Algorytmy równoległe – sprawozdanie Connected Components

Grzegorz Miejski

[NOTE: przepraszam za błąd - wszystkie czasy podane w tym sprawozdaniu są w milisekundach, a nie w sekundach jak sugerują rysunki i tabelki.]

1. Opis problemu:
   Problem polega na znalezieniu wspólnych składowych w grafie zgodnie z modelem Pregel. Model ten polega na ciągłym przesyłaniu komunikatów pomiędzy sąsiednimi wierzchołkami w grafie aż do zajścia warunku końcowego.
   W przypadku spójnych składowych algorytm wygląda następująco:
   - każdy wierzchołek zaczyna z **unikalną** przypisaną do siebie wartością - x
   - każdej iteracji każdy wierzchołek wysyła do swoich sąsiadów swoją wartość x
   - każdy wierzchołek bierze minimum z otrzymanych + swojej wartości i przypisuje sobie nową wartość
   - algorytm kończy się gdy żaden wierzchołek nie zmieni swojej wartości x
   - wierzchołki mające tę samą wartość x należą do wspólnej skłądowej

2. Podejście pierwsze:
   Podejście pierwsze było najbardziej naiwną implementacją. Tak jak i kolejna przedstawiona w późniejszym punkcie, wykorzystuje Spark'a.

   Wykorzystane oznaczenia:
   - edges - RDD[(VertexId, VertexId)] - każda krawędź jest skierowana, więc krawędzie muszą być zduplikowane co do kolejności wierzchołków w tuplu
   - vertexes - RDD[(VertexId, Long)] - każdy wierzchołek wraz z jego aktualną wartością

   Każda iteracja składała się z następujących kroków:
   - z połączenia edges i vertexes otrzymujemy informację do jakiego wierzhołka trafiają jakie wartości z sąsiednich nodów
   - dla każdego wierzchołka mapujemy łączymy otrzymane wartości z jego własną i bierzemy minimum
   - następuje sprawdzenie - łączymy nowe wartości wierzchołków ze starymi, filtrujemy te wierzchołki dla których wartość się zmieniła. Jeżeli jest to zbiór pusty, kończymy algorytm, jeżeli nie, kontynuujemy ze zmienną vertexes posiadającą nowe wartości

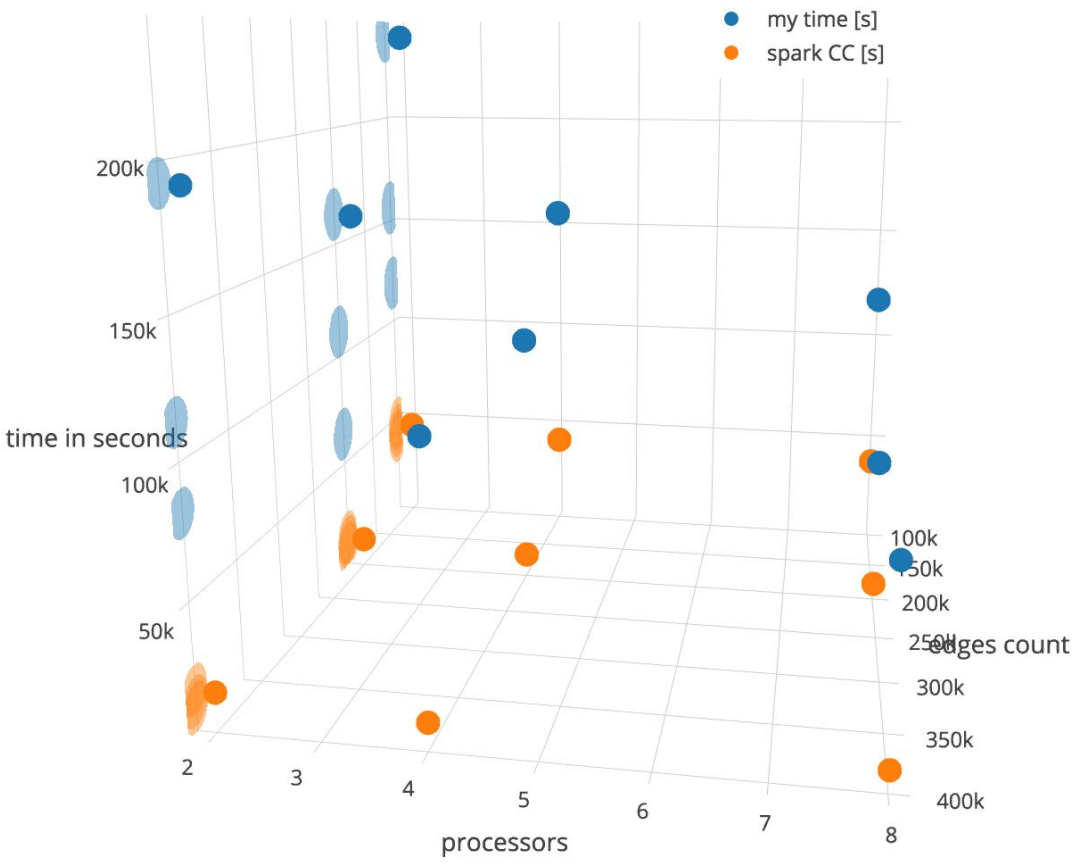
Jak widać pierwsza implementacja była bardzo prosta i naiwna - wykonywaliśmy bardzo wiele joinów nie tylko w celu obliczeń, ale także w celu sprawdzenia warunku końca algorytmu.

Każdy graf był generowany na podstawie 2 argumentów - ilość wierzchołków oraz ilość krawędzi. Następnie uruchamiałem program wykorzystując jedną z 3 wartości ilości procesorów - 2, 4 albo 8.
Dla porównania efektywności algorytmu na tych samych wartościach liczyłem również czas zbudowanej funkcji Spark Grapx która także rozwiązuje problem connected components za pomocą modelu pregel.

Poniżej przedstawiam wyniki tej części zadania. Wszystkie rezultaty zostały otrzymane przez uruchomienie na Zeusie, ale wykorzystując tylko jednego node'a, z programem uruchamianym z wykorzystaniem różnej ilości procesorów.

vertexes	edges	processors	my time [s]	spark CC [s]
1000000	100000	2	237883	46167
1000000	100000	4	155767	42294
1000000	100000	8	118440	39420
1000000	200000	2	162607	14258
1000000	200000	4	110432	12005
1000000	200000	8	63405	8924
1000000	400000	2	192570	17102
1000000	400000	4	113566	12756
1000000	400000	8	81761	10308
2000000	100000	2	84709	5299
2000000	100000	4	36183	3435
2000000	100000	8	21981	2771
2000000	200000	2	422296	44325
2000000	200000	4	297595	42345
2000000	200000	8	201667	51167
2000000	400000	2	396359	27441
2000000	400000	4	207642	19425
2000000	400000	8	144739	15397
4000000	100000	2	39191	2948
4000000	100000	4	25932	2328
4000000	100000	8	19839	1993
4000000	200000	2	175918	9560
4000000	200000	4	89113	5448
4000000	200000	8	80536	7514
4000000	400000	2	924859	58144
4000000	400000	4	NA	NA
4000000	400000	8	NA	NA

Przykład wizualizacji obrazującej skalowalność tego rozwiązania:



Jest to wykres dla liczby wierzchołków równej 1000000.
Niebieskie kropki przedstawiają czas mojego rozwiązania, pomarańczowe - wbudowaną funkcję ze Sparka.
Wyblakłe punkty na lewej ścianie sześcianu pokazują rzut czasu wykonania dla danych argumentów. Dzięki temu łatwo zauważyć jak redukuje się czas dla danych argumentów - pionowe linie tworzone przez projekcje są to czasy wykonania dla zwiększającej się liczby wykorzystanych procesorów.

Można wysnuć kilka wniosków z tabelki powyżej oraz przedstawionego rysunku:

- Naiwny algorytm jest nawet 10 krotnie wolniejszy od oryginalnego rozwiązania Sparka
- Naiwny algorytm skaluje się dosyć dobrze
- Dużo większe koszty obliczeniowe wnosi zwiększenie liczby krawędzi niż dodanie nowych wierzchołków

3. Podejście drugie
   Przy drugim podejściu postanowiłem wykorzystać funkcje Sparka, która umożliwia uruchamianie algorytmów za pomocą modelu pregel - metoda pregel w obiekcie grafu Spark'a.

   W tym przypadku uruchamiałem przykłady na Zeusie w ustawieniu :
   - liczba nodów - 3
   - ppn - 12
   - wykorzystanie wszystkich procesorów dostępnych na maszynach

vertexes	edges	processors	my time [s]	spark CC [s]
1000000	100000	*	57799	49424
1000000	100000	*	144528	136914
1000000	100000	*	42974	32598
1000000	200000	*	26705	15344
1000000	200000	*	26286	17009
1000000	200000	*	27243	17298
1000000	400000	*	28717	17577
1000000	400000	*	28957	19094
1000000	400000	*	30319	20645
2000000	100000	*	14052	4823
2000000	100000	*	14228	4820
2000000	100000	*	16145	6632
2000000	200000	*	56662	45642
2000000	200000	*	65979	55702
2000000	200000	*	67335	56470
2000000	400000	*	40916	29672
2000000	400000	*	41614	27819
2000000	400000	*	38761	27695
4000000	100000	*	11169	3002
4000000	100000	*	11452	2967
4000000	100000	*	10725	3333
4000000	200000	*	17462	7713
4000000	200000	*	23255	13315
4000000	200000	*	19013	8752
4000000	400000	*	87489	74775
4000000	400000	*	69911	58307
4000000	400000	*	69917	57477

Wnioski jakie można wyciągnąć:

- liczba krawędzi nie zmienia już diametralnie czasu obliczeń
- ciągle ręczna implementacja connected components jest w każdym przypadku wolniejsza od gotowego algorytmu CC w Sparku.
  Zapewne wynika to z wykorzystanych optymalizacji w implementacji.

4. Podejście trzecie
   Chcąc zbadać skalowalność drugiego podejścia, chciałem uruchomić na Zeusie to samo zadanie dla liczby nodów równej 6. Niestety po półtora tygodnia oczekiwań na uzyskanie zasobów postanowiłem zastopować zadanie w kolejce qsub. Następnie spróbowałem ponownie, lecz po kolejnym tygodniu dalej czekam na dostęp do zasobów, dlatego publikuję sprawozdanie bez tego porównania.

5. Uruchomienie na gotowym zbiorze danych
   Podczas zadania wykorzystałem również swoje implementacje na gotowym zbiorze danych: https://snap.stanford.edu/data/egonets-Facebook.html

   Tutaj przebadałem rozwiązanie na swoim lokalnym komputerze z maksymalną ilością procesorów.
   Całość zajęła tylko 9 iteracji algorytmu a rozwiązanie zajęło około 11 sekund.

6. Wnioski
   Dzięki tym laboratoriom można było empirycznie potwierdzić to, o czym mówią ludzie wykorzystujący/tworzący sparka. W przeciwieństwie do standartowego map-reduce'a idealnie nadaje się do zadań iteracyjnych. Do tego łatwość pisania swoich rozwiązań jest bardzo atrakcyjna. Największym minusem Sparka jest to, że pomimo łatwego poziomu wejścia w technologię, aby móc praktycznie i efektywnie wykorzystywać tę techonologię w pracy/na codzień, trzeba poznać bardzo wiele szczegółów implementacyjnych.

7. Linki
   - https://github.com/gmiejski/ar_spark_connected_components

Grzegorz Miejski