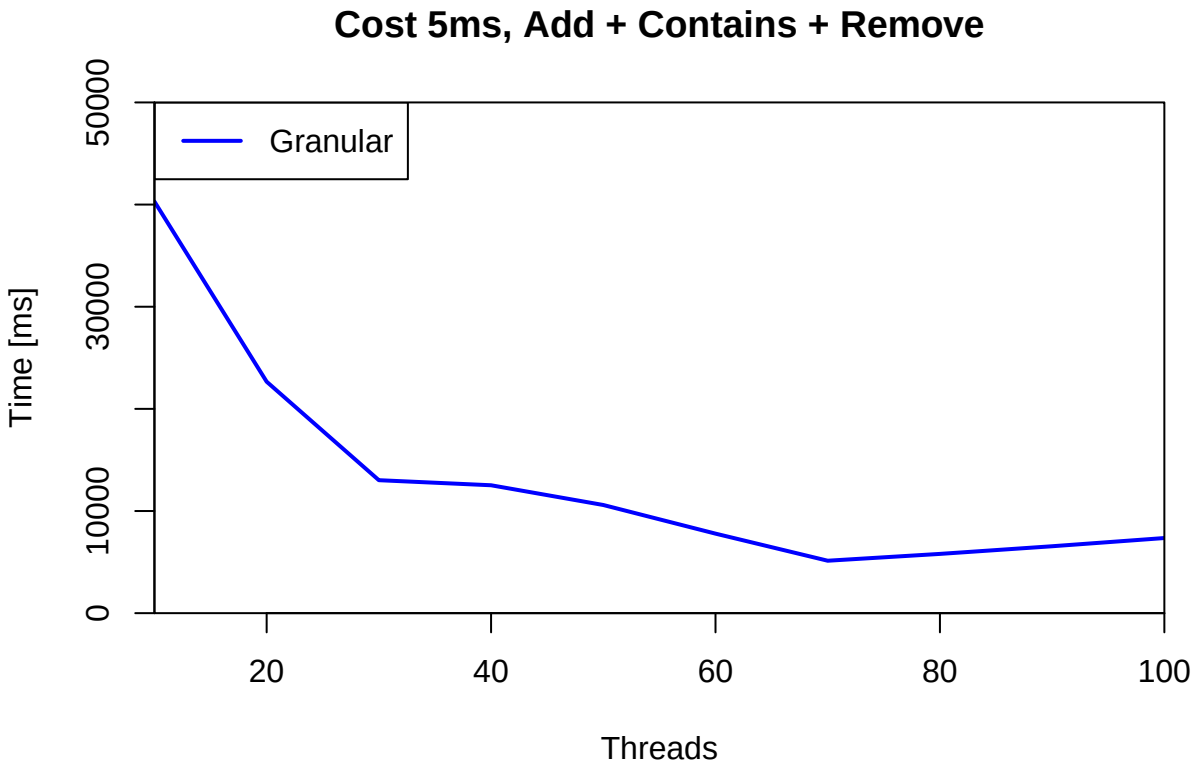

Cost 2ms, Add + Contains + Remove
Granular
Whole
Time [ms]
0
50000
100000
200000
20
40
60
80
100
Threads


Cost 3ms, Add + Contains + Remove
Granular
Whole
Time [ms]
0
50000
150000
250000
20
40
60
80
100
Threads


Cost 4ms, Add + Contains + Remove
Granular
Whole
Time [ms]
0
100000
200000
300000
20
40
60
80
100
Threads



Dla niezerowego kosztu możemy zauważyć że czas wykonania przy blokowaniu drobnoziarnistym spada wraz ze wzrostem ilości wątków, co nie ma miejsca przy blokowaniu całej listy. Dla zobrazowania wykres jedynie blokowania drobnoziarnistego dla kosztu 5ms:



Wnioski

Blokowanie drobnoziarniste pozwala w *znacznym* stopniu zwiększyć wydajność operacji na liście przy wykorzystaniu wielu wątków. Przy dużej ilości wątków różnice w czasie wykonania sięgały kilku rzędów wielkości.

Korzyść czasowa jest również tym większa, im większy koszt pojedynczej operacji.

Stosując blokowanie drobnoziarniste musimy jednak bardzo uważnie zaimplementować mechanizm blokowania / synchronizacji, ponieważ sporo łatwiej tutaj o wpadkę.