Algorytmy równoległe 2015 (zad. 2)

Michał Liszcz

2015-11-10

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
```

TODO: rysunek grafu

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.

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 E5-2697 v2 [2] (12C/24T, łącznie 48 logicznych procesorów).

```
$lscpu
```

UWAGA: Na potrzeby testów konfigurowałem **lokalną instalację** Apache Spark (`-master local[X]`).

4.1 Graf testowy

Do testów wydajności wykorzystałem graf *web-Stanford* [3].

Na graf testowy składało się 281'903 wierzchołków i 2'312'497 krawędzi.

4.2 Losowo generowane grafy

Drugi wariant zakładał testy na losowo wygenerowanym grafie. Wykorzystałem obiekt `GraphGenerators`. [4]

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.

[1] https://snap.stanford.edu/data/ca-GrQc.html
[2] http://ark.intel.com/products/75283/Intel-Xeon-Processor-E5-2697-v2-30M-Cache-2_70-GHz
[3] https://snap.stanford.edu/data/web-Stanford.html
[4] http://spark.apache.org/docs/latest/api/scala/#org.apache.spark.graphx.util.GraphGenerators$

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.

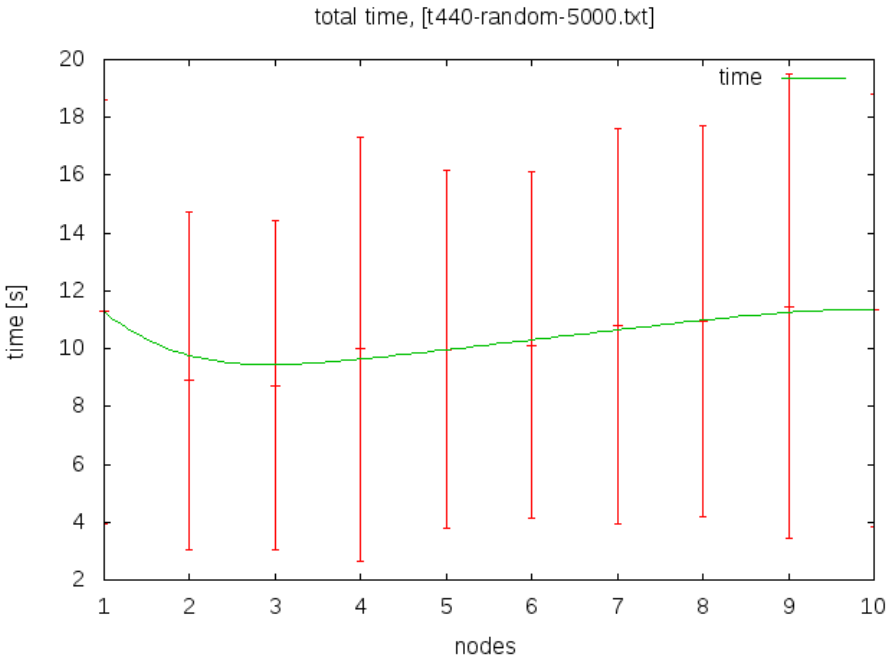

Figure 1: Czas wykonania programu - losowy graf o 5000 wierzchołkach (logNormalGraph)

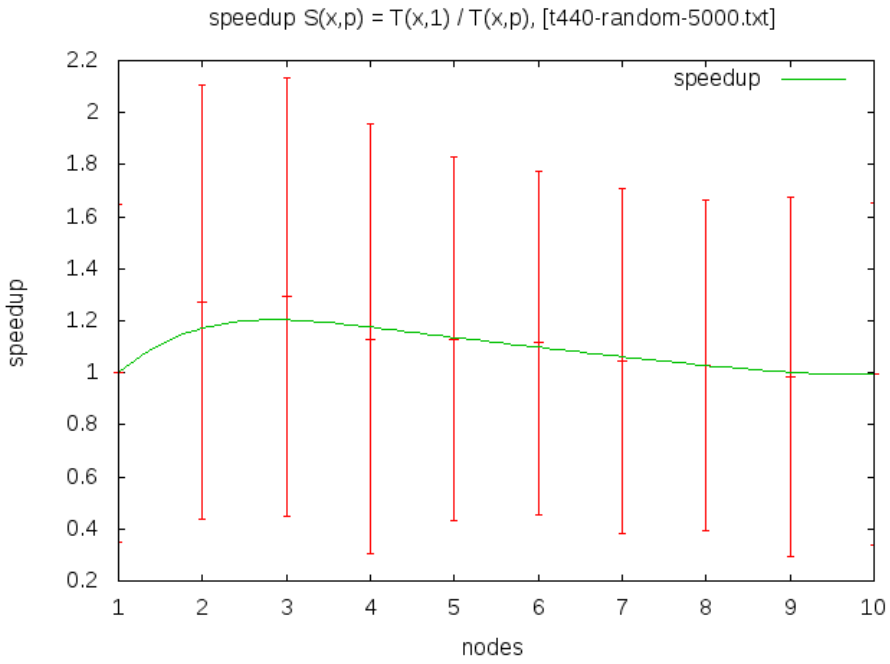

Figure 2: Przyspieszenie programu - losowy graf o 5000 wierzchołkach (logNormalGraph)

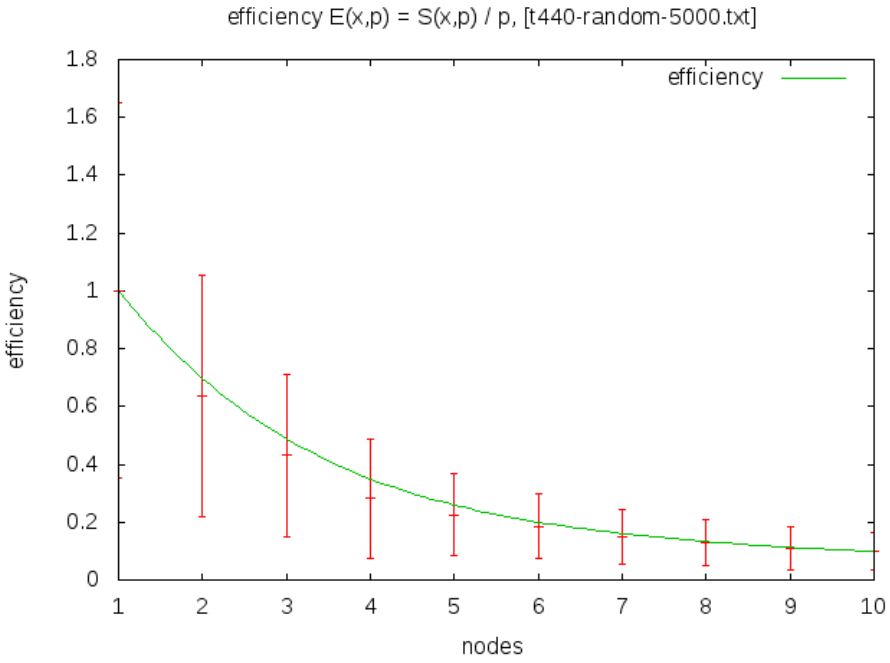

Figure 3: Efektywność programu - losowy graf o 5000 wierzchołkach (logNormalGraph)

5 Podział danych zgodnie z PCAM

6 Dyskusja wyników

TODO

References

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

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