Algorytmy równoległe 2015 (zad. 2)

Michał Liszcz

2015-11-25

Contents

1 Wstęp

Treść zadania została podana przez prowadzącego:

Zaimplementować algorytm obliczania spójnej składowej (*connected components*) zgodnie z modelem *Pregel* [1]. Zasada działania algorytmu:

1. Na początku każdy wierzchołek oznaczamy innym znacznikiem (liczbą).
2. Każdy wierzchołek wysyła swój znacznik do wszystkich sąsiadów.
3. Jeżeli minimum zotrzymanych zniaczników jest mniejsze od własnego, wierzchołek zastępuje swój znacznik przez minimum otrzymanych.
4. Powtarzamy krok 2 aż przestaną zachodzić zmiany

2 Implementacja z wykorzystaniem Apache Spark

Implementacja bazuje na strukturze `org.apache.spark.rdd.RDD`.

2.1 Struktura grafu

Graf zadany jest jako lista **skierowanych** krawędzi, łączących wierzchołki o zadanych indeksach, przykładowo:

```
0 1
0 2
0 3
2 4
5 6
6 7
7 4
8 9
```

Struktura taka ma bezpośrednie przełożenie na `RDD`:

```
type IntMapRDD = RDD[(Int, Int)]
```

Taki graf można łatwo przekształcić w graf nieskierowany:

```
def makeUndirected(edges: IntMapRDD) =
    (edges ++ edges.map(_.swap)).distinct
```

2.2 Realizacja algorytmu

Dążymy do zdefiniowania funkcji transformującej graf w zbiór spójnych składowych:

```
def connectedComponents(graph: IntMapRDD): RDD[Iterable[Int]]
```

Definiujemy połączenia w grafie jako mapę: `K -> zbiór wierzchołki wychodzących z K` oraz wprowadzamy wagi wierzchołków (każdy wierzchołek zaczyna z wagą równą jego indeksowi):

```
val connections = graph.groupByKey
val initWeights = connections map { case (k, _) => (k, k) }
```

W każdym kroku iteracji zmieniamy wagi: każdy z wierzchołków `K` otrzymuje wagę będącą minimum z wag jego sąsiadów (i jego samego). Taką operację należy wykonać, unikając zagnieżdżonych operacji na `RDD`. Można to osiągnąć następująco:

1. `join` (względem klucza) zbiorów `connections` i `weights`,
2. pobranie wartości (*values*) powyższego `RDD`. Otrzymamy pary zawierające wagę wierzchołka `K` i listę wierzchołków do których ta waga będzie przesłana,
3. każdy element z poprzedniego wyniku mapujemy na pary wierzchołek -> nowa waga, wynik wypłaszczamy (*flatten*)
4. poprzedni wynik grupujemy po kluczu, otrzymamy pary wierzchołek -> lista wag które otrzyma od sąsiadów
5. mapujemy wartości w poprzednim wyniku, wybierając minimum z zadanej listy

Punkty 4. i 5. można zastąpić jedną operacją: `combineByKey(weight => weight, Math.min, Math.min)`, lub prościej: `reduceByKey(Math.min)`

Implementacja tych operacji:

```
val newWeights = connections.join(weights).values.flatMap {
    case (indices, weight) => indices map { (_, weight) }
} reduceByKey(Math.min)
```

Wyliczone nowe wartości wag nie uwzględniają poprzedniej wagi (jeżeli jest najmniejsza, wierzchołek nie powinien zmieniać wagi). Należy złączyć oba zbiory wag, dla każdego wierzchołka wybierając mniejszą wagę:

```
val mergedWeights = weights.join(newWeights)
                .mapValues((Math.min _).tupled)
```

Powyżej zdefiniowane operacje należy powtarzać, dopóki w wagach zachodzą zmiany. Oczywistym rozwiązaniem wydaje się być rekurencja:

```
@tailrec
def performStep(weights: IntMapRDD): IntMapRDD = {

    val newWeights = connections.join(weights).values.flatMap {
        case (indices, weight) => indices map { (_, weight) }
    } reduceByKey(Math.min)

    val mergedWeights = weights.join(newWeights)
        .mapValues((Math.min _).tupled)

    if (weights.subtract(mergedWeights).count == 0)
        mergedWeights else performStep(mergedWeights)
}
```

UWAGA: *we współczesnych wersjach Apache Spark dostępna jest metoda `RDD.isEmpty`. Można jej użyć zamiast przyrównywania rozmiaru do zera.*

Wynik otrzymany z powyższej rekurencji można ostatecznie zamienić na zbiory spójnych składowych transformacją:

```
performStep(initWeights).map(_.swap).groupByKey.map(_._2)
```

UWAGA: *w rozwiązaniu należy pamiętać o cache-owaniu zbiorów wielokrotnie używanych.*

3 Testy lokalne

W celu zbadania wpływu ilości procesorów na czas rozwiązywania problemu, uruchomiłem program w konfiguracji lokalnej na maszynie z procesorem Intel Core i5-4200u, 2C/4T. Docelowo testy będą przeprowadzone na klastrze Zeus w ACK Cyfronet AGH.

Do testów wybrałem zbiór *ca-GrQc* [1] o 5242 wierzchołkach i 14496 krawędziach.

[1] https://snap.stanford.edu/data/ca-GrQc.html

Ilość procesorów dostępnych dla Apache Spark zmieniałem w zakresie 1-4. W każdym wypadku pomiar powtórzyłem czterokrotnie. Wyniki przedstawia poniższa tabela.

wątki	czas	błąd
1	4.366	1.308
2	3.834	1.215
3	3.853	1.488
4	3.615	1.527

Widać nieznaczny wzrost wydajności.

4 Testy na klastrze Zeus

Z powodu problemów z dostępem do klastra Zeus, testy przeprowadziłem na maszynie wyposażonej w dwa procesory Intel Xeon E5-2680 v3 [2] (12C/24T, łącznie 48 logicznych procesorów).

```
/var/fpwork/liszcz $ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                48
On-line CPU(s) list:   0-47
Thread(s) per core:    2
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Stepping:              2
CPU MHz:               2500.000
BogoMIPS:              5051.02
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-11,24-35
NUMA node1 CPU(s):     12-23,36-47
```

UWAGA: Na potrzeby testów konfigurowałem **lokalną instalację** Apache Spark (`-master local[X]`). Komunikacja odbywa się w obrębie jednej instancji maszyny wirtualnej Java.

4.1 Graf testowy

Do testów wydajności wykorzystałem graf *p2p-Gnutella24* [3] (26'518 wierzchołków i 65'369 krawędzi), *p2p-Gnutella31* [4] (62'586 wierzchołków i 147'892 krawędzi) oraz użyty wcześniej *ca-GrQc*.

[2] http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz
[3] https://snap.stanford.edu/data/p2p-Gnutella24.html
[4] https://snap.stanford.edu/data/p2p-Gnutella31.html

4.2 Losowo generowane grafy

Drugi wariant zakładał testy na losowo wygenerowanym grafie. Wykorzystałem obiekt `GraphGenerators`. [5]

Generator pozwala na stworzenie grafu o zadanej liczbie wierzchołków i losowych krawędziach, przy czym stopnie wierzchołków grafu są losowane z rozkładem logarytmicznie normalnym.

Wygenerowany graf można przekształcić na opisany wcześniej `IntMapRDD`:

```
GraphGenerators.logNormalGraph(sc, vertices, seed = 1).edges.map {
    edge => (edge.srcId.toInt, edge.dstId.toInt)
}
```

Przyjąłem stałą wartość dla ziarna generatora pseudolosowego, aby zapewnić porównywalność wyników uzyskanych w kolejnych uruchomieniach.

4.3 Wyniki

Dla ustalonego rozmiaru klastra mierzyłem czas wyznaczania spójnych składowych. W każdym przypadku pomiar powtórzyłem czterokrotnie. Jako niepewność przyjąłem odchylenie standardowe średniej otrzymanych wyników.

Wykorzystałem następujące definicje przyspieszenia $S(x,p)$ i efektywności $E(x,p)$:

$$S(x,p) = \frac{T(x,1)}{T(x,p)} \tag{1}$$

$$E(x,p) = \frac{S(x,p)}{p} \tag{2}$$

W powyższych definicjach x oznacza rozmiar problem, natomiast p to liczba procesorów. Niepewności oszacowałem metodą różniczki zupełnej.

Liczbę węzłow zmieniałem w zakresie od 1 do 12. Powyżej tej ilości nie było widać żadnej poprawy, natomiast występował spadek wydajności.

[5] http://spark.apache.org/docs/latest/api/scala/#org.apache.spark.graphx.util.GraphGenerators$

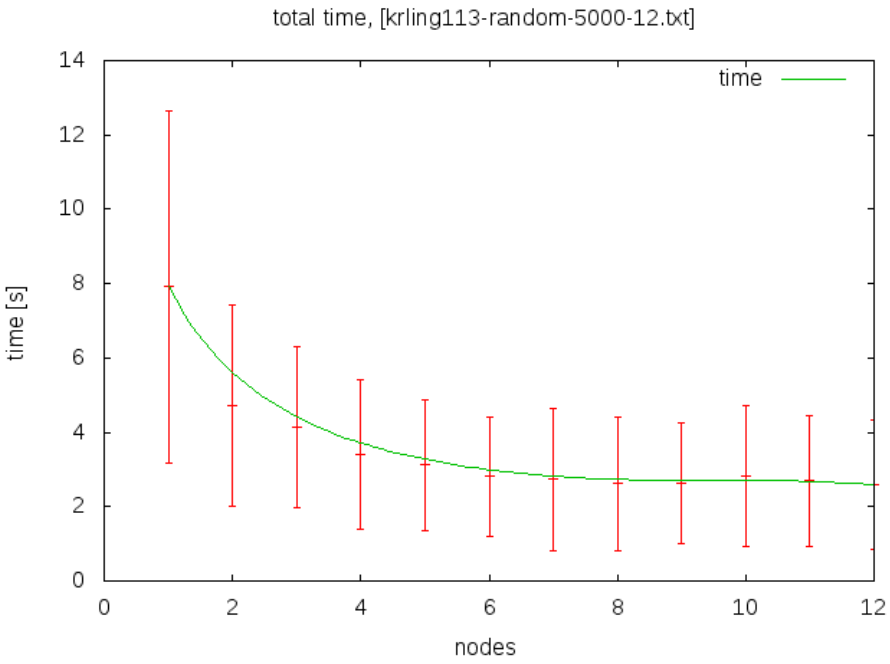

Figure 1: Czas wykonania programu - losowy graf o 5000 wierzchołkach (logNormalGraph)

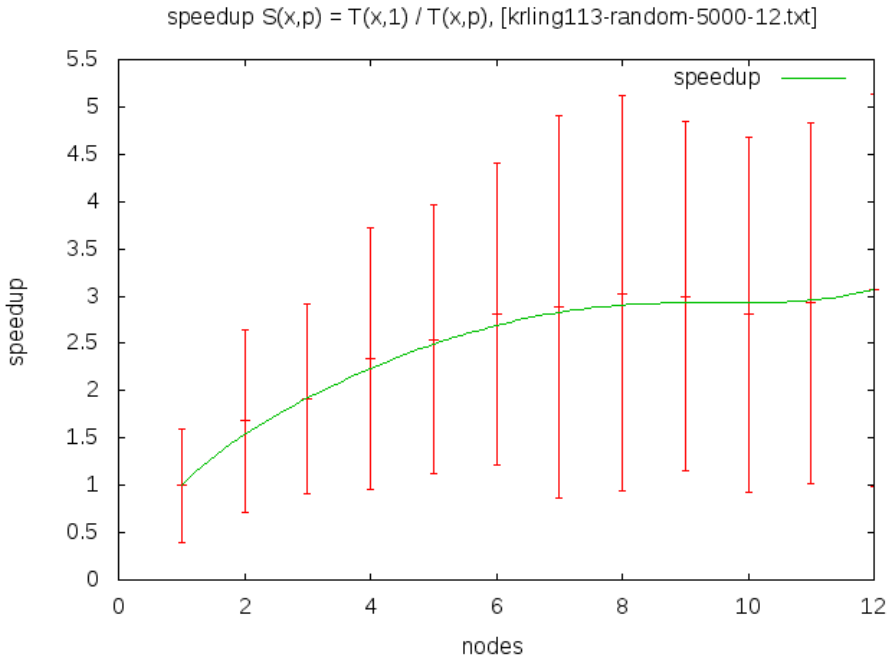

Figure 2: Przyspieszenie programu - losowy graf o 5000 wierzchołkach (logNormalGraph)

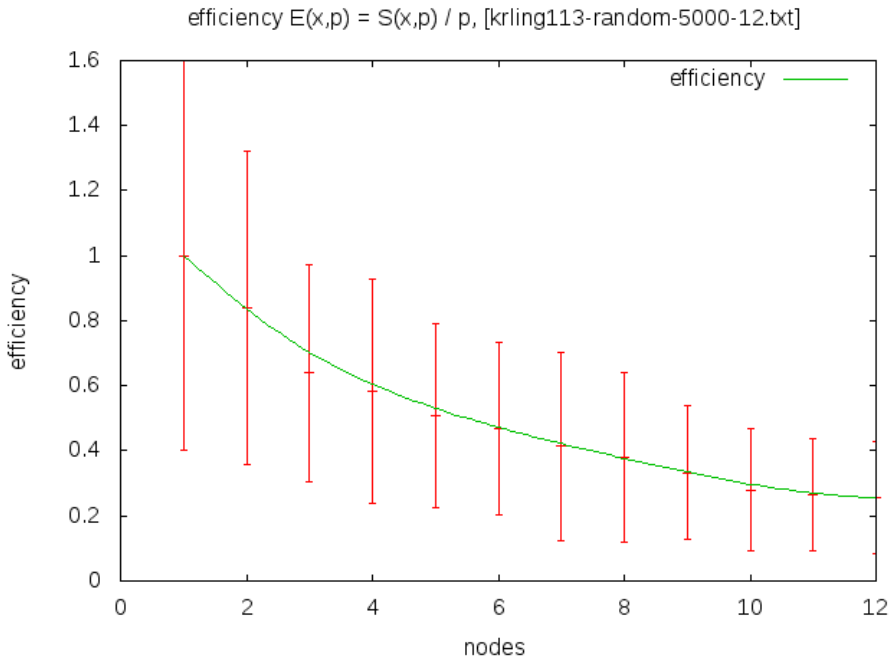

Figure 3: Efektywność programu - losowy graf o 5000 wierzchołkach (logNormalGraph)

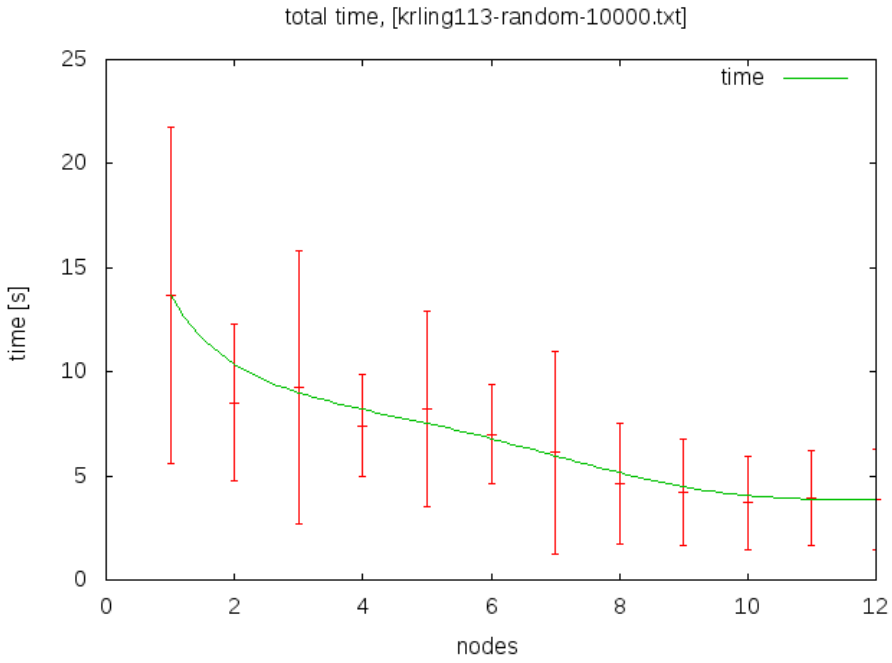

Figure 4: Czas wykonania programu - losowy graf o 10000 wierzchołkach (logNormalGraph)

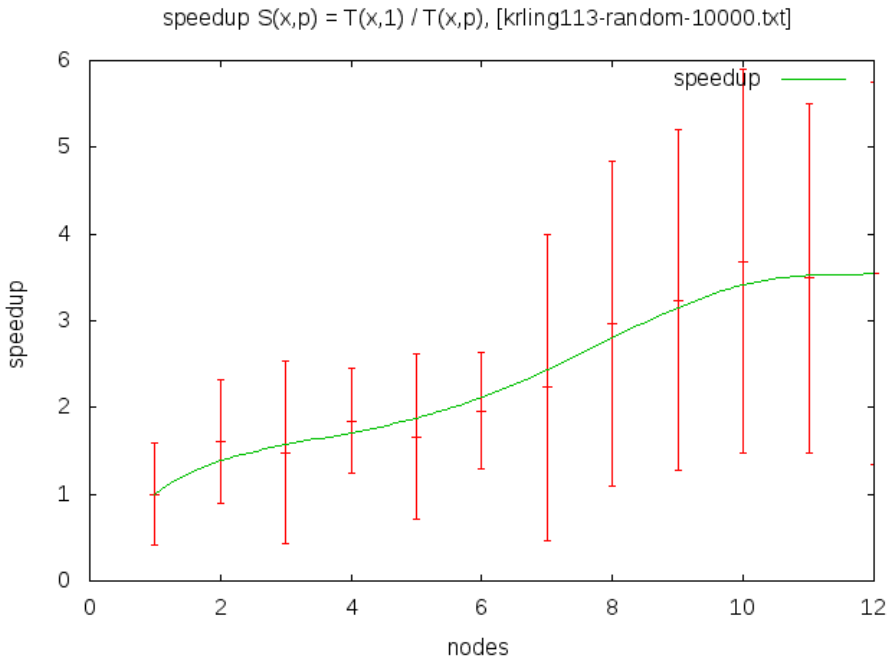

Figure 5: Przyspieszenie programu - losowy graf o 10000 wierzchołkach (logNormalGraph)

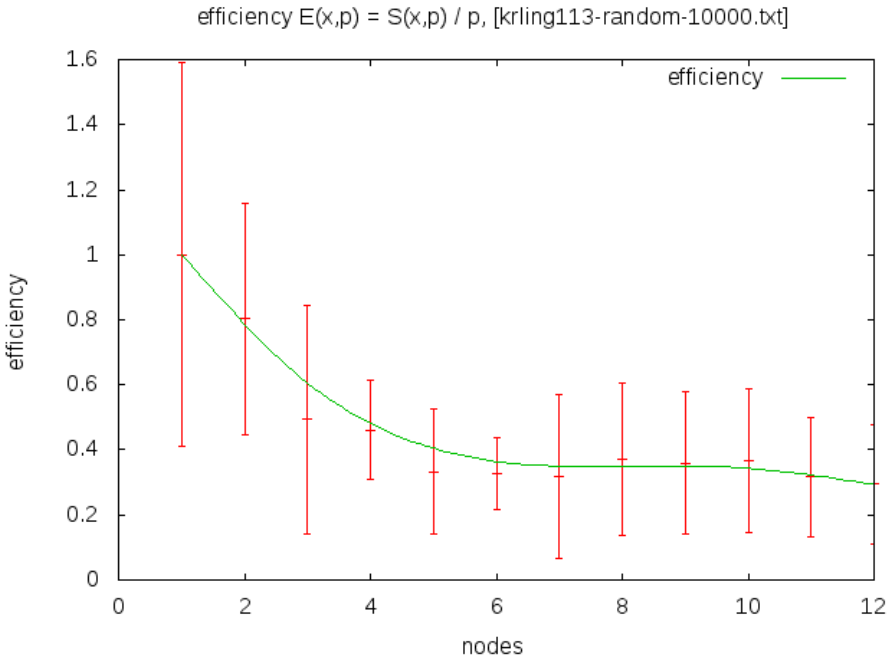

Figure 6: Efektywność programu - losowy graf o 10000 wierzchołkach (logNormalGraph)

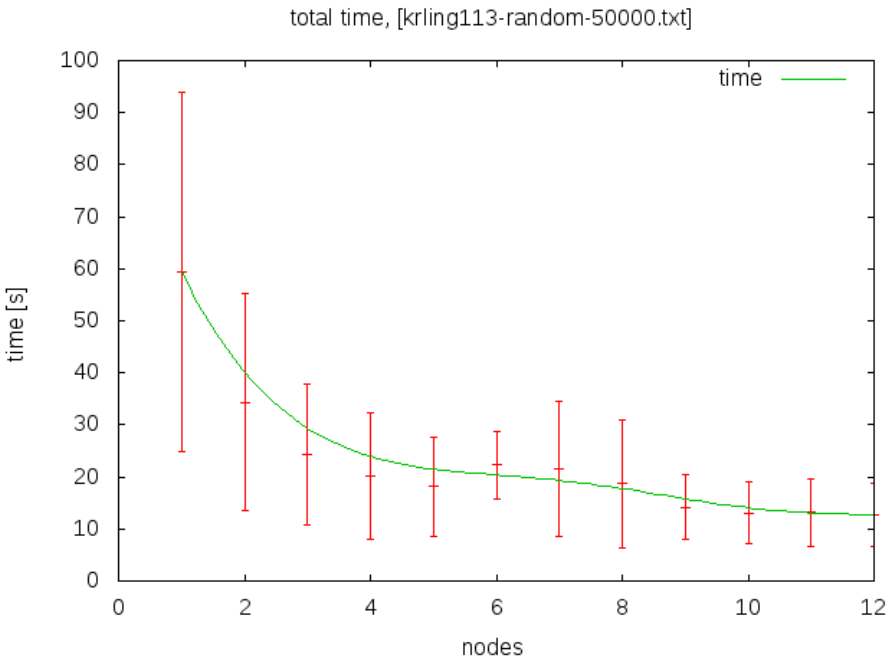

Figure 7: Czas wykonania programu - losowy graf o 50000 wierzchołkach (logNormalGraph)

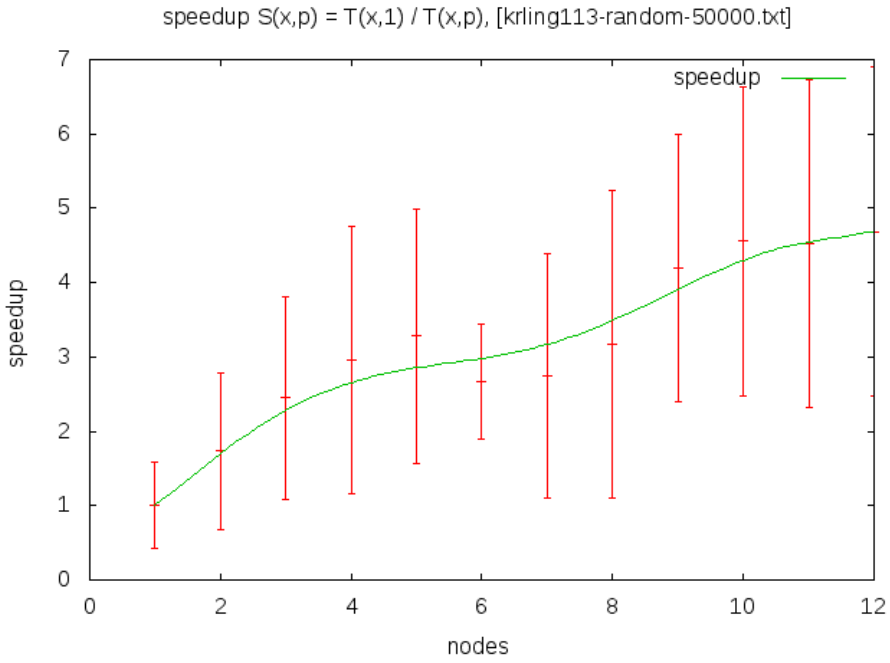

Figure 8: Przyspieszenie programu - losowy graf o 50000 wierzchołkach (logNormalGraph)

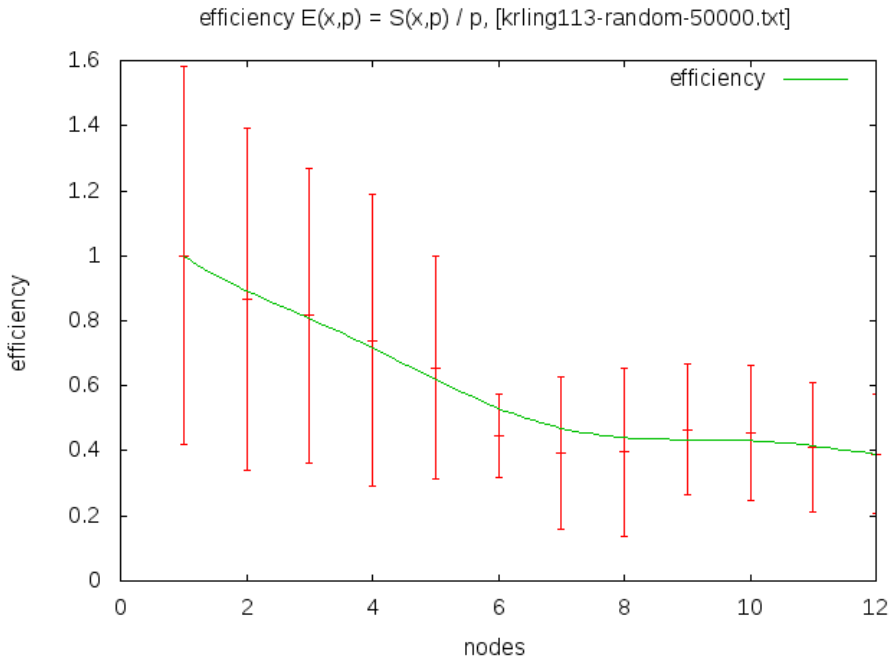

Figure 9: Efektywność programu - losowy graf o 50000 wierzchołkach (logNormalGraph)

Wyniki dla rzeczywistego grafu okazały się znacznie gorsze.

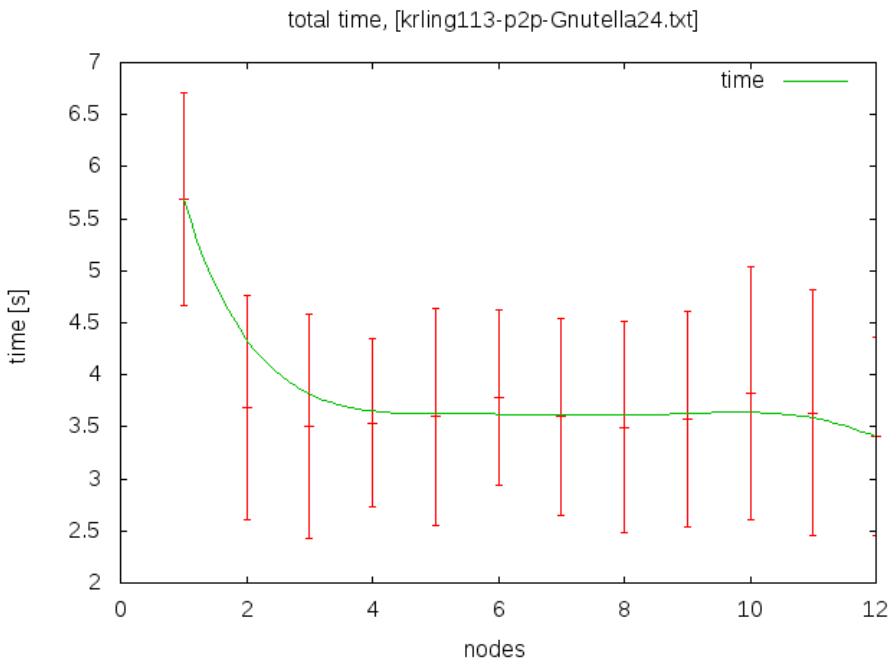

Figure 10: Czas wykonania programu - graf p2p-Gnutella24

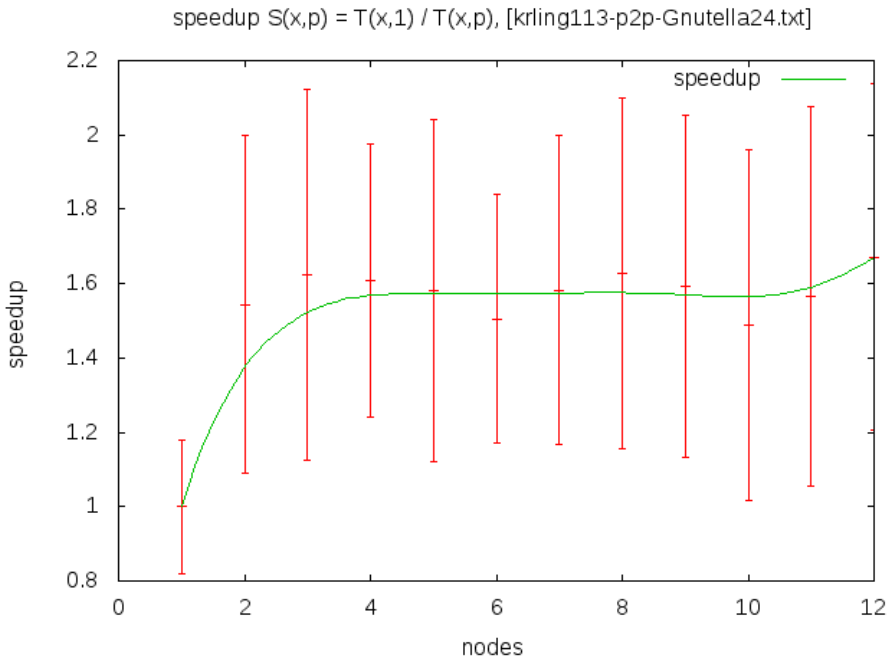

Figure 11: Czas wykonania programu - graf p2p-Gnutella24

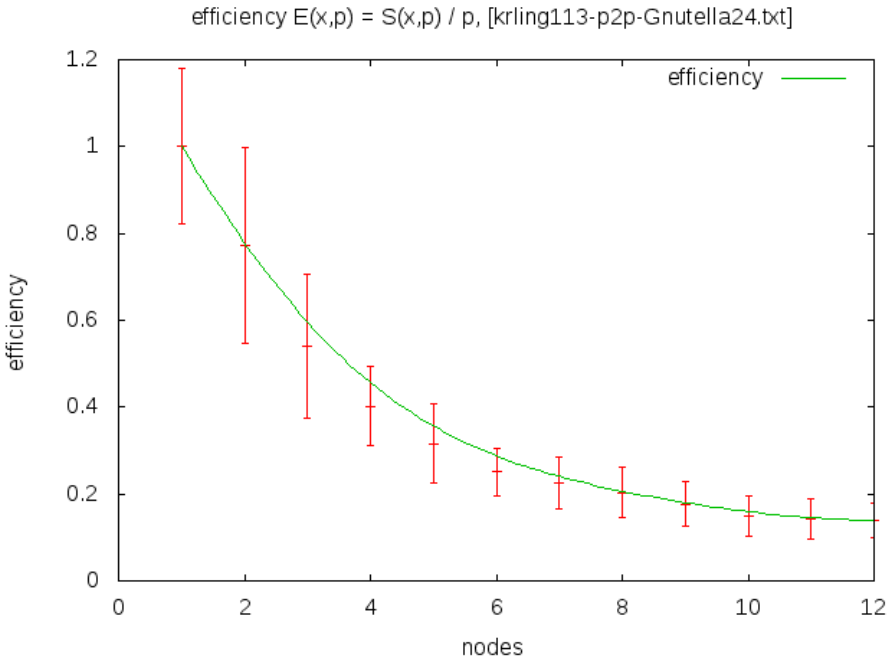

Figure 12: Czas wykonania programu - graf p2p-Gnutella24

5 Podział danych zgodnie z PCAM

Rozważanie platformy Apache Spark w kontekście metodologii PCAM niesie ze sobą pewne ograniczenia, wynikające z braku pełnej kontroli nad podziałem zadań i komunikacją.

5.1 Partitioning

Optymalny (ze względu na komunikację) przydział wierzchołków do procesorów to przydzielenie wszytkich połączonych wierzchołków o jednego procesora.

Jest to równoważne problemowi znalezienia spójnych składowych, który próbujemy rozwiązać. Przy reprezentacji grafu pozostaje wykorzystanie standardowego partycjonowania.

5.2 Komunikacja

W pętli algorytmu wielokrotnie wykonywane są złączenia (*join*) zbiorów o mocy równej liczbie wierzchołków w grafie. Może to stanowić wąskie gardło w konfiguracji gdzie program jest uruchomiony na klastrze zbudowanym z maszyn komunikujących się przez sieć.

5.3 Agglomeration

Uwzględniając komunikację, można dokonać ponownego podziału danych, z wykorzystaniem mechanizmu `Partitioner`, udostępnionego przez Apache Spark.

Rozwiązanie analogiczne do przedstawionego w [3] zakłada podział zbiorów na podstawie funkcji haszującej zastosowanej dla kluczy (indeksów wierzchołków).

```
val connections = graph.groupByKey
    .partitionBy(new HashPartitioner(4))
    .cache
```

Niestety w lokalnej konfiguracji nie zaobserwowałem poprawy wydajności.

5.4 Mapping

Przydziałem zadań do procesorów zajmuje się platforma Spark. Użytkownik nie kontroluje w bezpośredni sposób tego procesu.

References

[1] Zadeh, R., *Distributed Algorithms and Optimizations*, http://stanford.edu/~rezab/dao/notes/lec8.pdf, 2008.

[2] Foster, I., *Designing and Building Parallel Programs*, www.mcs.anl.gov/~itf/dbpp/.

[3] Zaharia, M., *Advanced Spark Features*, http://ampcamp.berkeley.edu/wp-content/uploads/2012/06/matei-zaharia-amp-camp-2012-advanced-spark.pdf, 2012.