Programowanie Współbieżne

Algorytm Borůvki

Mateusz Hurkała

Spis treści

Rozdział 1

Algorytm

1.1 Idea algorytmu

Samą idee można przedstawić jako pętle po 3 częściach algorytmu.

```
Dopóki liczba składowych jest > 1:
    (1)Dla każdego wierzchołka v znajdź najlżejszą krawędź wychodzącą z v, taką że
    jej drugi koniec leży w innej składowej
    (2)Dla każdej składowej, znajdź najlżejszą krawędź pośród wierzchołków,
    które do niej należą
    (3)Dla każdej składowej, najlżejszą krawędź o ile nie łączy już dwóch
    połączonych składowych oraz połącz składowe
```

Okazuje się, że po każdym obrocie pętli liczba składowych spada conajmniej dwukrotnie co daje nam złożoność $\mathcal{O}((|V| + |E|)\log(|V|))$.

Dla wersji jednowątkowej, jak i dla wersji z strukturami bezpiecznymi wielowątkowo część (1) oraz (2) wykonamy jednocześnie, lecz potem przyda się nam ono do zrównoleglenia.

1.2 FindUnion

1.2.1 Idea FindUnion

Jak możemy zobaczyć w ideii algorytmu, potrzebna nam będzie struktura, która wystawi nam następujący interfejs:

```
public interface IFindUnion
{
    public int Find(int x);
    public void Union(int x, int y);
}
```

Funkcja Find(int x), będzię odpowiedzialna za zwrócenie reprezentanta składowej, co umożliwi nam stwierdzenie w jakiej składowej jesteśmy.

Natomiast funkcja Union(int x, int y) będzie odpowiedzialna za scalenie dwóch składowych (składowej, w której znajduje się x oraz składowej, w której znajduje się y).

1.2.2 Struktura reprezentowania FindUnion

FindUnion będziemy reprezentować jako listę następujących struktur:

```
private class Component
{
    public int Element { get; set; }
    private int rank { get; set; }

    public Component(int element)
    {
        Element = element;
        rank = 0;
    }
}
```

Taka struktura będzie nam pozwalała utrzymać odpowiednie informację. Pole **Element** będzie odpowiadało za przetrzymywanie naszego reprezentanta (Nie musi to być reprezentant całej składowej! Wystarczy nam fakt, że idąc cały czas po tym polu, ostatecznie do niego dojdziemy oraz fakt, że reprezentant wskazuje na samego siebie), oraz pole **rank** będzie oznaczało jak głębokie drzewo rozpina nasza składowa (jaka jest najdłuższa scieżka z dowolnego wierzchołka do reprezentanta, enumerując się po polu **Element**.

1.2.3 Inicjalizacja komponentów

Inicjalizacja samej klasy jest bardzo prosta i wystarczy zainicjalizować tablicę struktur gdzie pole **Element** ustawimy jako indeks tablicy;

```
private Component[] _list;

public FindUnionStructure(int n) => _list = Enumerable.Range(0, n)
    .Select(x => new Component(x)).ToArray();
```

1.2.4 Realizacja Find()

Implementacja funkcji Find(int x) jest bardzo prosta i wygląda następująco (dodatkowo zastosujemy tutaj kompresje scieżek, aby poprawić czas działania).

```
public int Find(int x) => FindInternal(x).Element;

private Component FindInternal(int x)
{
    var current = _list[x];
    if(current.Element == x)
        return current;
    var result = FindInternal(current.Element);
    current.Element = result.Element;
    return result;
}
```

1.2.5 Realizacja Union()

Implementacja Union(int x, int y) również jest dosyć prosta. Najpierw zobaczmy funkcje, która mając dwóch reprezentantów, scali ich w jedną składową (będzie to funkcja z klasy Component):

```
public void MergeWith(Component component)
{
    if(Element == component.Element)
        return;
    if(rank > component.rank)
        component.Element = Element;
    else if(rank < component.rank)
        Element = component.Element;
    else
    {
        Element = component.Element;
        rank++;
        component.rank++;
    }
}
```

Funkcja ta odpowiednio ustawia komponentowi o niższej głębokości reprezentanta na drugi komponent, oraz aktualizuje odpowiednio pole **rank**.

Mając taką funkcję możemy bardzo prosto zrealizować funkcje Union(int x, int y):

```
public void Union(int x, int y) => FindInternal(x).MergeWith(FindInternal(y));
```

Można pokazać, że tak wykonane findUnion będzie miało złożoność czasową $\mathcal{O}(n \cdot \alpha(n))$, gdzie $\alpha(n)$ jest odwrotną funkcją Ackermanna.

1.3 Implementacja jednowątkowa

Jeżeli chodzi o samą inicjalizację komponentów to potrzebujemy jedynie listy krawędzi oraz stuktury FindUnion, więc sama inicjalizacja może wyglądać następująco:

```
public class BoruvkaMST : IMST
{
    private IFindUnion _findUnion;
    private List<List<Pair<Edge<int>, long>>> _adj;
    private int n;

    public BoruvkaMST(List<List<Pair<Edge<int>, long>>> adj, int n)
    {
        _adj = new (adj);
        this.n = n;
        _findUnion = new FindUnionStructure(n);
    }
}
```

Następnie sam kod algorytmu możemy przedstawić za pomocą wcześniej opisanego schematu (w tym przypadku punkt 1) oraz 2) zrobimy jednoczesnie):

```
public (IEnumerable<Edge<int>> edges, long cost) GetMST()
{
    if(n < 2)
        return (Enumerable.Empty<Edge<int>>(), 0);
    var resultTree = new List<Edge<int>>();
    long cost = 0;
    while(true)
    {
        Dictionary<int, Pair<Edge<int>, long>> dict = new Dictionary<int, Pair<Edge<
        for(int k=0;k<n;k++)
        {
            var edge = FindMinimalEdgeForVertex(k);
            if(edge is null)
                continue;
            AddOrUpdate(dict, _findUnion.Find(k), edge);
        }
        if(dict.Count <= 1)
            break;
        foreach(var x in dict)
        {
            var firstIdx = _findUnion.Find(x.Value.First.First);
            var secIdx = _findUnion.Find(x.Value.First.Second);
            if(firstIdx == secIdx)
                continue;
            _findUnion.Union(firstIdx, secIdx);
            resultTree.Add(x.Value.First);
            cost += x.Value.Second;
        }
    }
    return (resultTree, cost);
}
```

Zauważmy, że inicjalizacja słownika jak i pierwsza pętla for jest odpowiedzialna za punkty 1) oraz 2). Słownik przetrzymuje nam dla każdej składowej najlżejszą krawędź prowadzącą do innej składowej. Natomiast pętla foreach realizuje dokładnie punkt 3).

Funkcja AddOrUpdate() jest odpowiedzialna jedynie za podmiane wartości krawędzi na mniejszą dla danej składowej. Natomiast troszkę ciekawsza jest funkcja FindMinimalEdgeForVertex(int k).

```
private Pair<Edge<int>, long> FindMinimalEdgeForVertex(int x)
{
    var conectedNumber = _findUnion.Find(x);
    var edges = _adj[x].Where(x => _findUnion.Find(x.First.Second) != conectedNumber
    _adj[x] = edges;
    if(edges is null || !edges.Any())
    {
        if(_adj[x].Any())
            _adj[x] = new();
        return null;
    }
```

```
        return FindMinimalEdge(edges);
    }

    private Pair<Edge<int>, long> FindMinimalEdge(
            IEnumerable<Pair<Edge<int>, long>> edges)
        => edges.Aggregate((prev, next) => prev.Second > next.Second ? next : prev);
```

Sama funkcja FindMinimalEdge(Edges) jest poprostu owiniętą pętlą wyłuskującą najmniejszą krawędź listy więc nie ma tutaj dużo do omówienia. Natomiast w funkcji FindMinimalEdgeForVertex(int x) ważną optymalizajcą jest ucięcie krawędzi nie potrzebnych. Najpierw odfiltrowywujemy, które krawędzie prowadzą do innej składowej a potem podmieniamy listę krawędzi dla naszego wierzchołka na przefiltorwaną (przyśpiesza to program około dwukrotnie!). Mając tak napisany kod cały algorytm już działa.

Rozdział 2

Zrównoleglanie

2.1 Implementacja ze strukturami wielowątkowymi i Thread poolem

O dziwo najszybszym programem okazuje się implmentacja na ConcurrentDictornary oraz wykorzystaniem thread poola. Sama inicjalizacja klasy zostaje dokładnie taka sama, natomiast zmienia się troszkę pętla w programie.

```
public (IEnumerable<Edge<int>> edges, long cost) GetMST()
{
    if(n < 2)
        return (Enumerable.Empty<Edge<int>>(), 0);
    var resultTree = new ConcurrentBag<Edge<int>>();
    long cost = 0;
    while(true)
    {
        ConcurrentDictionary<int, Pair<Edge<int>, long>> dict =
            new ConcurrentDictionary<int, Pair<Edge<int>, long>>();
        Parallel.For(0, n, new ParallelOptions
            { MaxDegreeOfParallelism = MAX_THREADS},
            k => ConsiderVertex(k, dict));
        if(dict.Count < 2)
            break;
        dict.AsParallel().ForAll(x => MergeComponents(x, ref cost, resultTree));
    }
    return (resultTree, cost);
}
```

Funkcja ConsiderVertex() dodaje do słownika dla każdego wierzchołka minimalną jego krawędź idącą do innej składowej (krawędź dodawana jest pod indeks składowej). Niestety w tym przypadku jak i w pozostałych nie udało mi się zrównolegilić 3) punktu algorytmu więc funckja MergeComponents() zakłada lock'a na strukturze FindUnion. Przykładem dla którego algorytm nie zadaziała bez owego locka jest graf trójkątny, który ma wszytkie krawędzie tego samego kosztu (ten przypadek akurat dosyc prosto rozwiązać, lecz wtedy znajdą się inne trudniejsze przypadki), a więc funkca wygląda następująco:

```
private void MergeComponents(int x)
```

```
    {
        var pair = _list[x];
        lock(_findUnion)
        {
            var firstIdx = _findUnion.Find(pair.First.First);
            var secIdx = _findUnion.Find(pair.First.Second);
            if(firstIdx == secIdx)
                return;
            _findUnion.Union(pair.First.First, pair.First.Second);
            resultTree.Add(pair.First);
            cost += pair.Second;
        }
    }
```

2.2 Zrównoleglanie punktu 2)

Punkt 2) odpowiadał za znalezienie krawędzie o najmniejszym koszcie dla danej składowej, więc możemy założyć, że mamy już policzoną najlżejszą krawędź dla każdego wierzchołka z punktu 1). Aby wykonać ten punkt całkowicie równolegle musismy rozszerzyć iterfejs naszego FindUnion o dodatkową metodę:

```
public List<int> GetMembers(int x)
```

Która ma za zadanie zwrócić listę wierzchołków, które są w danej składowej (zakładamy, że zawsze do metody będziemy przekazywali lidera). Realizacja takiej funkcji jest bardzo prosta. Wystarczy, że teraz każdy komponent będzie trzymał listę swoich dzieci oraz podczas Union lider, który zostaje scalony zostanie dodany do listy nowego lidera. Przykładowa implementacja:

```
    public List<int> GetMembers(int x)
    {
        var result = new List<int>{x};
        GetMembers(x, result);
        return result;
    }

    private void GetMembers(int x, List<int> result)
    {
        foreach(var item in _list[x]._list)
        {
            result.Add(item);
            GetMembers(item, result);
        }
    }
```

2.3 Wersja na puli wątków

Implementacja na puli wątków wygląda najprzyjemniej:

```
        while(true)
        {
            isEdge = false;
```

```
            Parallel.For(0, n,
                new ParallelOptions { MaxDegreeOfParallelism = MAX_THREADS},
                ConsiderVertex);
            if(!isEdge)
                break;
            Parallel.For(0, n,
                new ParallelOptions { MaxDegreeOfParallelism = MAX_THREADS},
                ComposeComponent);
            Parallel.For(0, n,
                new ParallelOptions { MaxDegreeOfParallelism = MAX_THREADS},
                MergeComps);
        }
```

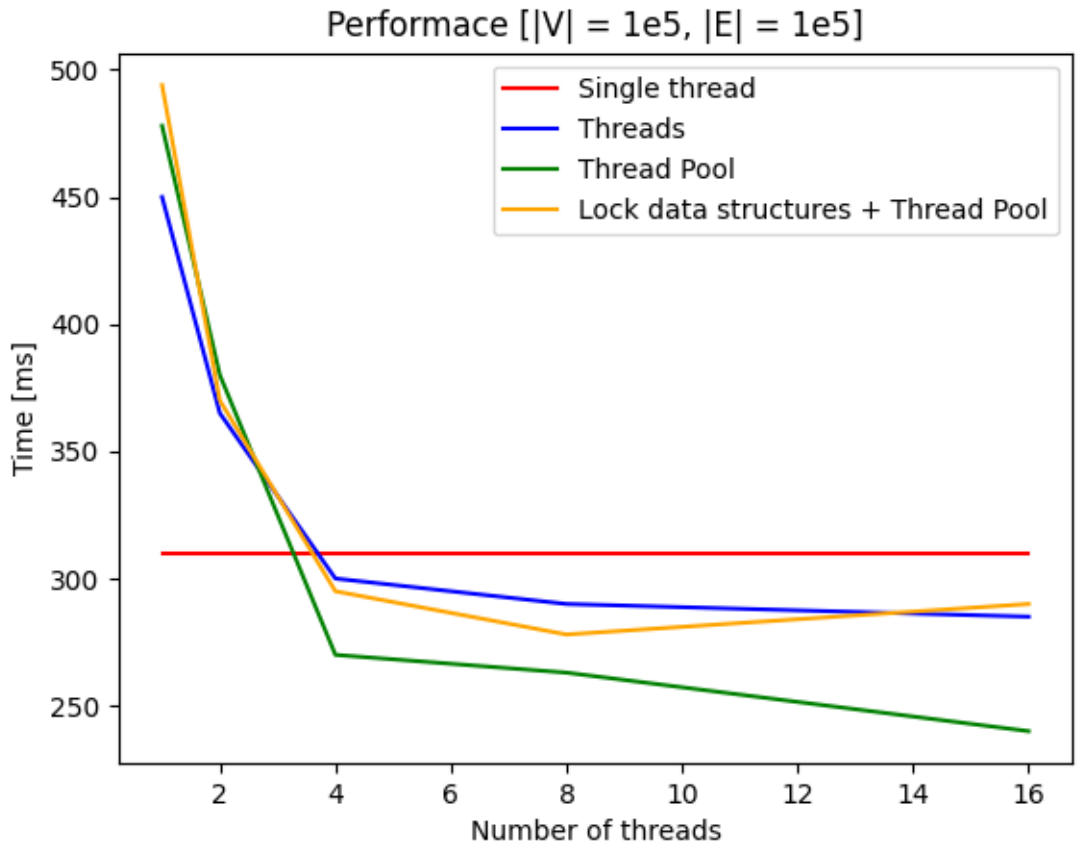
Sama implementacja od poprzedniej różni się tylko faktem, że nie musimy synchronizować się ręcznie lecz wykonujemy każdą z wcześniej wymienionych funkcji w równoległej pętli for. (funkcja ComposeComponents() jest wyciągniętą do funkcji implementacją z poprzedniej wersji)

Rozdział 3

Porównanie wyników

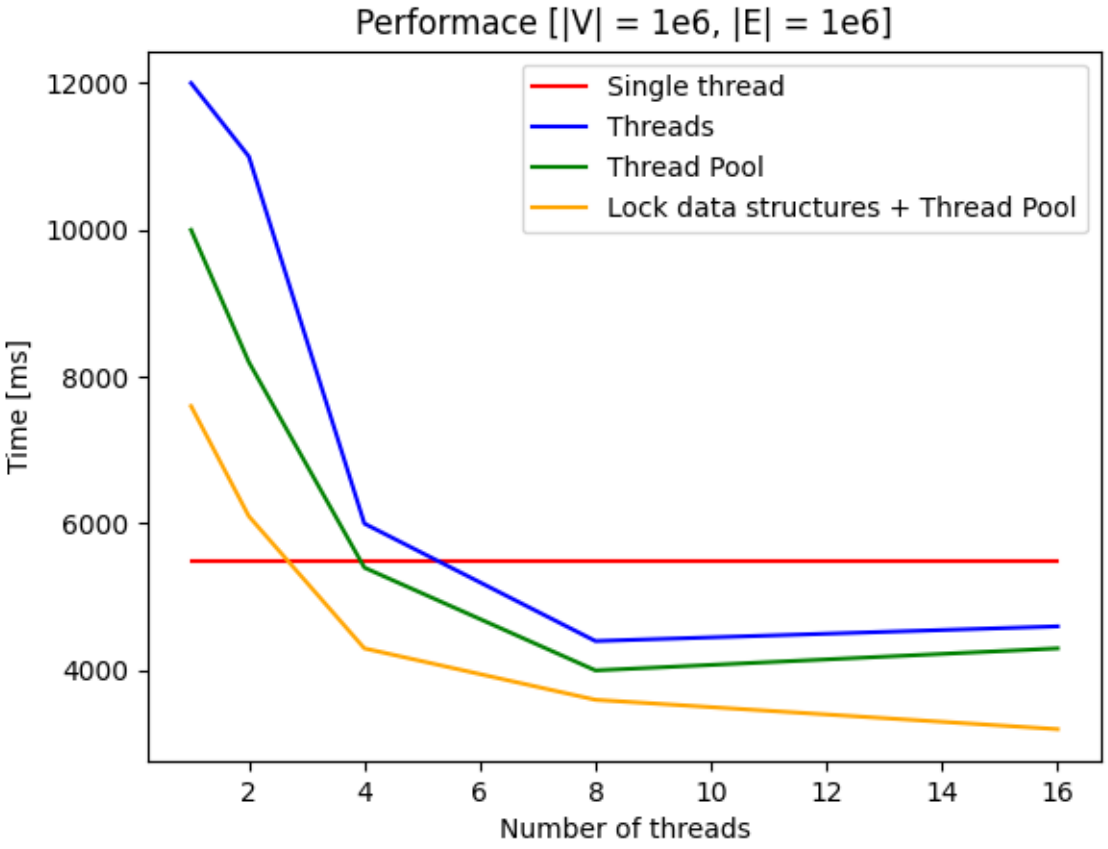
3.1 Graf rzadki małego rozmiaru

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^5, |E| = 10^5$



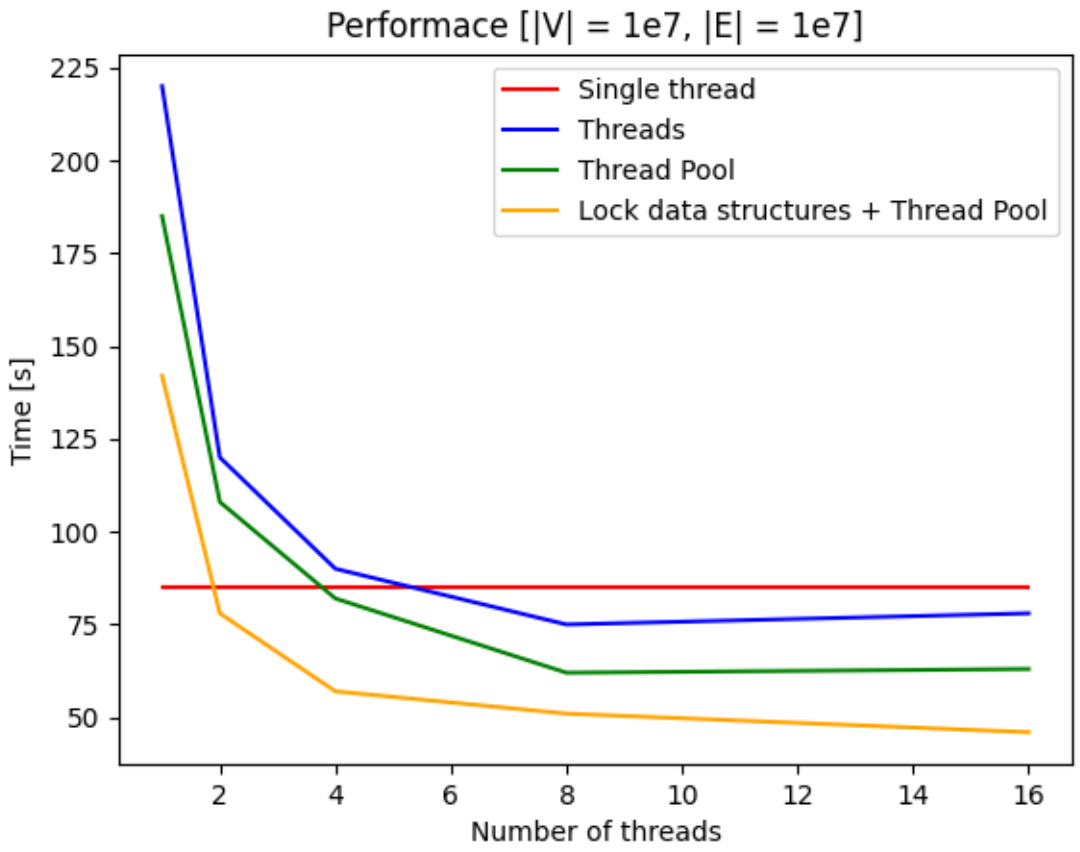
3.2 Graf rzadki średniego rozmiaru

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^6, |E| = 10^6$



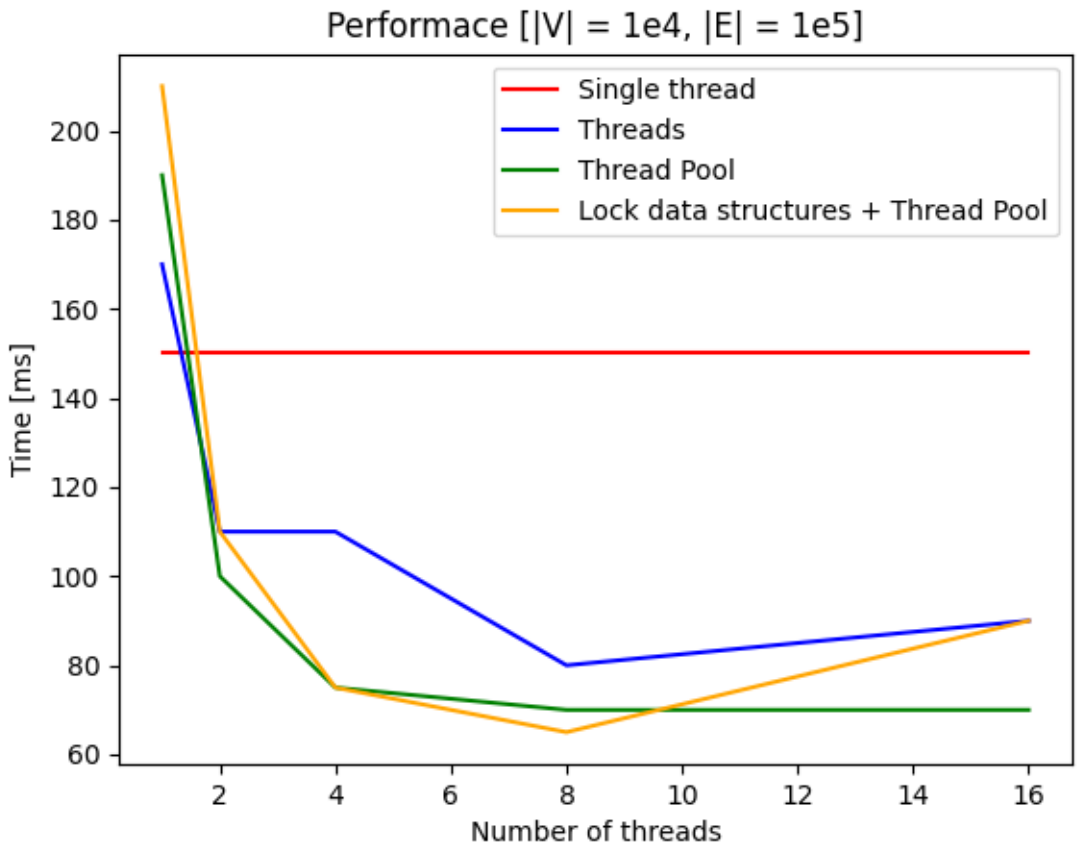
3.3 Graf rzadki dużego rozmiaru

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^7, |E| = 10^7$



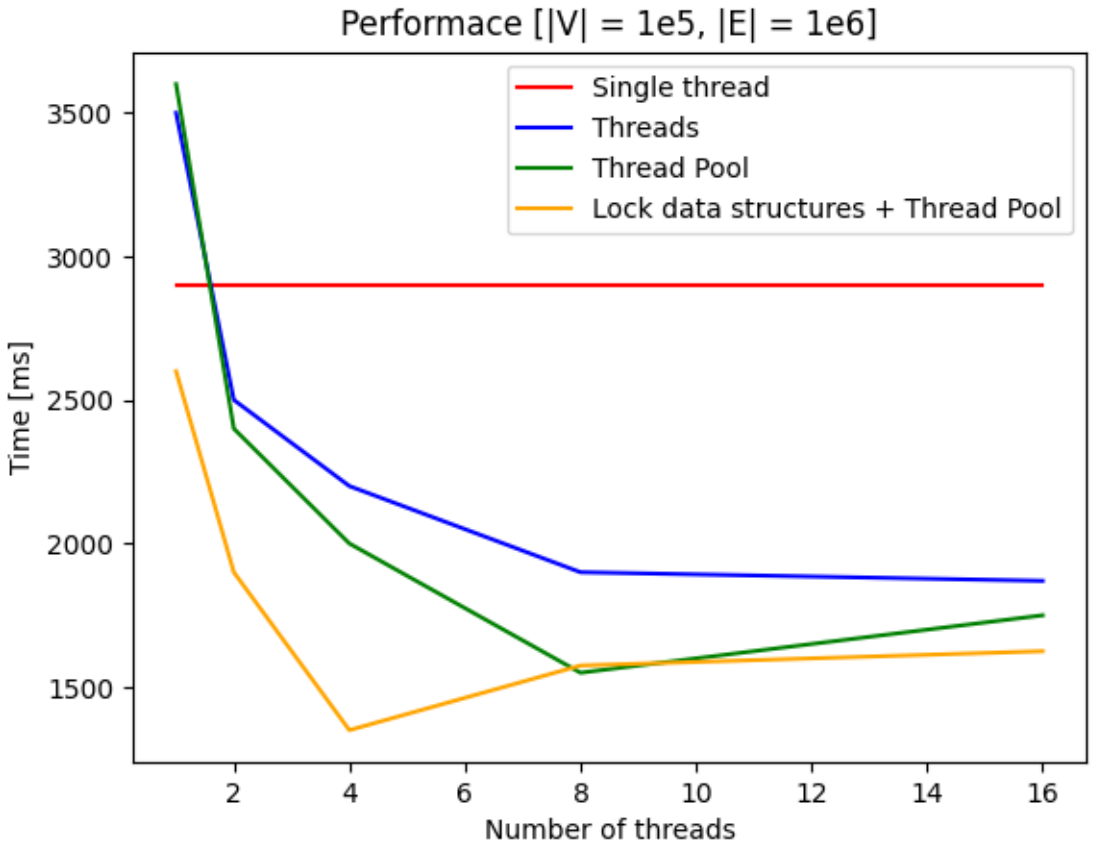
3.4 Graf z dziesięciokrotną ilością krawędzi względem ilości wierzchołków małego rozmiaru

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^4, |E| = 10^5$



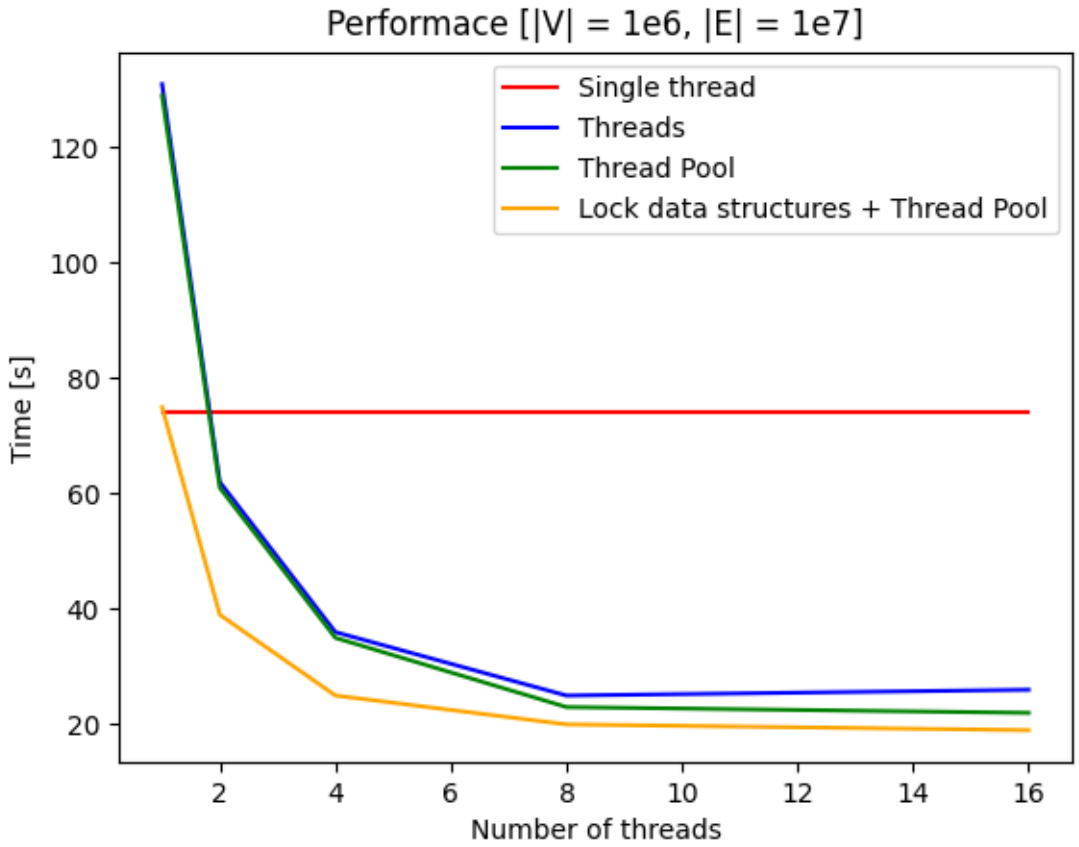
3.5 Graf z dziesięciokrotną ilością krawędzi względem ilości wierzchołków średniego rozmiaru

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^5, |E| = 10^6$



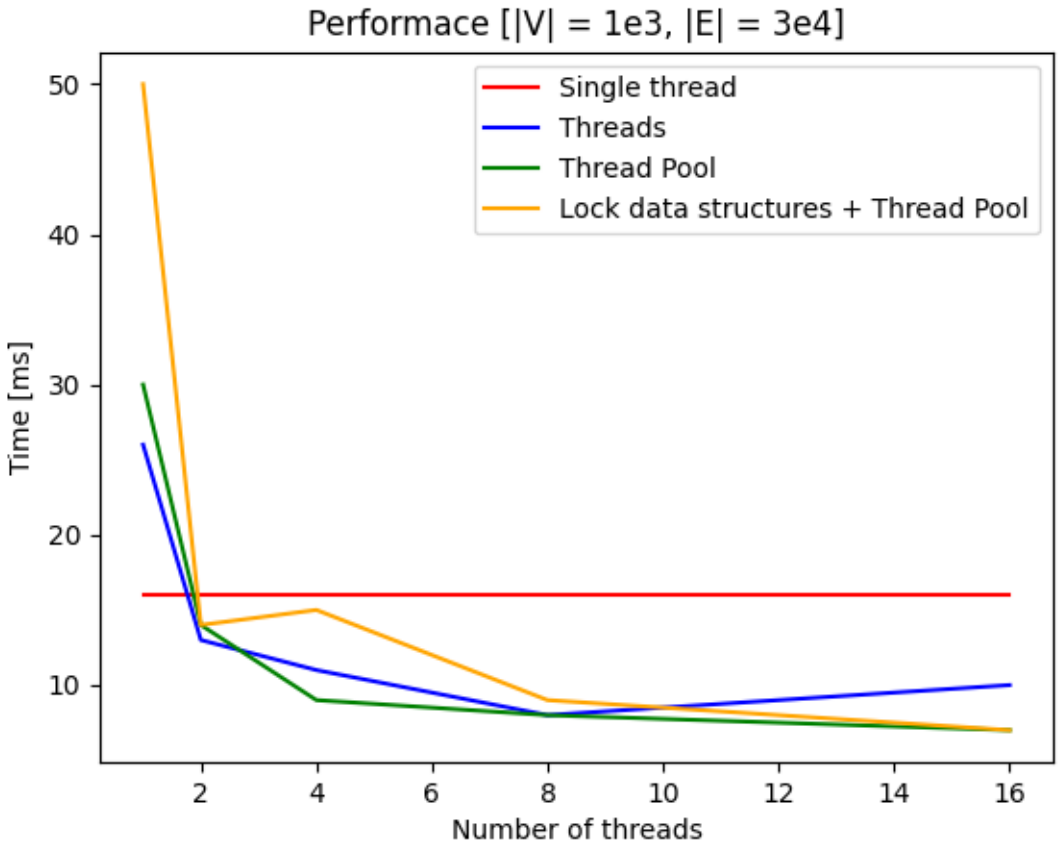
3.6 Graf z dziesięciokrotną ilością krawędzi względem ilości wierzchołków dużego rozmiaru

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^6, |E| = 10^7$



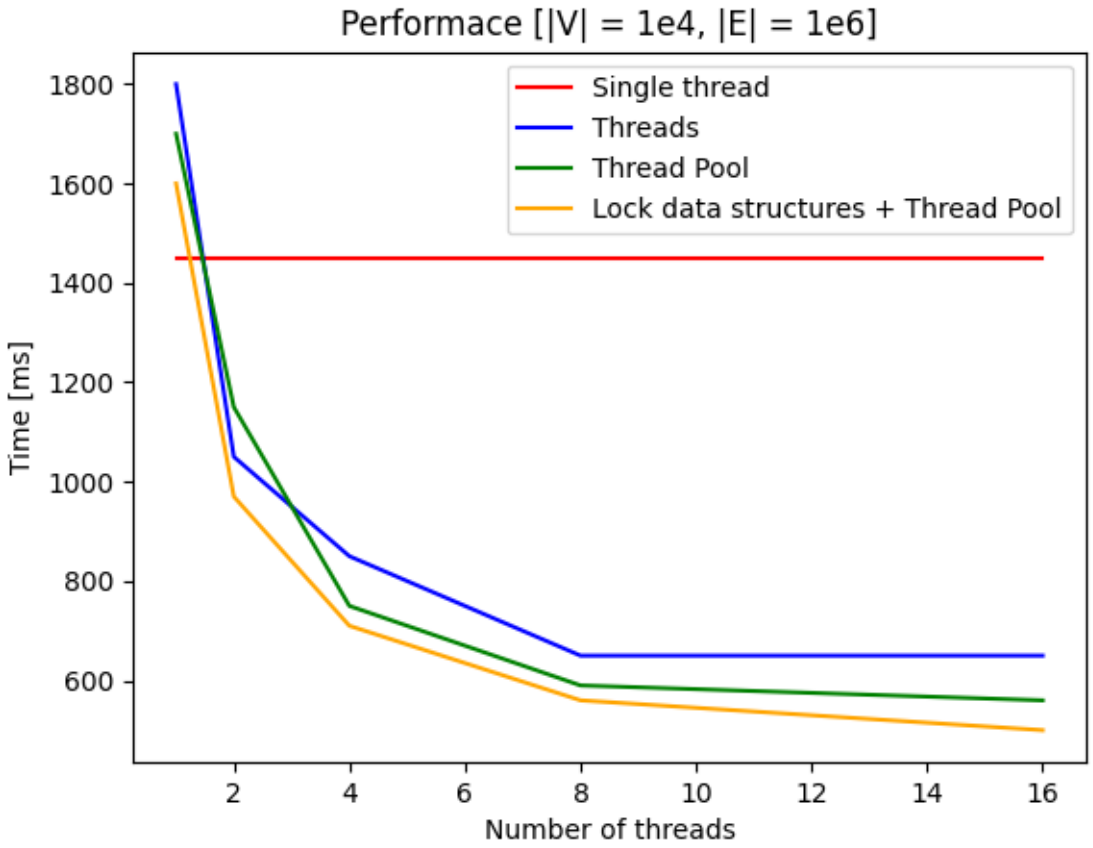
3.7 Graf małego rozmiaru z ilością krawędzi rzędu $|V| \cdot \sqrt{|V|}$

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^3, |E| = 3 \cdot 10^4$



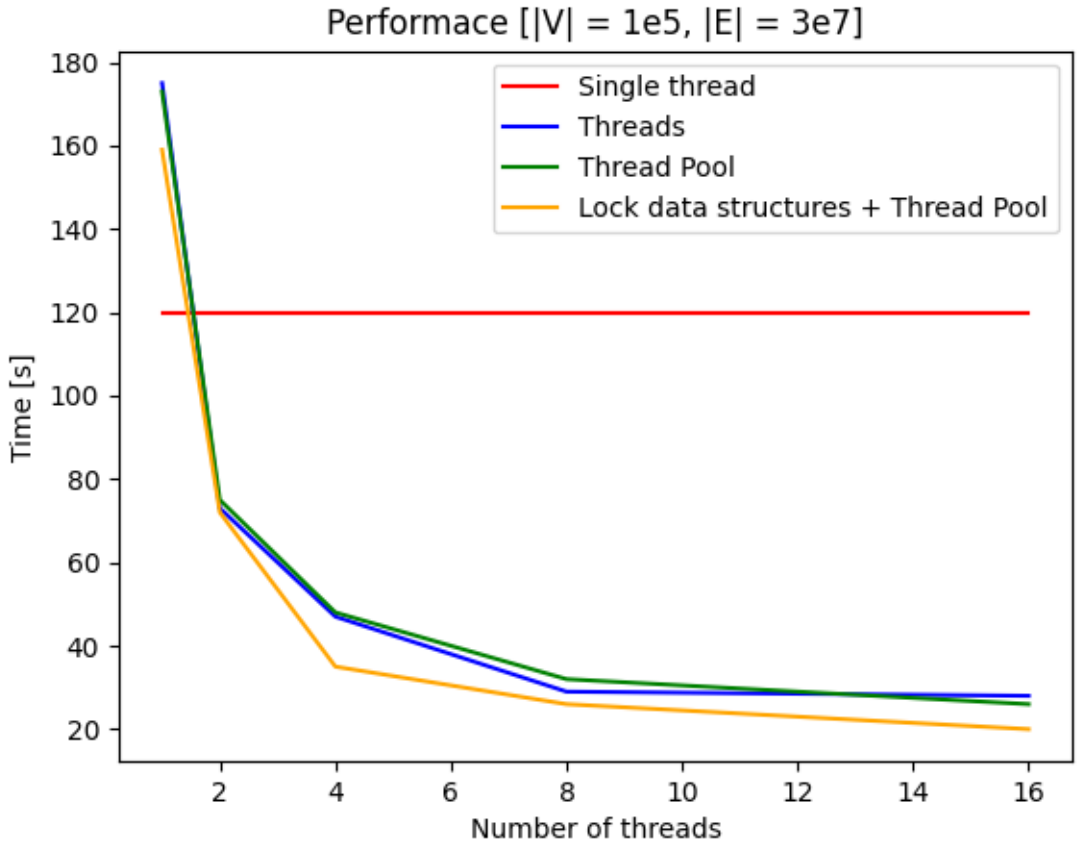
3.8 Graf średniego rozmiaru z ilością krawędzi rzędu $|V| \cdot \sqrt{|V|}$

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^4, |E| = 10^6$



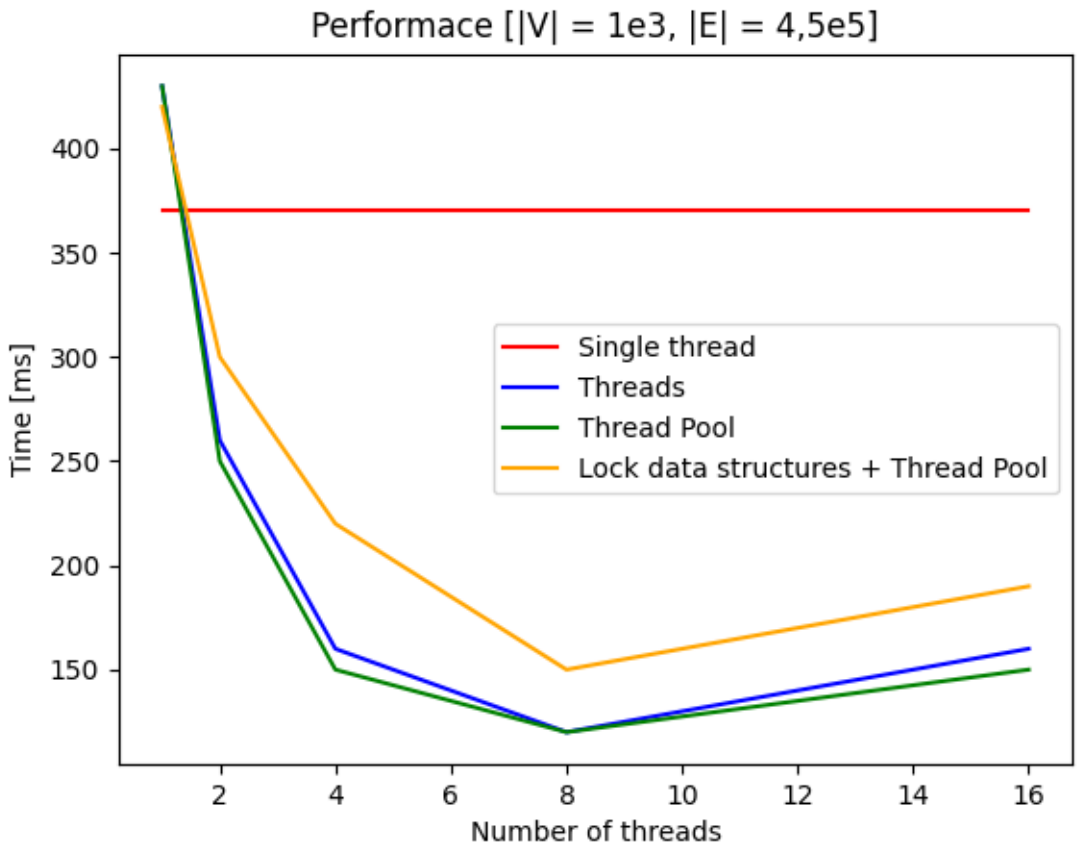
3.9 Graf dużego rozmiaru z ilością krawędzi rzędu $|V| \cdot \sqrt{|V|}$

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^5, |E| = 3 \cdot 10^7$



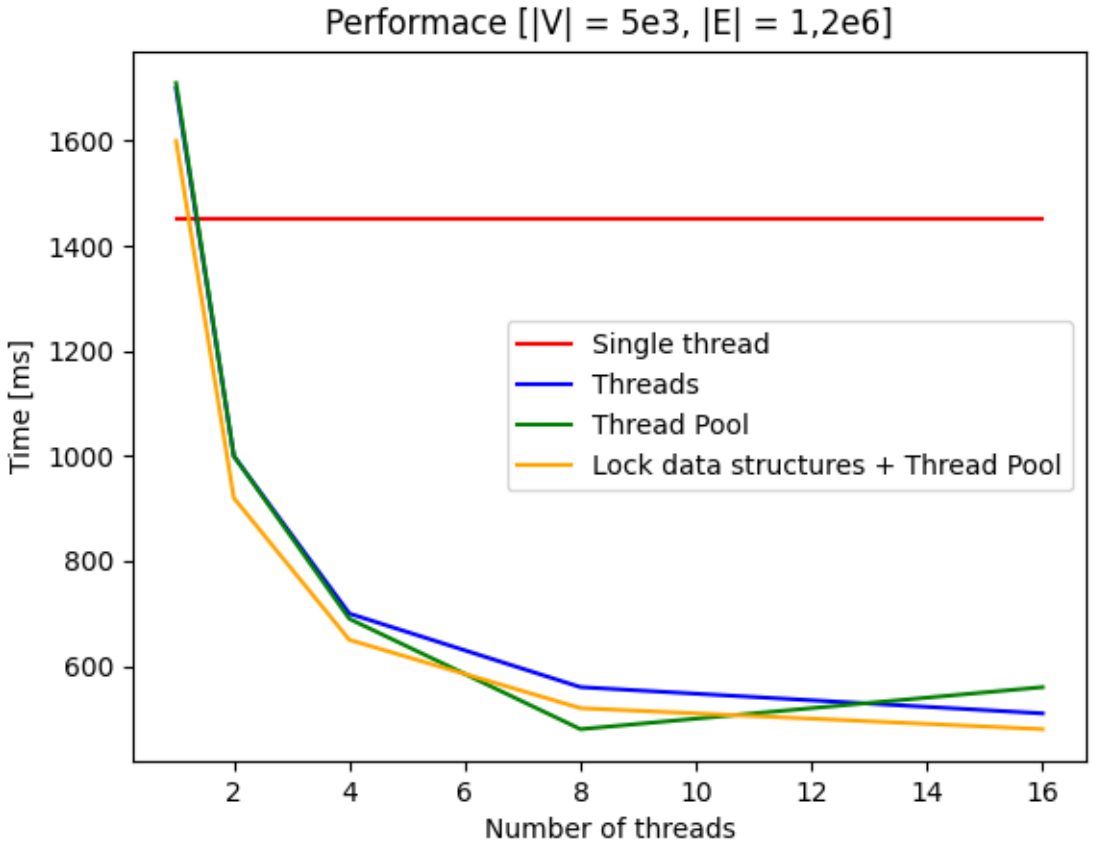
3.10 Graf prawie pełny małego rozmiaru

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^3, |E| = 4,5 \cdot 10^5$



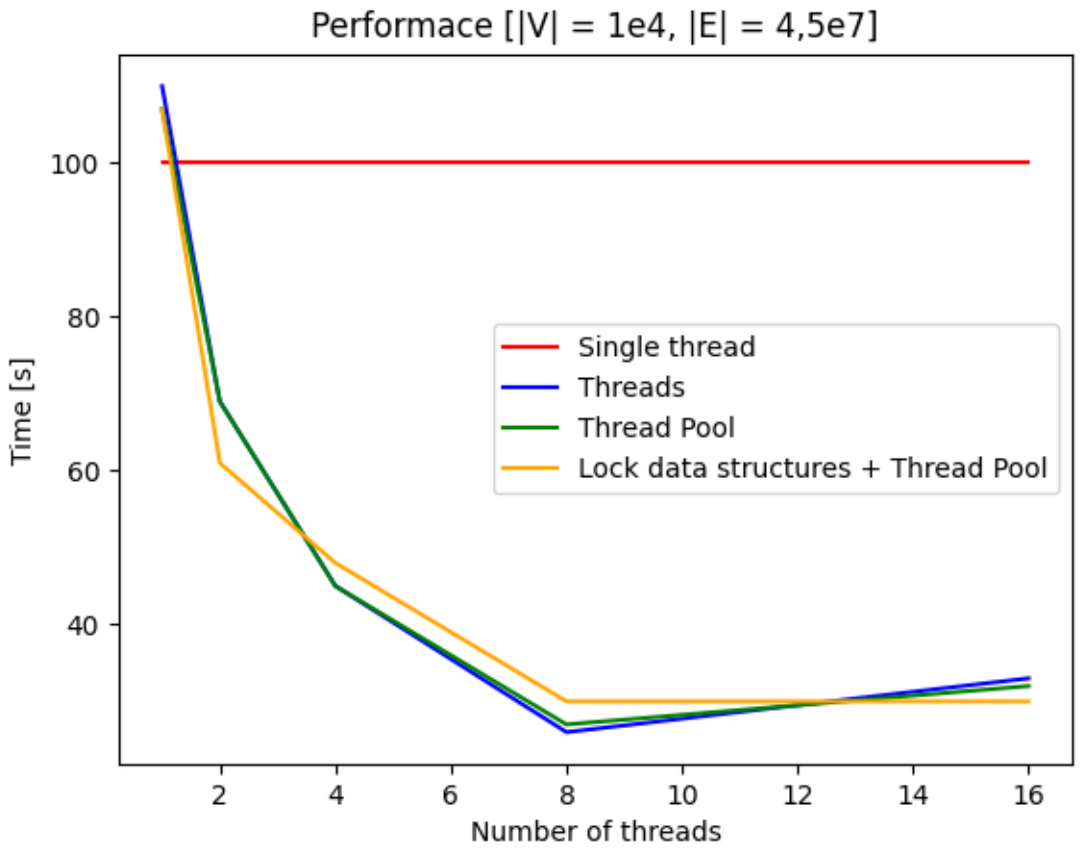
3.11 Graf prawie pełny średniego rozmiaru

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 5{\cdot}10^3, |E| = 1,2{\cdot}10^6$



3.12 Graf prawie pełny dużego rozmiaru

Porównanie podejścia czasu wykonania algorytmu dla grafu rozmiaru $|V| = 10^4, |E| = 4,5 \cdot 10^7$



3.13 Podsumowanie wyników

Posumowując powyższe wykresy możemy zobaczyć, że wersje wielowątkowe osiągają do 4-krotnie lepszego czasu wykonania. (Testy zostały uruchomione na 8 wątkowym komputrze dlatego dla większej ilości niż 8 wątków czasy zaczynają się pograszać. Zazwyczaj najszybszą wersją okazywała się implementacja na strukturach wielowątkowych i puli wątków, lecz poza nią najlepiej działała wersja na samej puli wątków.