Sprawozdanie – Przetwarzanie Równoległe – Zadanie Pi

1. Na wstępie przedstawię parametry środowiska, które posłużyły mi do wykonania zadania:
   -Procesor: Intel(r) Core(TM) i7-6700HQ CPU @ 2.60ghz 2.59Hhz
   -System operacyjny: Windows 10
   -Kompilator: Microsoft Visual Studio 2017

   Liczba procesorów logicznych: 8

   Liczba procesorów fizycznych: 4

2. Następnie przedstawię tabelę zawierającą wyniki pracy z odpowiednią ilością wątków:

   Czas uzyskany dla przetwarzania sekwencyjnego: 1,411 s.

Czas (s)	PI2	PI3	PI4	PI5	PI6
8 wątków	1,696	135,094	0,475	0,443	1,222
4 wątki	1,133	90,502	0,829	0,747	1,062
2 wątki	1,874	47,759	1,42	1,437	1,913

Współczynnik przyspieszenia obliczeń dla uruchomień równoległych kodu:

Przyspieszenie	PI2	PI3	PI4	PI5	PI6
8 wątków	0,831958	0,010445	2,970526	3,185102	1,154664
4 wątki	1,245366	0,015591	1,702051	1,888889	1,328625
2 wątki	0,752935	0,029544	0,993662	0,981907	0,737585

Kolejno, postaram się omówić uzyskane wyniki porównując do siebie czasy uzyskane podczas przetwarzania sekwencyjnego i równoległego przy użyciu różnej ilości wątków. W omawianiu wyników będę się posługiwał nazwami zmiennych używanymi w kodzie źródłowym, podanym na stronie jako szkielet programu.

a) W programie PI2 należało zrównoleglić obliczenia poprzez dodanie regionu równoległego oraz dyrektywę podziału pracy (#pragma omp for). W ramach tego programu zmienna „x" oraz „sum" nie są zmiennymi prywatnymi. Zmienna iterująca „i" jest prywatna w ramach kolejnego przetwarzania, to znaczy, że każdy nowy wątek, który otrzymał zadanie w ramach danego kroku, otrzymał je z przypisaną wartością „i" i jest ona dla niego wyłączna.
   Podczas użycia 8 wątków, czas przetwarzania wydłużył się. Jest to spowodowane tym, że liczba fizycznych rdzeni procesora jest równa 4. Zatem 4 procesory wykonują pracę ośmiu wątków, a pamięć podręczna, której używają, jest współdzielona w ramach jednego procesora. Dwa procesory logiczne w technologii Hypertreading cechuje się jedną pamięcią podręczna i jedną kopią linii pamięci dostępnej przez wszystkie wątki. Aby wątki mogły działać wspólnie, wymagane jest aby dane znajdujące się w pamięci podręcznej były aktualne podczas wykonywania na nich operacji. Rozgłaszanie i uaktualnianie danych oznacza wzrost kosztu realizacji, co przekłada się na czas operacji. W sytuacji, gdy używamy cztery wątki na 4

procesorach, każdy wątek korzysta ze swojej pamięci podręcznej i częsta synchronizacja nie jest wymagana (dopiero w momencie kończenia pracy w ramach przetwarzania równoległego w obrębie zadania, należy zsynchronizować otrzymane wyniki). Brak ciągłej synchronizacji przekłada się na wzrost prędkości przetwarzania, co możemy zaobserwować na otrzymanym wyniku. W sytuacji, gdy używamy dwóch wątków, również każdy procesor używa swojej pamięci podręcznej w ramach wykonywanego zadania. Zmniejszyła się jednak ilość fizycznych procesorów, co przekłada się na spowolnienie pracy w stosunku do 4 wątków. Konieczność konkatenacji wyników otrzymanych w ramach przetwarzania równoległego ostatecznie wpływa na pogorszenie czasu działania w stosunku do sekwencyjnego przetwarzania. Jednak pomimo zastosowania przetwarzania równoległego, wyniki nie są poprawne. Jest to spowodowane tym, że zmienna „x" jest współdzielona pomiędzy wątki, tak samo jak zmienna „sum". Obliczanie wyrażenia nie jest atomowe, co za tym idzie, występuje wyścig w dostępie o dane i wartość zmiennej, która została pobrana. Może się ona zmienić po jej przetworzeniu, co skutkuje brakiem spójności danych, a to z kolei przekłada się na błędny wynik (Zmienna przechowywana w rejestrze może mieć równocześnie inną wartość niż zmienna w pamięci.). Zmienna sum, może używać innej wartości „x" niż wartość wyliczona w ramach danego przetwarzania równoległego. Nie istnieje tu synchronizacja wątków w dostępie do zapisywanych i odczytywanych wartości tej samej zmiennej przez różne wątki.

b) W kodzie PI3 należało zrealizować ciąg dostępów (odczyt, zapis) do współdzielonej sumy w sposób niepodzielny przy użyciu „#pragma omp atomic". W celu usunięcia błędów współdzielenia zmiennych, użyłem konstrukcji „private(x)", która sprawia, że zmienna „x" staje się zmienną prywatną, lokalną daną wątku, kopią niewidoczną dla innych wątków. Dla każdego wątku tworzony jest nowy obiekt o typie określonym przez typ zmiennej. Przy użyciu takiego rozwiązania, wyniki stają się poprawne, lecz czas przetwarzania znacznie się wydłużył. Jest to spowodowane tym, że klauzula atomic, która zapewnia niepodzielność operacji uaktualnienia specyfikowanej lokacji pamięci. Jednak jej realizacja, dokonywana jest poprzez wykluczenie dostępu innych wątków do zmiennej i zapewnienie odczytu oraz zapisu do pamięci przez jeden wątek. W sytuacji, gdy mamy bardzo dużo operacji, klauzula atomic zadeklarowana wewnątrz regionu równoległego spowalnia czas pracy, co możemy zaobserwować na otrzymanych wynikach. Wraz ze zmniejszaniem się ilości użytych wątków, maleje czas wykonywania zadania, ponieważ mniej wątków oczekuje na dostęp do zmiennej i mniej wątków dokonuje operacji na tejże niewiadomej.

c) W kodzie PI4 należało utworzyć i użyć zmienne prywatne każdego z wątków do zapamiętania wyniku lokalnej pracy oraz użyć dyrektywy #pragma omp atomic do zapewnienia poprawności scalenia w ramach zmiennej globalnej wyników pracy na lokalnej zmiennej. Również tutaj zastosowałem zmienną prywatną „x" w ramach jednego wątku w celu eliminacji wyścigu o dostęp do danych. W ramach regionu równoległego utworzyłem zmienną do przechowywania wyniku lokalnej pracy. Zniwelowało to konieczność używania klauzuli atomic w ciele pętli for. Każdy wątek działający w ramach pętli for, sumuje wyniki obliczone na podstawie zmiennej prywatnej „x" i przechowuje je w zmiennej lokalnej. Gdy praca wszystkich się zakończy (pętla for dotrze do końca), następuje konieczność scalenia wyników do jednej zmiennej. Po wyjściu z pętli konieczna jest synchronizacja, która posiada wewnętrzną barierę. Wymaga ona zakończenia pracy wszystkich wątków ze zbioru realizujących blok kodu. Następnie, dzięki dyrektywie atomic i wyliczonemu wcześniej wyrażeniu, operacja scalania wyniku jest poprawna i nie występuje wyścig o dostęp do danych. Z racji, że używamy zmiennej prywatnej i lokalnej, większa ilość wątków przekłada się na przyspieszenie pracy działania programu, gdyż każdy wątek posiada własne miejsce w rejestrze do składowania wyniku i nie jest konieczne uaktualnianie danych. W stosunku do przetwarzania sekwencyjnego, możemy zauważyć

wzrost prędkości wykonywania dla 8 i 4 wątków, a dla dwóch, nieznaczne pogorszenie wynikające z małej liczby równoległych operacji, konieczności scalenia wyniku i synchronizacji (konieczność czekania na zakończenie pracy wszystkich wątków).

d) W kodzie PI5 należało zrealizować automatycznie scalenie wartości obliczanych w lokalnych sumach częściowych przy pomocy klauzuli reduction. Klauzula powoduje realizację operacji redukcji (scalenie) w oparciu o podaną zmienną skalarną (współdzieloną). Konsekwencją użycia tej klauzuli jest powstanie prywatnego obiektu każdej ze zmiennych listy reduction. Na końcu regionu wartość oryginalnego obiektu (współdzielony) jest aktualizowany do wyniku będącego połączeniem za pomocą podanego operatora jego wartości początkowej i ostatecznych wartości każdego z prywatnych obiektów. Wartość obiektu oryginalnego podlegającego redukcji jest niezdeterminowana do momentu zakończenia bloku posiadającego klauzulę, co oznacza, że dopiero zakończenie wszystkich operacji scalania da poprawny wynik, a każde pośrednie wartości są niepoprawne. W moim kodzie, w celu eliminacji wyścigu w dostępie o dane, ustawiłem zmienną „x" jako prywatną. Do listy klauzuli reduction dodałem operator sumowania „+" oraz zmienną „sum". Program zachowuje się bardzo podobnie do programu PI4. Każdy z wątków posiada prywatny obiekt, w którym składuje częściowe wyniki obliczeń. Wartość oryginalnego obiektu jest aktualizowana i scalana w oparciu o znak „+" czyli sumy. Nie jest konieczne stosowanie klauzuli atomic, ponieważ operacja scalania w tym programie gwarantuje poprawność operacji. Również tutaj większa ilość wątków skutkuje przyspieszeniem pracy. Czas również jest lepszy w stosunku do realizacji sekwencyjnej. Wyjątkiem jest użycie dwóch wątków, które dają nam czas zbliżony do realizacji z programu PI1 i jest to spowodowane mniejszą ilością operacji wykonywanych równolegle oaz koniecznością scalania wyników.

e) W kodzie PI6 należało zaimplementować sumy częściowe wyznaczane przez poszczególne wątki w ramach współdzielonej przez wątki tablicy zmiennych typu double – każdy wątek modyfikuje jedno słowo w tablicy zapisując w nim swoją sumę częściową – modyfikowane przez różne wątki słowa są sąsiednimi elementami tablicy. W realizacji tego zadania użyłem dyrektywy volatile w celu wywołania niezamierzonego współdzielenia i zaobserwowania spodziewanych wyników. Każdy z wątków modyfikuje jedno słowo tablicy, której indeks odpowiada indeksowi rozpatrywanego wątku (pobranego przy pomocy omp_get_thread_num()). Również tutaj, zmienna „x" jest prywatna w celu eliminacji wyścigu o dostęp do danych. W tym przypadku można zaobserwować spadek wydajności działania programu w stosunku do zadania 4. W przypadku wykorzystania ośmiu wątków, prędkość przetwarzania spadła w porównaniu do użycia czterech wątków. Jest to spowodowane przetwarzania dwóch wątków na jednym fizycznym procesorze (jedna pamięć podręczna i jedna kopia linii pamięci dostępna przez wątek). Spadek wydajności w stosunku do programu czwartego spowodowany jest wystąpieniem zjawiska fałszywego współdzielenia danych. Zjawisko to polega na wielokrotnym, cyklicznym (wywołanym przez wielokrotne zapisy w różnych procesorach wartości do tej samej linii pamięci podręcznej procesora). Rozwiązanie polega na unieważnianiu powielonych linii pamięci w pamięci podręcznej procesorów, które przestają być aktualne w wyniku zapisu danych do jednej z wielu kopii tej linii oraz konieczności sprowadzenia do pamięci podręcznej procesora realizującego kod wątku (korzystającego z danych sąsiednich) aktualnej kopii danych z unieważnionej linii. Zjawisko to ma największe znaczenie w przypadku użycia dwóch wątków, ponieważ czas realizacji zadania stał się dłuższy niż w sytuacji sekwencyjnego przetwarzania.

3. Kolejną częścią zadania było określenie długości linii pamięci podręcznej procesora. Do obliczenia wykorzystałem jedną zmienną tablicową zajmującą 50 słów 8 bajtowych czyli

kolejnych 400 bajtów pamięci. Po implementacji kodu i przeprowadzeniu testu uzyskałem wynik:

Od tab[2] do tab[9] włącznie, czyli sumarycznie osiem elementów. Każdy element zmiennej tablicowej jest typu double, czyli również osiem bajtów. Zatem łączna długość pamięci podręcznej wynosi 8 * 8 = 64B.

4. W celu realizacji zadania wykorzystałem pętlę for mającą 50 kroków. Każdy krok odpowiadał za przesunięcie się w tablicy o jeden element, co skutkowało zmianą miejsca zapisu danych przez konkretny wątek. Dzięki takiemu rozwiązaniu mogłem zaobserwować różne czasy przetwarzania w zależności od tego, czy słowa użyte znajdują się w jednej czy dwóch liniach. Czas krótszy wystąpi wtedy, gdy praca 2 wątków przebiega przy użyciu 2 różnych linii, nie ma wtedy unieważnienia kopii linii pamięci podręcznej procesora. Czas krótszy określa zatem położenie jednego z końców linii. Po zakończeniu działania programu, mogłem łatwo znaleźć krótsze czasy przetwarzania, które powtarzały się cyklicznie. Miejsca te odnotowałem i dzięki nim mogłem wyznaczyć długość pamięci podręcznej wynoszącą 64 bajty.