Zwielowatkowienie problemu Longest Common Sequence (LCS)

Maciej Sanocki

November 2023

1 Wprowadzenie

Znajdowanie najdłuższych podciagów dwóch tekstów okazuje sie kluczowe w rozważaniu wielu problemów z różnych dziedzin. Czestym jest, że interesuje nas porównanie bardzo długich ciagów, naprzykład kodów genetycznych, w jak najkrótszym czasie co tłumaczy cheć optymalizacji czasu wykonania.

Rozważe dwa algorytmy dynamiczne rozwiazujace problem LCS:

- Najcześciej stosowany algorytm dynamiczny, obliczajacy LCS w czasie $O(n^2)$, i pamieci $O(n^2)$
- Algorytm Hirschberga, który również działa w czasie $O(n^2)$, ale za to potrzebuje liniowej pamieci.

2 Programy

Wszystkie programy na wejściu przyjmója liczbe testów z i długość każdego ciagu znaków w teście n, a nastepnie z par ciagów, każdy o długości n.

Sekwencyjne:

LCS_01.cpp Algorytm wyliczajacy LCS, przechodzacy po kolejnych wierszach tablicy wynikowej.

LCS_02.cpp Algorytm wyliczajacy LCS, przechodzacy po kolejnych przekatnych tablicy wynikowej, jednak tablica wynikowa nadal jest ustawiona wierszami.

LCS_03.cpp Algorytm wyliczajacy LCS, przechodzacy po kolejnych przekatnych tablicy wynikowej, specjalnie ułożonej.

LCS_04.cpp Algorytm wyliczajacy LCS, przechodzacy po kolejnych wierszach tablicy wynikowej, ale używajacy trick-u(numer paragrafu) pozwalajacego na nie odwoływanie sie do tego samego wiersza w tablicy wynikowej.

Hirschberg_0.cpp Algorytm Hirshberg-a.

Zrównoleglone: **LCS_threads1.cpp** Bardzo zła implementacja programu wielowatkowego, głównie przez tworzenie watków niepotrzebnie wiele razy.

LCS_threads2.cpp zwielowatkowiona wersja programu programu LCS_02.cpp używajaca std::barrier do synchronizacji przed przejściem do kolejnej przekatnej.

LCS_omp2.cpp zwielowatkowiona wersja programu programu LCS_02.cpp używajaca biblioteki omp.

LCS_threads3_1.cpp i LCS_0threads3_2.cpp zwielowatkowiona wersja programu programu LCS_03.cpp używajaca std::barrier do synchronizacji przed przejściem do kolejnej przekatnej, Obydwa programy różnia sie sposobem chodzenia przez watki po przekatnej, wersja 3_1 skacze o liczbe watków, a w wersji 3_2, każdy watek oblicza spójny kawałek przekatnej. (wersja 3_2 z bardzo dziwnych powodów kompletnie nie działa dla wiecej niż 4 watków, czyli ilości rdzeni na mojej maszynie).

LCS_omp3.cpp zwielowatkowiona wersja programu programu LCS_03.cpp używajaca biblioteki omp.

LCS_omp4.cpp zwielowatkowiona wersja programu programu LCS_04.cpp używajaca biblioteki omp.

Hirschberg_threads.cpp Zwielowatkowiony algorytm Hirschberg-a z użyciem techniki dziel i zwyciężaj dla poczatkowych głebokości.

Hirschberg_omp.cpp Zwielowatkowiony algorytm Hirschberg-a z użyciem techniki dziel i zwyciężaj z użyciem kolejki zadań z biblioteki omp.

3 Problem

Zanajdowanie najdłuższego wspólnego podciagu najcześciej jest rozwiazywane przez użycie algorytmów dynamicznych. Problem dla ciagów $X = (x_1 x_2 \dots x_n)$, $Y = (y_1 y_2 \dots y_m)$, oraz $X_0, X_1, \dots, X_n$ i $Y_0, Y_1, \dots, Y_n$ jako odpowiednio ich prefixy, możemy zapisać rekurencyjnie w postaci:

$$LCS(X_i, Y_j) = \begin{cases} \epsilon, & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1})x_i, & \text{if } x_i = y_j \\ \max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)), & \text{else} \end{cases}$$

Analizujac te funkcje widzimy, że pola w tablicy wynikowej sa zależne od wartości w polach:

- w tym samym wierszu, ale kolumne wwcześniej.
- w tej samej kolumnie, ale wiersz wyżej.
- kolumne wcześniej i wiersz wyżej.

4 Optymalizacja algorytmów sekwencyjnych

W przypadku optymalizacji dla programów sekwencyjnych możemy zauważyć, jak wielka role odgrywa dostep do pamieci. W szczególności jak porównamy programy LCS_03.cpp i LCS_02.cpp. W obydwu programach przechodzimy po kolejnych przekatnych, ale tablica wynikowa w programach jest opowiednio ułożona wzdłuż przekatnych i wzdłuż wierszy. W rezultacie możemy zauważyć, że program LCS_03.cpp działa najszybciej wśród wszystkich programów sekwencyjnych i aż trzy razy szybciej od LCS_02.cpp na sekwencjach długości 10 tysiecy znaków. Jest też znaczaco szybszy od algorytmu chodzacego po wierszach(LCS_01.cpp), który też *cache-uje* pamieć dosyć optymalnie, ale wyniki w tablicy sa zależne od wartości obliczanych bezpośrednio przed nimi, co ogranicza możliwość stosowania operacji takich jak chociażby SIMD.

Nie jest też niczym dziwnym, że algorytm Hirschberg-a radzi sobie gorzej, to znaczy jest 2 razy wolniejszy od podstawowego LCS_01.cpp.

5 Zwielowatkowienie klasycznego algorytmu dynamicznego

Metoda narzucajaca sie podczas analizy funkcji z sekcji 3. jest iterowanie sie po tablicy wynikowej wzdłuż jej przekatnych ponieważ komórki macierzy należace do tej samej przekatnej sa od siebie niezależne. Jeśli chcielibyśmy sie iterować wzdłuż kolejnych wierszy, to natrafilibyśmy na problem ponieważ każda komurka zależy od wartości w tym samym wierszu.

Program LCS_threads1.cpp ma za zadanie pokazać jak drogie potrafi być tworzenie watków bedac 8-10 razy wolniejszym od implementacji sekwencyjnej na średniej wielkości testach.

W przypadku wielowatkowych implementacji, zarzadzanie pamiecia staje sie jeszcze ważniejsze, o czym świadcza wieksze różnice w czasach wykonania, to znaczy program LCS_threads2.cpp jest 4 razy wolniejszy od programu LCS_threads3_1.cpp na średnich testach.

Możemy też zauważyć, że *parallel for* z biblioteki openMP radzi sobie wyraźnie gorzej z duża ilościa stosunkowo małych i prostych zadań, bedac pare razy gorszym w porównaniu z implementacjami na zwykłych threadach z bariera.

Dość sporym minusem korzystania z przekatnych jest fakt, że przekatne sa różnej wielkości, co jest problemem zwłaszcza na poczatku i na końcu wywoływania programu gdzie watki wykonuja bardzo mało pracy, a i tak musza sie synchronizować zabierajac czas. Dlatego spróbowałem w LCS_omp4.cpp i LCS_04.cpp implementacji tricku który obchodzi ten problem używajac dodatkowej tablicy pamietajacej ostatnie wystapienia kolejnych elementów z alfabetu, co pozwala na odwoływanie sie tylko do wiersza wyżej. Wprowadza to jednak dodatkowa złożoność w obliczeniach radzac sobie istotnie gorzej z obliczaniem problemu.

6 Zwielowatkowienie algorytmu Hirschberg-a

Algorithm 1 bardzo ogólny schemat algorytmu Hirschberga

if $m <= 2$ *or* $n <= 2$ **then**
  zwróć rozwiazanie obliczone algorytmem pamieciochłonnym
end

oblicz ostatni wiersz tablicy wynikowej dla pierwszej połowy pierwszej sekwencji i drugiej sekwencji.(1)

oblicz ostatni wiersz tablicy wynikowej dla odwróconej drugiej połowy pierwszej sekwencji i odwróconej drugiej sekwencji.(2)

Znajdź taki punkt podziału k.(3)

zwróć sume dla wywołań na mniejszych podzadaniach według podziału k.(4)

Analizujac algorytm Hirschberg-a możemy, go podzielić na cześci które moga być wykonane asynchronicznie i cześci wykonywane synchronicznie.

Tak wiec cześci (1) i (2) sa od siebie niezależne i moga być wywołane równolegle, ale musza sie zsynchronizować przed wywołaniem (3), po czym (4) cześć również może być wykonana równolegle.

Zdecydowałem sie na spróbowanie dwóch rozwiazań zrównoleglania dla tego algorytmu:

threads obliczamy operacje (1) i (2), oraz (4) równolegle tylko dla pierwszych paru głebokości rekursji.

OpenMP dodajac jako podzadania do puli watków operacje które moga być wykonane równolegle.

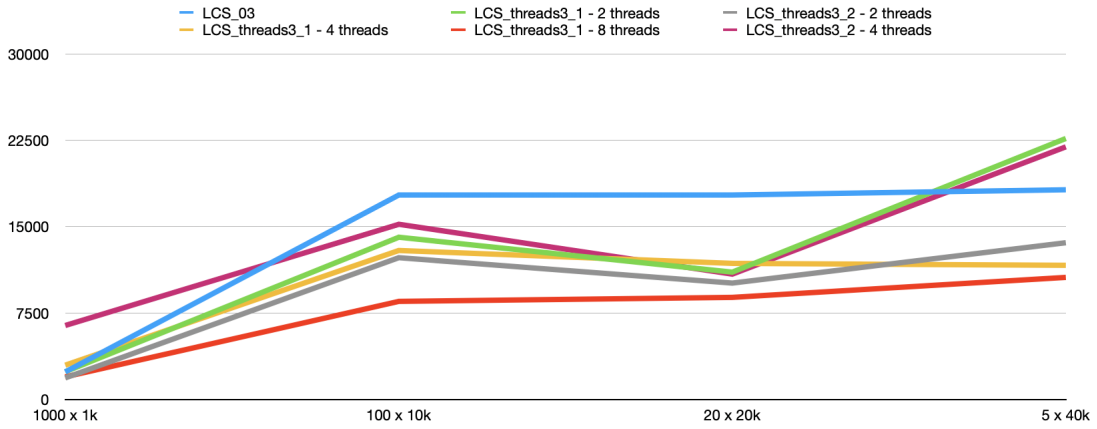
Co było dla mnie dość sporym zaskoczeniem, wersja threads okazała sie bardzo dobrze zrównoleglajaca sie, osiagajac wyniki konkurujace z najlepszymi programami urzywajacymi pamieci kwadratowej na średnich testach i miażdżac je na bardzo dużych testach. Jest to po cześci zasługa tego, że watki rzadko odwołuja sie do tych samych miejsc w pamieci, oraz z faktu, że środowisko wykonania posiada bardzo mało pamieci RAM(16 GB).

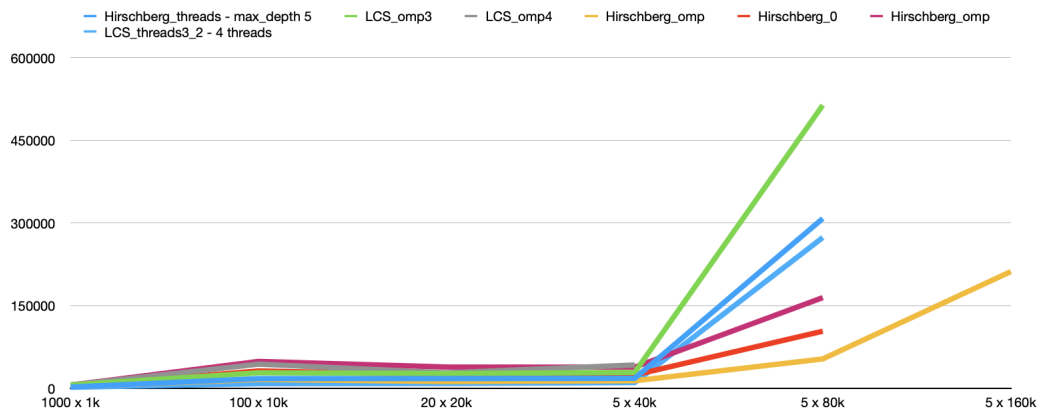
Z drugiej jednak strony wersja używajaca openMP, która na pierwszy rzut oka mogła lepiej zarzadzać opciażeniem wszystkich watków okazała sie znaczaco gorsza, chociaż i tak poprawiła czas o 36% wzgledem wersji sekwencyjnej.

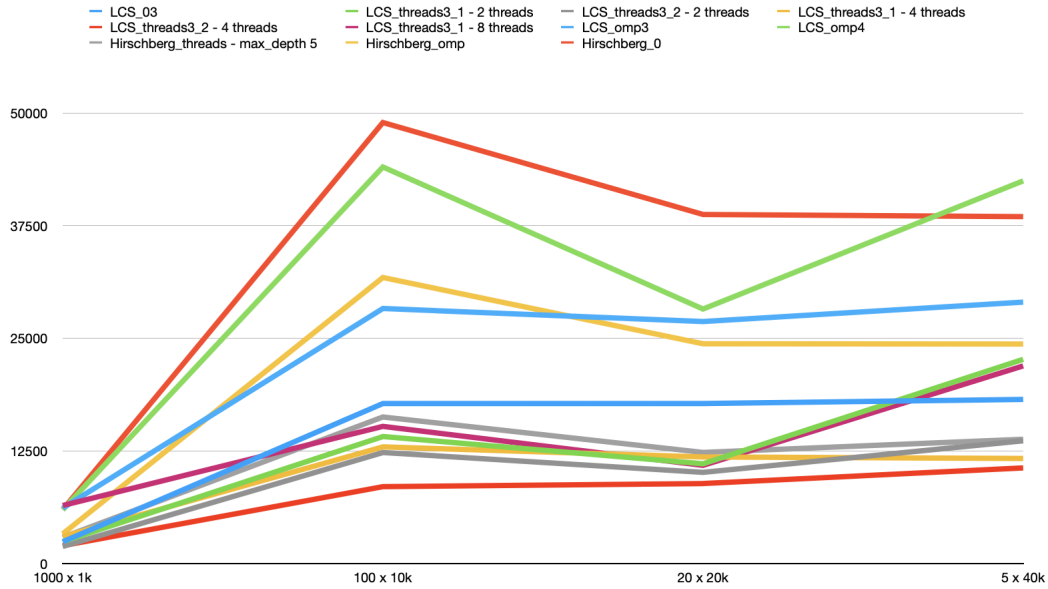
7 Porównanie wyników

Wyniki testów prezentuja sie nastepujaco. Możemy zauważyć, że bardzo duża role w rozwiazywaniu problemu odgrywa dostepna pamieć, ponieważ programy oparte na algorytmie Hirschberga osiagaja znaczaco lespsze rezultaty.

	1000 x 1k	100 x 10k	20 x 20k	5 x 40k	5 x 80k	5 x 160k
LCS_03	2388	17744	17744	18201	308269	
LCS_threads3_1 - 2 threads	2402	14081	11049	22652	330317	
LCS_threads3_2 - 2 threads	1868	12304	10099	13604	320884	
LCS_threads3_1 - 4 threads	2973	12928	11810	11642	302726	
LCS_threads3_2 - 4 threads	1958	8511	8858	10595	273794	
LCS_threads3_1 - 8 threads	6419	15205	10875	21925	381821	
LCS_omp3	6130	28302	26852	29010	513836	
LCS_omp4	5950	44020	28241	42477	b.d	
Hirschberg_threads - max_depth 5	2883	16249	12312	13775	53459	211923
Hirschberg_omp	3274	31742	24384	24353	103771	
Hirschberg_0	5959	48958	38746	38504	164715	







8 Dalsza Możliwość poprawy

Wszystkie programy, używaja instrukcji warunkowych *if* co powoduje, że kompilator nie stosuje instrukcji SIMD, których zastosowanie mogłoby skutkować dalsza redukcja czasu wykonania.