Sprawozdanie z wykonanego ćwiczenia:

Koszty współdzielenia danych w systemach równoległych Linux/Windows

Szymon Sroka 141312

9 kwietnia 2021r.

Procesor: Intel Core i5-5200U CPU @ 2.20GHz (4CPUs), ~2.2GHz

System operacyjny: Windows 10 Pro 64-bitowy

Kompilator: Microsoft C++ compiler (MSVC) w Visual Studio 2017

Procesory logiczne: **4**

Procesory fizyczne: **2**

Wersja kodu	Czas przetwarzania [s]
PI1	0,210

Wersja kodu	4 wątki		2 wątki		1 wątek	
	Czas przetwarzania [s]	wsp. przyspiesz.	czas przetwarzania [s]	wsp. przyspiesz.	czas przetwarzania [s]	wsp. przyspiesz.
PI2	0,276	0,761	0,205	1,024	0,340	0,618
PI3	10,722	0,020	4,120	0,051	1,802	0,117
PI4	0,118	1,780	0,122	1,721	0,201	1,045
PI5	0,119	1,765	0,112	1,875	0,193	1,088
PI6	2,570	0,082	3,290	0,064	1,680	0,125

PI1:

- pierwsza wersja kodu zakładała sekwencyjne wykonanie zadania. Czas przetwarzania zmierzony w tym wariancie wyniósł 0,21 sekundy i został użyty do obliczenia współczynnika przyspieszenia w pozostałych wariantach zadania.

Pl2:

- zmienna *i* jest prywatna (ze względu na zastosowaną dyrektywę #pragma omp for), zmienna *x* również jest prywatna. Pozostałe zmienne są współdzielone: *step*, *num_steps* do odczytu, ponadto *sum* jest współdzielona także do zapisu. Poza blokiem równoległym zmienne *pi*, *start*, *stop* są globalne do odczytu i zapisu.
- współdzielenie zmiennej *sum* w tym kodzie nie jest bezpieczne, stąd wartość liczby pi otrzymana w tym wariancie zadania jest niepoprawna - zachodzi zjawisko wyścigu do zmiennej *sum* (brak synchronizacji - możliwe do usunięcia poprzez zastosowanie dyrektywy *atomic*, w wariancie jednowątkowym zmienna PI jest obliczana poprawnie, ponieważ nie zachodzi wspomniane wyżej zjawisko).
- W tym wariancie zadania nie zachodzi unieważnienie kopii linii pamięci procesora, ponieważ nie mamy do czynienia z false sharing'iem czy zastosowaniem dyrektyw *#pragma*, które by ją wywoływały. Wnioskuję to także po krótkim czasie przetwarzania – gdyby czasy były długie, dane byłyby przechowywane w pamięci i mogłyby wymagać unieważniania, stąd czas przetwarzania uległby wydłużeniu. Czas przetwarzania w tym zadaniu jest (w najlepszym przypadku) nieco szybszy od programu sekwencyjnego (co wynika z podziału pracy, jaką musi wykonać pętla for, między kilka wątków)

PI3:

- w tym wariancie zadania zarówno podział pracy, jak i wynik są poprawne (bez wyścigu – dbamy o to, aby jednocześnie wyłączne jeden proces mógł operować na zmiennej *sum*)
- czas przetwarzania znacznie wzrósł, ponieważ w porównaniu z sekwencyjną wersją programu tu zachodzi konieczność zamykania i otwierania zamków przez procesy przy wykonywaniu operacji `sum += 4.0 / (double)(1. + x * x)`, a także odczytywania, zmieniania wartości, zapisywania zmiennej *sum* w pamięci, unieważniania linii danych oraz pobierania aktualnej kopii linii danych zawierających zmienną *sum*.
- dyrektywa atomic synchronizuje wartość zmiennej *sum* z pamięcią, przez to wszystkie wątki znają poprawną aktualną wartość tej zmiennej
- w tym wariancie zadania jest zapewniona spójność danych powielonych
- na poziomie wątku atomowość (niepodzielność) uaktualnienia zmiennej współdzielonej w systemie można zapewnić za pomocą opisanej niżej dyrektywy atomic:

Dyrektywa atomic powoduje, że podczas działania program oblicza najpierw prawą stronę przypisania wartości do zmiennej, następnie następuje założenie zamka i flush dla prawej strony (synchronizacja zmiennej sum *z pamięcią), modyfikacja sumy, flush dla lewej strony, zdjęcie zamka. W takim wypadku zmienna sum jest zapisywana w pamięci, a w rejestrze pojawia tylko tymczasowo na czas obliczeń.*

PI4:

- w tym wariancie zadania otrzymana wartość liczby pi jest poprawna. Zapewniliśmy lokalność dostępu do zmiennej *lsum* i przez to przetwarzanie jest bardziej efektywne niż w poprzedniej wersji kodu – czasy przetwarzania są wielokrotnie krótsze niż w przypadku sekwencyjnej wersji. Na skrócenie czasu przetwarzania ma również wpływ zmniejszenie liczby zakładanych zamków, opisane w punkcie poniżej
- każdy wątek pracuje na swojej prywatnej zmiennej *lsum* i za jej pomocą modyfikujemy zmienną globalną *sum*, jednak tych modyfikacji jest znacznie mniej (stąd i unieważnień linii pamięci procesora jest znacznie mniej) niż w poprzedniej wersji kodu– wcześniej blokada była zakładana wiele milionów razy, w PI4 – tylko raz przez każdy z procesów, stąd i czas przetwarzania jest o wiele krótszy – znacznie krótszy nawet od wersji sekwencyjnej PI1.

PI5:

- w tej wersji kodu usunęliśmy deklarację prywatnej zmiennej lsum. Za pomocą dyrektywy *#pragma omp for reduction(+:sum)* sprawiliśmy, że dotychczas współdzielona (globalna) zmienna *sum* staje się – w ramach bloku, który jest rozpoczęty za pomocą redukcji –

zmienną prywatną. W momencie gdy blok się kończy następuje scalenie (w oparciu o operację dodawania) wartości przygotowanych przez poszczególne wątki. Wartość początkowa zmiennej sum w bloku rozpoczętym za pomocą redukcji jest neutralna dla wybranego operatora – w przypadku PI5 wynosi *0*.

- Wartość pi otrzymana w tym programie jest poprawna. Nie zauważam znacznych zmian w czasie przetwarzania w stosunku do PI5, wyniki i wnioski z poprzedniego podpunktu pozostają aktualne – oba programy operują na prywatnej zmiennej, z tą różnicą że w PI4 deklarowaliśmy ją wprost, a w PI5 za pomocą dyrektywy reduction

PI6:

- w tym wariancie zadania zdefiniowaliśmy współdzieloną tablicę double *tab[80],* przy pomocy której scalamy wyniki przygotowane przez wątki*;* każdy z wątków na podstawie swojego identyfikatora wybiera miejsce, do którego zapisuje obliczony wynik w tablicy *tab*.
- scalanie wyników przygotowanych przez wątki ma miejsce jeszcze w rejonie równoległym programu. Ta operacja wymaga skorzystania ze współdzielonej tablicy *tab*, stąd potrzebne jest zastosowanie dyrektywy #pragma omp atomic, które zapewni synchronizację w dostępie do pamięci między wątkami i pozwoli zaobserwować zjawisko zwane f*alse sharing'iem*:

„Zjawisko to polega na wielokrotnym, cyklicznym (wywołanym przez wielokrotne zapisy w różnych procesorach wartości do tej samej linii pamięci podręcznej procesora):

- *unieważnianiu powielonych linii pamięci w pp procesorów, które przestają być aktualne w wyniku zapisu danych do jednej z wielu kopii tej linii,*
- *konieczności sprowadzenia do pamięci podręcznej procesora realizującego kod wątku (korzystającego z danych sąsiednich) aktualnej kopii danych z unieważnionej linii."*

(definicja false sharingu pochodzi z treści zadania)

- W tym wariancie zadania obserwuję znaczne wydłużenie czasu przetwarzania w porównaniu z wariantem PI4. Ze względu na to, że wartości wykorzystywane przez różne wątki są umieszczone obok siebie w pamięci, wykonanie modyfikacji na elemencie tab[id] powoduje unieważnienie linii pamięci i w efekcie znaczne wydłużenie czasu przetwarzania.
- *Dodatkowa obserwacja: podczas laboratoriów omówiliśmy również taki wariant zadania, w którym każdy z procesów odczytuje i zapisuje do elementu tab znajdującego się w różnych liniach pamięci. Wtedy dostęp był realizowany jedynie do pamięci prywatnej – nie było unieważnień, każdy wątek pracował na swojej linii pamięci. Dostęp do pamięci był konieczny, ale tylko do pamięci używanego procesora, stąd czas przetwarzania był krótszy niż w wariancie opisanym w podpunkcie wyżej.*

PI7:

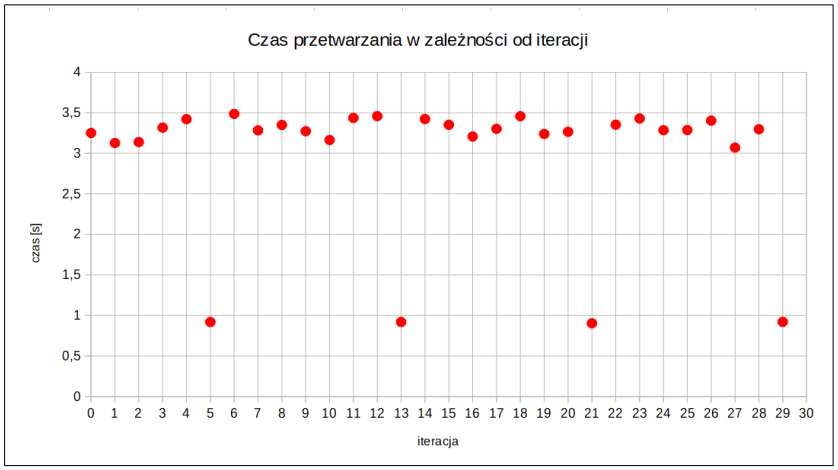
W celu wyznaczenia długości linii pp procesora najpierw upewniłem się, czy spełniłem wszystkie konieczne warunki do poprawnego przeprowadzenia eksperymentu:

- ciągły obszar danych – jedna tablica przez wszystkie iteracji obliczeń,
- dwa wątki przetwarzania, dwa procesory fizyczne – dwie pamięci podręczne procesora
- zapisy do pamięci wywołujące niezamierzone współdzielenie
- wyrównanie słowa użytej tablicy do brzegu linii

Następnie zmodyfikowałem kod z zadania PI6 tak, aby jedno uruchomienie kodu spowodowało wykonanie przez dwa wątki pracujące na dwóch procesorach fizycznych z różnymi pamięciami podręcznymi procesora wielu odczytów i zapisów do sąsiednich słów tej samej tablicy przechowującej sumy częściowe algorytmu obliczania liczby pi (*tab[50]*).

Gdy kod został zmodyfikowany, uruchomiłem go: dwa wątki rozpoczęły wielokrotne obliczanie liczby pi przy użyciu za każdym razem innej pary słów z tablicy: tab[0] i tab[1]; tab[1] i tab[2] i tak dalej, aż do przejścia przez całą tablicę. Dla każdej spośród 30 iteracji wyświetlałem czas obliczeń wraz z numerem iteracji odpowiadającej użytym słowom – elementom tablicy.

W dalszym etapie przeszedłem do analizy otrzymanych wyników:



Zauważyłem, że przy użyciu 2 wątków na dwóch słowach pamięci z jednej tablicy uzyskałem różne czasy przetwarzania. Zależą one od tego, czy słowa użyte do obliczeń znajdują się w jednej, czy dwóch liniach. Czas krótszy zaobserwowałem w iteracjach: 5, 13, 21, 29 – w tych iteracjach praca wątków przebiegała przy użyciu dwóch różnych linii, nie nastąpiło unieważnienie kopii linii pamięci podręcznej procesora, stąd wnioskuję, że każda z wymienionych iteracji oznacza jeden z końców linii pamięci podręcznej. Do obliczeń wybiorę koniec pamięci wskazywany przez iterację numer 5. Sąsiednim do niej końcem linii jest iteracja 13. Odległość między kolejnymi końcami odpowiada poszukiwanej długości linii pamięci podręcznej procesora, stąd wyznaczona przeze mnie długość będzie wynosiła (13-5)*sizeof(double) = 8 * (8 bajtów) = 64 bajty.