Algorytm b-adoratorów

Piotr Ambroszczyk

***Streszczenie*—Opis implementacji oraz analiza wydajności algorytmu b-adoratorów.**

I. Wstęp

W tym raporcie opiszę szczegółowo moją implementację algorytmu b-adoratorów[1]. Przedstawię zastosowane optymalizacje oraz analizę zależności czasu wykonania algorytmu od liczby dostępnych wątków.

II. Implementacja

W celu implementacji równoległej wersji algorytmu atomiców.

W pierwszym kroku wczytuję graf, nadając kolejnym wierzchołkom nowe indeksy, w celu łatwiejszej iteracji po grafie. W tym celu użyłem mapy, natomiast stare identyfikatory trzymam w wektorze `old_indices` dzięki czemu mam do nich dostęp w czasie stałym. Sam `graph` jest wektorem wektorów.

Następnie algorytm współbieżnie sortuje krawędzie każdego wierzchołka.

Po posortowaniu inicjalizujemy potrzebne struktury takie jak wektor kolejek priorytetowych - `suitors`, wektor wskaźników na atomowe boole - `s_vertex`, wektory trzymające informacje ile możemy lub mogliśmy mieć maksymalnie adoratorów - `b`, `bd` oraz `primary_b`, wektor którego sąsiada powinniśmy wybrać - `t_next`, kolejki wierzchołków - `v_queue` oraz `next_v_queue`, wektor atomiców mówiących jaka jest najsłabsza propozycja - `last`, atomic boole synchronizujące kolejki - `s_v_queue` oraz `s_v_next_queue`.

Po inicjalizacji uruchamiamy nasz algorytm kolejno dla każdego numeru metody z przedziału $[0, \ldots, \texttt{blimit}]$. Każdy wątek, włącznie z głównym wątkiem programu, dostaje z kolejki wierzchołek do obsłużenia. Następnie iterując się monotonicznie po jego sąsiadach, zaczynając od tego do którego prowadzi największa krawędź, sprawdza za pomocą tablicy `last` czy jego propozycja jest dostatecznie dobra. Jeżeli rzeczywiście jest, to zakładając spinlocka ponownie sprawdza czy ma dobrą propozycję, jeśli tak oferuje się swojemu sąsiadowi.

Dystrybucja wierzchołków zaimplementowana została z użyciem metody `next_vertex_from_v_queue()` która w sposób atomowy zwraca kolejny wierzchołek.

Dla każdej metody wypisujemy wynik, oraz usuwamy zawartość struktur pomocniczych. Obliczenie opiera się na własności algorytmu, że $u \in S(v) \iff v \in S(u)$.

III. Optymalizacje

Najistotniejszą optymalizacją jest początkowe, współbieżne posortowanie krawędzi wychodzących z każdego wierzchołka. Wykonujemy to tylko raz, a na posortowanym grafie operacje przeprowadzane są dużo szybciej. Dystrybucja wątków w funkcji `sort_vertex_edges` nie korzysta z wolnych mutexów lecz atomiców.

Kolejną optymalizacje daje obserwacja, że po posortowaniu krawędzi nie potrzebujemy trzymać tablicy użytych krawędzi, tak jak to było przedstawione w artykule [1]. Wynika to z faktu, że jeżeli ktoś odrzucił naszą propozycję, to już nigdy więcej jej nie przyjmie. To który wierzchołek powinniśmy wybrać mówi nam wektor `t_next`. W jednym momencie dwa wątki nie mogą obsługiwać tego samego wierzchołka, dlatego `t_next` jest *zwykłym* wektorem i nie musiałem synchronizować do niego dostępu za pomocą semaforów ani atomiców.

Problematyczne okazało się dodawanie wierzchołka do `next_v_queue` dokładnie raz. Tę zagwozdkę, oraz zapewnienie aktualizacji wektora `bd` rozwiązałem implementując `bd` jako wektor atomiców. Skorzystałem tutaj z atomowej operacji $++$ która zwraca kopie obiektu, nie zaś referencję do niego. Tak jak poprzednio, porzucenie mutexów i użycie atomiców miało pozytywny wpływ na osiągi programu.

W poprzednich wersjach nie używałem tablicy `last`, tylko dla każdego sąsiada danego wierzchołka wątek zakładał zamek, a dopiero potem sprawdzał czy ma coś do zaoferowania. Dodanie tej tablicy i sprawdzanie minimalnej oferty przed założeniem zamka znacząco wpłynęło na osiągi programu.

Początkowo mój algorytm wykonywał obliczanie wyniku współbieżnie. Każdy wątek pobierał wierzchołki, obliczał sumę wag krawędzi wychodzących do wierzchołków z kolejki `suitors`, a następnie za pomocą mechanizmu future - promise przekazywał swój wynik do głównego wątku programu. Po przeprowadzeniu testów okazało się, że synchronizacja wątków zabiera zdecydowanie więcej czasu niż obliczenie wyniku przez jeden wątek, dlatego postanowiłem wrócić do wersji sekwencyjnej.

W pierwszych wersjach programu usunąłem jedynie flagi debugowania z pliku CMAKE. Nie skutkowało to jednak znaczącym wzrostem szybkości programu. Dopiero dodanie flagi `-o3` oraz ostawienie `-DCMAKE_BUILD_TYPE=Releas` sprawiło, że program przyśpieszył około 4 razy.

Z początku do ochrony sekcji krytycznej nie używałem spinlocków lecz mutexów. Jako, że spinlocki są efektywne w przypadku małych sekcji krytycznych postanowiłem użyć ich do synchronizacji dostępu wątków do kolejek wierzchołków. Po analizie czasu systemowego, stwierdziłem że liniowa ilość mutexów może stanowić w tym wypadku problem, dlatego zamieniłem je na spinlocki. Po modyfikacjach czasy działania algorytmu były porównywalne.

IV. Wydajność

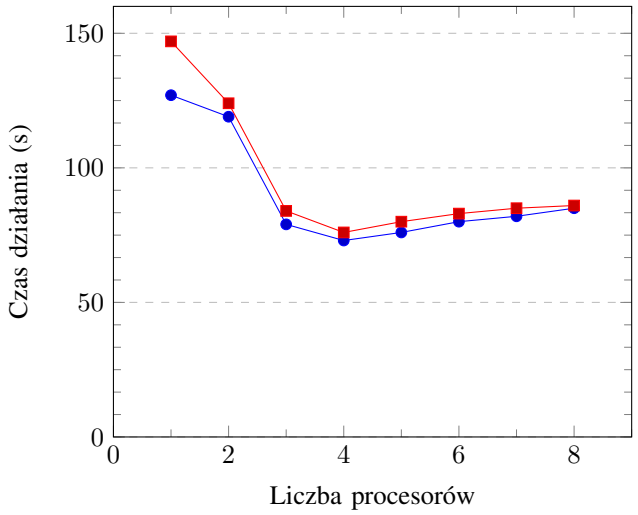
Poniższe dane otrzymałem testując program na mapie drogowej Pensylwanii [2]. Obliczenia wykonałem na komputerze ASUS ZENBOOK UX305F z procesorem INTEL CORE M-5Y10. Program uruchomiłem 11 razy dla $\texttt{blimit} = 100$.

Przyśpieszenie liczone jest na podstawie średniego czasu działania programu.

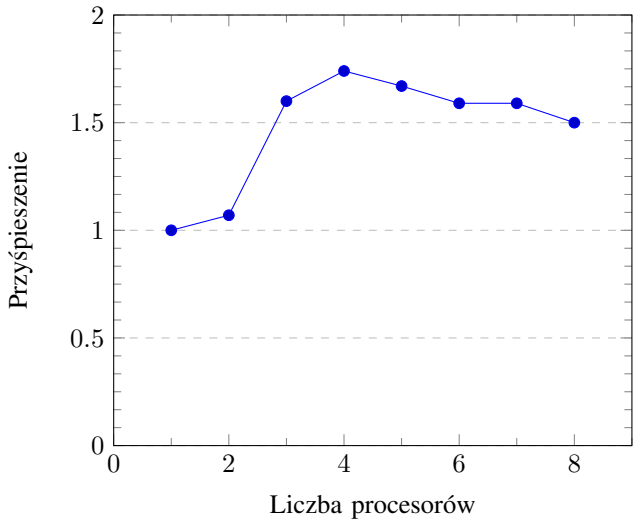
Dzięki dużemu `blimit` mamy pewność, że wczytywanie grafu nie zdominuje czasu działania algorytmu.

Aby testy były jak najbardziej wiarygodne zmodyfikowałem lekko program sprawiając, żeby wierzchołki w kolejce `v_queue` nie były zawsze w kolejności $0, 1, 2 \ldots$, lecz w losowej. Użyłem do tego funkcji `std::random_shuffle`.

Liczba wątków	Średni czas	Maksymalny czas	Przyśpieszenie
1	127s	147s	1
2	119s	124s	1.07
3	79s	84s	1.6
4	73s	76s	1.74
5	76s	80s	1.67
6	80s	83s	1.59
7	82s	85s	1.59
8	85s	86s	1.5



Rysunek 1. Czas działania programu w zależności od liczby rdzeni.



Rysunek 2. Przyśpieszenie programu w zależności od liczby rdzeni.

Widzimy, że algorytm osiąga maksymalną efektywności przy użyciu 4 wątków, użycie większej ilości wątków jedynie ją zmniejsza. Czasy wykonania algorytmu na jednym rdzeniu i na dwóch rdzeniach nie różnią się znacząco. Prawdopodobną przyczyną wyżej wymienionych zachowań jest poświęcanie dużej ilości czasu na synchronizacją procesów.

Przy testach dla dużej liczby wątków – rzędu 90 – algorytm jest wolniejszy niż uruchamiany na czterech, jednak cały czas szybszy niż implementacja sekwencyjna.

V. Podsumowanie

Zaproponowana przeze mnie implementacja algorytmu b-adoratorów działa efektywnie. Załączone wykresy pokazują, że szybkość programu bezpośrednio zależy od liczby użytych wątków. Algorytm osiąga maksymalną efektywność, gdy liczba dostępnych wątków jest równa liczbie dostępnych rdzeni procesora.

Literatura

[1] A. Khan and others. Efficient approximation algorithms for weighted b-maching. Mar. 2016.
[2] http://snap.stanford.edu/data/roadNet-PA.html