Zakres projektu

Projekt dotyczy implementacji algorytmu PrefixSpan w języku C++, opisanego w dokumencie:
Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach
Jian Pei, Member, IEEE Computer Society, Jiawei Han, Senior Member, IEEE,
Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen,
Umeshwar Dayal, Member, IEEE Computer Society, and Mei-Chun Hsu

Ponadto zostanie do niego dodana modyfikacja umożliwiająca wykonanie go wielowątkowo. Dzięki temu kosztem dodatkowej pamięci, obliczenia powinny zostać w teorii przyśpieszone.

Algorytm w obydwu wersjach zostanie zbadany pod kątem wydajności czasowej oraz pamięciowej. Język python zostanie użyty do stworzenia prostych skryptów, między innymi do porówania dwóch plików z transakcjami.

Implementacja

Idea algorytmu PrefixSpan polega na rekurencyjnym dodawaniu prefiksu. Prefiks jest definiowany jako sekwencja elementów, która jest rozwijana od lewej do prawej strony wyrazu.
W każdej rekursji rozpatrywana jest część bazy, która zawiera rozpatrywany prefix (na samym początku prefix ma zerową długość). Dla każdej transakcji dla każdej rekursji, zostaje zapamiętana pozycja, od której prefix się kończy. Algorytm sprawdza, czy dla danego rozpatrywanego elementu występuje ona na prawo od prefiksa w danej rozpatrywanej transakcji. Jeżeli tak, to zostaje ona dodana do nowej bazy transakcji wraz z pozycją końca nowego prefiksa.
W celu optymalizacji pamięci, przechowuje się w projekcji bazy wskazania na dane transakcje.

Szczegóły implemntacyjne

Program został napisany w taki sposób, aby umożliwić wykonanie go w wersji jedno oraz wielowątkowej bez potrzeby ponownej jego kompilacji. Numerem 1 oznaczone zostanie wersja jednowątkowa, a numerem 2 wielowątkowa. Na początku działania algorytm:

1. Wywołuje funkcję prefixProjectImpl.
2. Tworzy thread pool wraz z wątkami, które oczekują na funkcje do wykonania. Następnie dodaje do niego pojedynczą pracę prefixProjectImpl, a na końcu czeka na zakończenie wszystkich wątków.

Następnie w funkcji prefixProjectImpl wykonuje typowe sprawdzenia zakończenia rekursji, dotyczące wielości bazy danych oraz prefiksu. W tym miejscu następuje też zapis do pliku prefiksu oraz innych statystyk. Następnie baza danych jest analizowana pod względem występowania danych w niej elementów. Wymaga to przejścia przez wszystkie transakcje w bazie. Na końcu dla każdego elementu, który nazywamy item:

1. Wywołuje funkcję prefixProjectImplNext.
2. Dodaje do puli zadań funkcję prefixProjectImplNext wraz z odpowiednimi argumentami.

Funkcja prefixProjectImplNext tworzy nową bazę danych, która będzie przekazana dalej w rekursji. Przechodzi ona przez wszystkie transakcje, rozpoczynając przeszukiwanie transakcji od poprzednio zapamiętanej dla niej pozycji (na początku jest to 0), a następnie sprawdza po prawej stronie transakcji, czy nie występuje element równy wartości item. Jeżeli tak, to transakcja zostaje dodana do nowej bazy danych wraz z nową pozycją znalezionego elementu. Po sprawdzeniu całej bazy danych, nowy prefiks wraz z dodanym elementem item jest przekazywany jako argument dalej::

1. Wywołuje funkcję prefixProjectImpl.

2. Dodaje do kolejki zadań funkcję prefixProjectImpl

W tym kroku przekazuje się nowy prefiks oraz nową bazę danych.

W celu optymalizacji nadano pierwszeństwo tym pracom, które są najgłębiej w rekursji, ponieważ one mają największą szansę na to, że zostaną zakończone, przez co zostanie zwolniona pamięć. Trzeba zaznaczyć, że dane pomiędzy obydwoma częściami algorytmu są przekazywane za pomocą kopii z wyjątkiem danych dotyczących bazy danych. Dane na początku działania algorytmu zostały wczytane przy pomocy struktury std::vector, przechowując je jako wartości liczbowe.

Pseudokod

```
database = load_data()
emptyPrefix = []
prefixProject(database, emptyPrefix)

def prefixProject(database, prefix)
   if(mthread):
      create_threads()
      add_job(prefixProjectImpl, database, prefix)
   else:
       prefixProjectImpl(database, prefix)

def prefixProjectImpl(database, prefix):
   itemSet = getItemSet()

   for item in itemSet:
      if(mthread):
         add_job(prefixProjectImplNext, database, prefix, item)
      else:
         second(database, prefix, item)


def prefixProjectImplNext(database, prefix, item):
   newDatabase
   for trans in database:
      check, newPos = exist(trans, prefix, item)
      if(check):
         newDatabase.add(trans, newPos)

   prefix += item
   if(mthread):
      add_job(prefixProjectImpl, newDatabase, prefix)
   else:
      first(database, prefix)
```

Założenia

- Dostarczone dane są w formie tabelarycznej, w której to każda linia reprezentuje jedną transakcję. Elementy w transakcji są oddzielone spacją.
- Kompilacja programu została wykonana z flagami c++17 -O3.
- Wyniki są zapisywane do plików w formacie txt, csv oraz, jeżeli użytkownik zechce, na standardowe wyjście.

- Czas podawany jest w nanosekundach. Różnica czasu jest podawana od wykonania pierwszego pomiaru. Pierwszy pomiar następuje przed wczytaniem danych do pamięci.
- Dane na początku zostają wczytane do pamięci w całości.
- Wykorzystana pamięć podawana jest w KB.
- W przeprowadzonych eksperymentach w trybie wielowątkowym wykorzystano 4 wątki.
- Program zwraca co najmniej 4 pliki dotyczące statystyk czasu oraz zużycia pamięci przez program wraz z plikiem zawierającym znalezione wzorce sekwencyjne, spełniające podane ograniczenia. Ostatni plik informuje o użytych flagach.

Słownik

Wyjaśnienie flag programu

- -ms <integer [1;], wymagany> – minimum support – minimalna ilość transakcji wspierających dany wzorzec sekwencyjny.
- -mp <integer [1;], wymagany> – maximum pattern lenght – maksymalna długość wzorca sekwencyjnego.
- -f <ścieżka do pliku, wymagany> – plik wejściowy zawierający transakcje.
- -o <ścieżka do folderu, wymagany> – folder wyjściowy, w którym program stworzy pliki wyjścia programu. Wszystkie foldery w ścieżce zostaną stworzone, jeżeli nie istnieją. Folder na końcu ścieżki nie może wcześniej istnieć.
- -v – wypisz wszystkie znalezione prefiksy na standardowe wyjście.
- -pt – zapisz numery transakcji, które zawierają znaleziony prefiks. Wraz z flagą -v wypisze to na standardowe wyjście.
- -r <integer [1;]> - liczba powtórzeń wykonania algorytmu PrefixSpan. Każde wywołanie posiada własną bazę transakcji w pamięci, struktury oraz wyjście. Domyślnie 1.
- -t – użyj implementacji wielowątkowej. Bez tej flagi inne opcje wielowątkowości nie będą brane pod uwagę.
- -thr <integer [1;]> - liczba wątków programu.

Eksperymenty

Zbadano takie kwestie programu w wersjach jedno oraz wielowątkowym jak:
- czas wykonania,
- zajętość pamięciowa całego programu,
- wpływ parametrów programu względem różnych zbiorów danych na zużycie pamięci oraz czasu.

Prawie każdy eksperyment został powtórzony co najmniej 3 razy w celu uzyskania średniej z wyników. Średnia ta będzie liczona jako włączenie do jednego pliku wszystkich wykonanych pomiarów, a następnie wykonanie średniej ruchomej. Przyjmuje się taki algorytm z powodu nieregularności w czasie wykonania oraz zapisu programu oraz nieregularności zapisu danych w trybie wielowątkowym. Ilość powtórzeń będzie uzależniona od długości trwania pojedynczego eksperymentu. Należy zwrócić uwagę, że powstały wykres nie pokazuje maksymalnego uzyskanego zużycia pamięci.

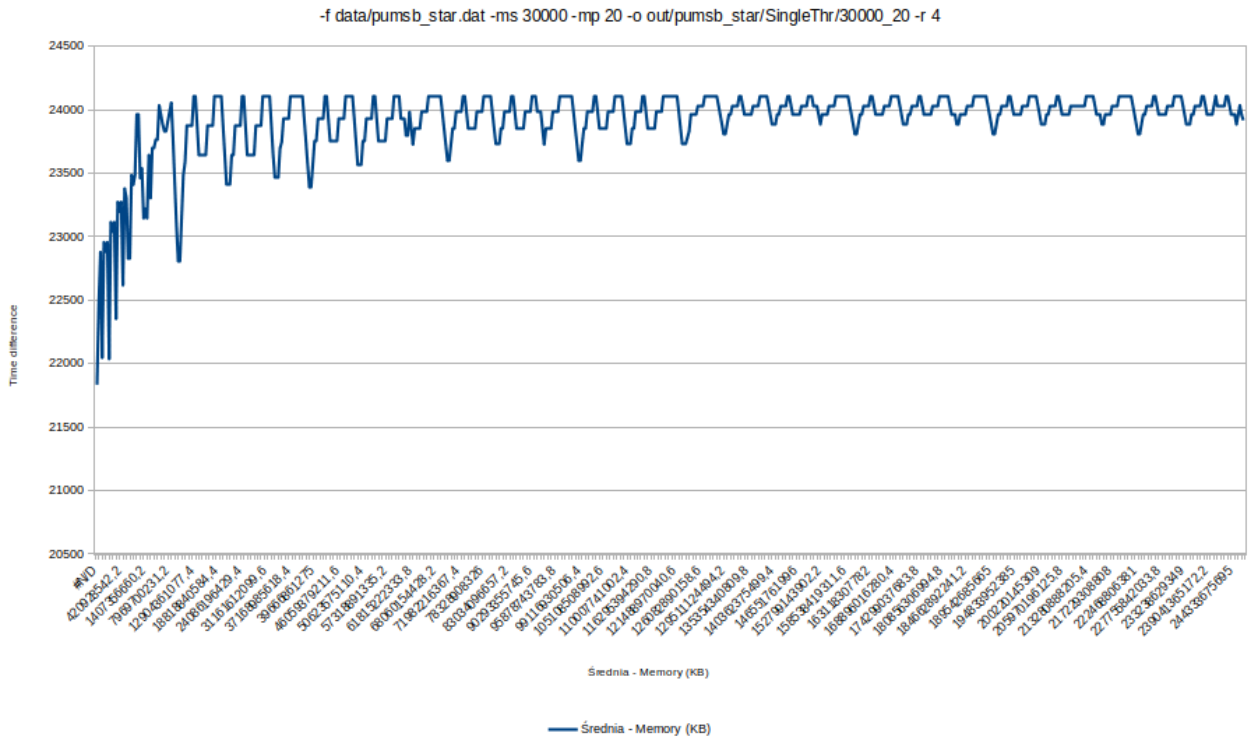
Zbiory danych, które zostały użyte podczas badań oraz testów znajdują się na stronie http://fimi.uantwerpen.be/data/ Planuje się wykorzystanie plików:

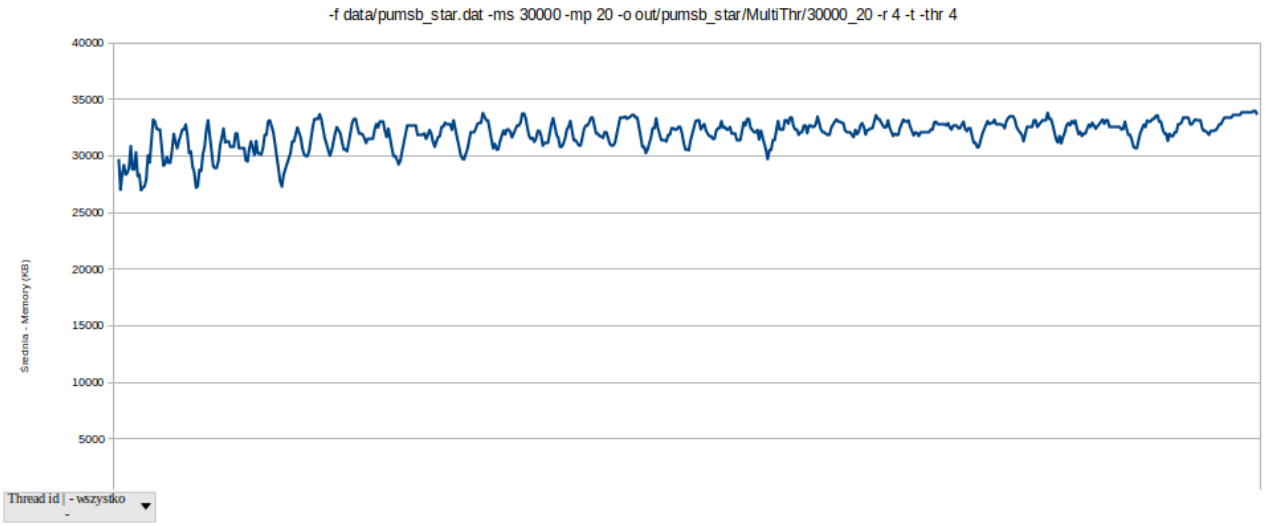
- mushroom.dat – mały zbiór danych, zawierający około 8100 transakcji, 570,4 KB
- pumsb_star.dat – średni zbiór danych, zawierający około 49000 transakcji, 11571 KB
- accidents.dat – duży zbiór danych, zawierający około 340000 transakcji, 36352 KB, wykonano go tylko jeden raz z powodu długości wykonania pojedynczego eksperymentu.

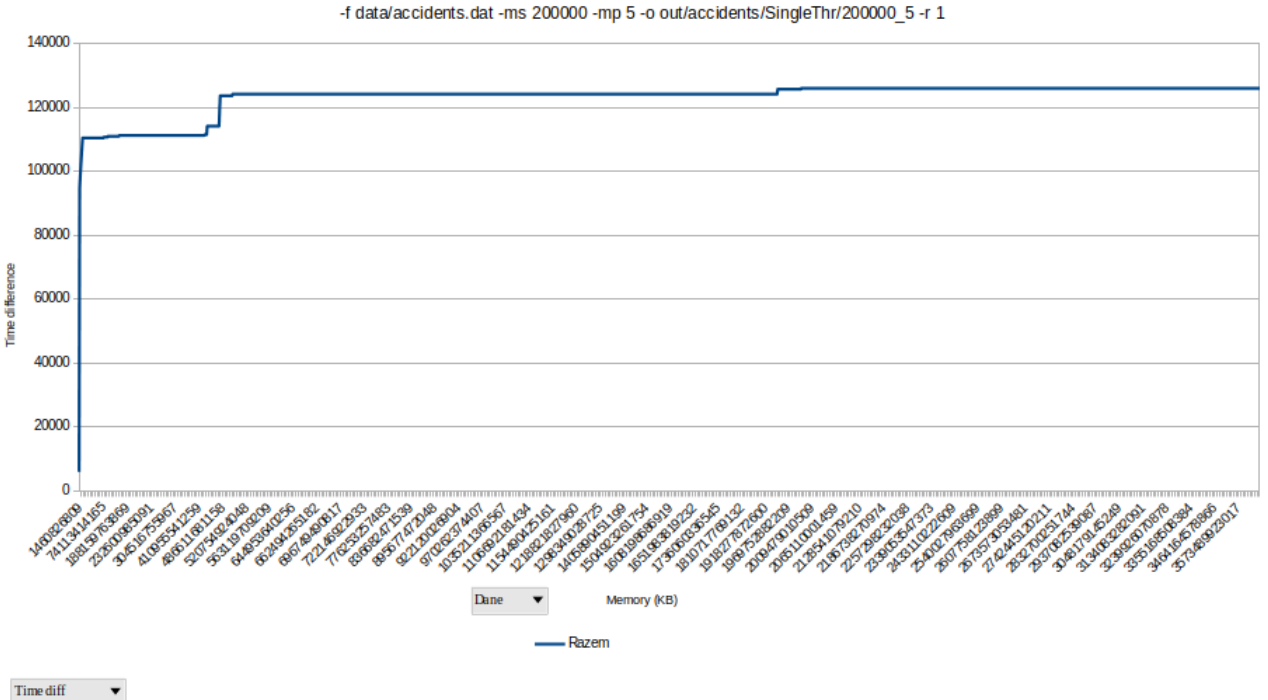
Wyniki eksperymentów

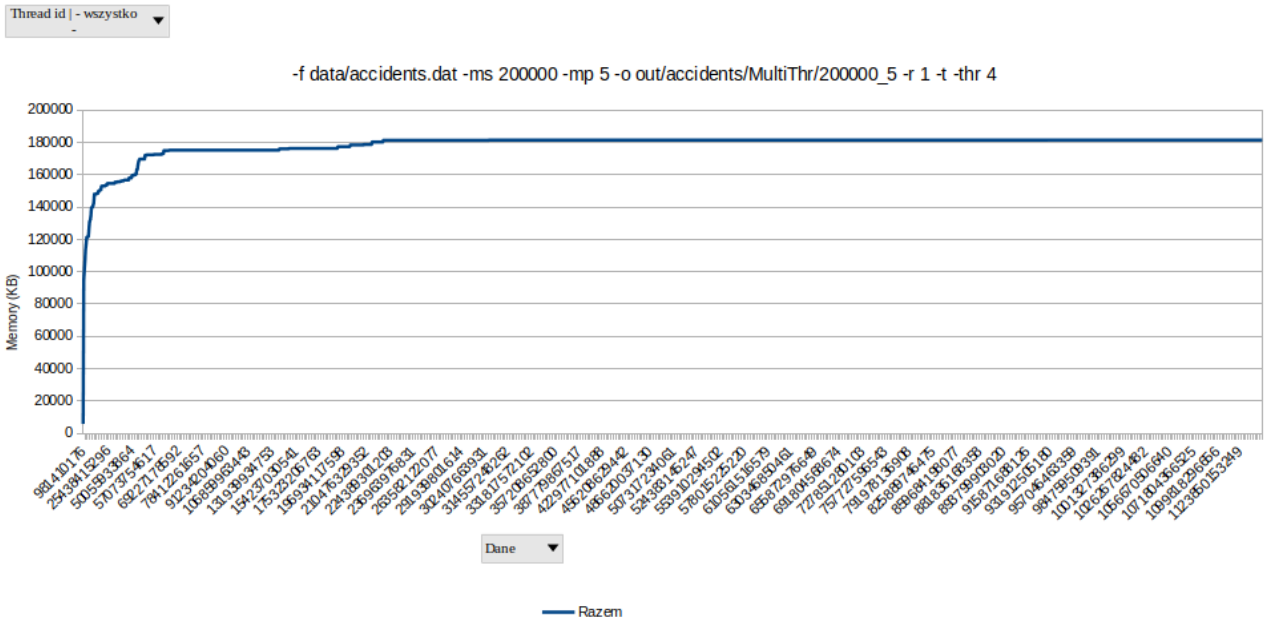
Wykonano eksperymenty dla podanych paramentów wejściowych:

- mushroom, jednowątkowe:
  - -f data/mushroom.dat -ms 2000 -mp 5 -o out/mushroom/SingleThr/2000_5 -r 10
  - -f data/mushroom.dat -ms 3000 -mp 10 -o out/mushroom/SingleThr/3000_10 -r 10
  - -f data/mushroom.dat -ms 4000 -mp 10 -o out/mushroom/SingleThr/4000_10 -r 10
  - -f data/mushroom.dat -ms 5000 -mp 10 -o out/mushroom/SingleThr/5000_10 -r 10
- mushroom, wielowątkowe:
  - -f data/mushroom.dat -ms 2000 -mp 5 -o out/mushroom/MultiThr/2000_5 -r 10 -t -thr 4
  - -f data/mushroom.dat -ms 3000 -mp 10 -o out/mushroom/MultiThr/3000_10 -r 10 -t -thr 4
  - -f data/mushroom.dat -ms 4000 -mp 10 -o out/mushroom/MultiThr/4000_10 -r 10 -t -thr 4
  - -f data/mushroom.dat -ms 5000 -mp 10 -o out/mushroom/MultiThr/5000_10 -r 10 -t -thr 4
- pumsb_star, jednowątkowe:
  - -f data/pumsb_star.dat -ms 25000 -mp 10 -o out/pumsb_star/SingleThr/25000_10 -r 4
  - -f data/pumsb_star.dat -ms 30000 -mp 10 -o out/pumsb_star/SingleThr/30000_10 -r 4
  - -f data/pumsb_star.dat -ms 30000 -mp 20 -o out/pumsb_star/SingleThr/30000_20 -r 4
- pumsb_star, wielowątkowe:
  - -f data/pumsb_star.dat -ms 25000 -mp 10 -o out/pumsb_star/MultiThr/25000_10 -r 4 -t -thr 4
  - -f data/pumsb_star.dat -ms 30000 -mp 10 -o out/pumsb_star/MultiThr/30000_10 -r 4 -t -thr 4
  - -f data/pumsb_star.dat -ms 30000 -mp 20 -o out/pumsb_star/MultiThr/30000_20 -r 4 -t -thr 4
- accidents:
  - -f data/accidents.dat -ms 200000 -mp 5 -o out/accidents/MultiThr/200000_5 -r 1 -t -thr 4
  - -f data/accidents.dat -ms 200000 -mp 5 -o out/accidents/SingleThr/200000_5 -r 1


-f data/pumsb_star.dat -ms 30000 -mp 20 -o out/pumsb_star/SingleThr/30000_20 -r 4
Time difference
Średnia - Memory (KB)
Średnia - Memory (KB)


-f data/pumsb_star.dat -ms 30000 -mp 20 -o out/pumsb_star/MultiThr/30000_20 -r 4 -t -thr 4
Średnia - Memory (KB)
Thread id | - wszystko


-f data/accidents.dat -ms 200000 -mp 5 -o out/accidents/SingleThr/200000_5 -r 1
Time difference
Dane
Memory (KB)
Razem
Time diff



Wywołanie	Maksymalna pamięć (KB)	Czas wykonania
-f data/accidents.dat -ms 200000 -mp 5 -o out/accidents/MultiThr/200000_5 -r 1 -t -thr 4	181412	00:19:14.407578805
-f data/accidents.dat -ms 200000 -mp 5 -o out/accidents/SingleThr/200000_5 -r 1	125980	01:01:19.992160898
-f data/pumsb_star.dat -ms 25000 -mp 10 -o out/pumsb_star/MultiThr/25000_10 -r 4 -t -thr 4	36020	00:03:21.980339892
-f data/pumsb_star.dat -ms 30000 -mp 10 -o out/pumsb_star/MultiThr/30000_10 -r 4 -t -thr 4	33984	00:01:24.024249800
-f data/pumsb_star.dat -ms 30000 -mp 20 -o out/pumsb_star/MultiThr/30000_20 -r 4 -t -thr 4	34004	00:01:19.954379831
-f data/pumsb_star.dat -ms 25000 -mp 10 -o out/pumsb_star/SingleThr/25000_10 -r 4	24848	00:09:48.811068075
-f data/pumsb_star.dat -ms 30000 -mp 10 -o out/pumsb_star/SingleThr/30000_10 -r 4	24160	00:04:10.190950441
-f data/pumsb_star.dat -ms 30000 -mp 20 -o out/pumsb_star/SingleThr/30000_20 -r 4	24108	00:04:07.424634643
-f data/mushroom.dat -ms 2000 -mp 5 -o out/mushroom/MultiThr/2000_5 -r 10 -t -thr 4	7924	00:00:00.555339740
-f data/mushroom.dat -ms 3000 -mp 10 -o out/mushroom/MultiThr/3000_10 -r 10 -t -thr 4	7544	00:00:00.274789720
-f data/mushroom.dat -ms 4000 -mp 10 -o out/mushroom/MultiThr/4000_10 -r 10 -t -thr 4	7352	00:00:00.092072678
-f data/mushroom.dat -ms 5000 -mp 10 -o out/mushroom/MultiThr/5000_10 -r 10 -t -thr 4	7220	00:00:00.048988576
-f data/mushroom.dat -ms 2000 -mp 5 -o out/mushroom/SingleThr/2000_5 -r 10	5892	00:00:01.923576080
-f data/mushroom.dat -ms 3000 -mp 10 -o out/mushroom/SingleThr/3000_10 -r 10	5896	00:00:00.930553389

-f data/mushroom.dat -ms 4000 -mp 10 -o out/mushroom/SingleThr/4000_10 -r 10	5892	00:00:00.270654238
-f data/mushroom.dat -ms 5000 -mp 10 -o out/mushroom/SingleThr/5000_10 -r 10	5932	00:00:00.113570290

Reszta wyników wraz ze szczegółami znajduje się w folderze ‘out’.

Wnioski

Dla tych samych parametrów wejściowych, program zdołał wykonać się ponad dwukrotnie, a nawet trzykrotnie szybciej przy wykorzystaniu 4 wątków niż tylko w wersji jednowątkowej. Zużycie pamięci natomiast jest zwiększone o około 40-50% w stosunku do wersji jednowątkowej. Jednocześnie w wersji wielowątkowej widać większe wahania zużycia pamięci. Może być to spowodowane tym, że dla jednowątkowej wersji głębokość rekursji jest z góry ogranicza oraz algorytm przez większość czasu nie wraca z rekursji o kilka poziomów na raz. W przypadku wielowątkowego przejścia wraz z przyjętą polityką pierwszeństwa wykonywania prac najgłębszych w rekursji, można spodziewać się dużych wahań.

W jedno oraz wielowątkowym wykonaniu algorytmu można zauważyć duże znaczenie parametru minimum support dotyczące czasu wykonania. Im większa jest jego wartość, tym potrzeba mniej czasu do jego zakończenia. Jest to związane najpewniej z tym, że ten parametr pośrednio zmniejsza możliwą głębokość rekursji w programie. Jednocześnie zajętość pamięciowa wydaje się minimalnie zwiększać wraz ze zwiększeniem parametru minimum support. To samo można powiedzieć o parametrze maximum pattern length, którego zwiększenie również wpływa na głębokość rekursji, w tym zużycia pamięci.

Dane zgromadzone podczas eksperymentów pozwalają stwierdzić, iż przy zwiększonym zapotrzebowaniu na pamięć możliwe staje się kilkukrotnie szybsze wykonanie algorytmu. Same dane zajmują niewiele miejsca w porównaniu z wykorzystaną pamięcią podczas działania programu. Im większe dane, tym mniejszy wydaje się stosunek zajętości programu do zajętości samych danych. Stosunek ten wykazuje od 5 do 10 większe zapotrzebowanie na pamięć względem danych zapisanych na dysku.

Operacje wykonywane podczas działania algorytmu są w większości niezależne od siebie. Dzięki temu prawdopodobnie w nowych wersjach algorytmu użycie GPU pozwoli uzyskać jeszcze lepsze rezultaty.

Wykorzystany sprzęt

- Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz
- 2 x 8GiB SODIMM DDR3 Synchronous 1600 MHz (0,6 ns)
- 1TB WDC WD10S21X-24R
- Linux Mint 5.4.0-91-generic

Użyte technologie

- C++17 - główna implementacja algorytmu
- Python – pomocnicze skrypty do obróbki danych

Sposób użycia

Program został napisany pod platformę Linux.

Kompilacja – w folderze src wykonanie komendy ‘make’

Wywołanie - ‘./src/prefixspan.out’ wyświetli opcje programu. Należy go wywołać z wymaganymi argumentami.