Przetwarzanie równoległe – sprawozdanie

Mateusz Kreczmer 151736

Numer programu	Liczba wątków-procesów	Czas przetwarzania Równoległego [s]	Czas procesorów [s]	Przyspieszenie
P2	4	0.054000	0.054000	2.167
	8	0.048000	0.048000	2.500
P3	4	3.113000	3.113000	0.042
	8	4.240000	4.240000	0.028
P4	4	0.049000	0.049000	2.082
	8	0.047000	0.047000	2.277
P5	4	0.054000	0.054000	2.000
	8	0.042000	0.043000	2.605
P6	4	2.393000	2.393000	0.061
	8	1.883000	1.883000	0.098

Obliczenia zostały wykonane na procesorze posiadającym 4 rdzenie.

Liczba iteracji pętli for wynosi w wersjach kodu PI2-PI6 wynosi 100 000 000. Natomiast w przypadku PI7 wynosi ona 10 000 000.

1. Wersja druga kodu – PI2

zmienne prywatne – x

zmienne współdzielone – pi i sum

Współdzielenie dotyczy zarówno odczytu, jak i zapisu dla zmiennej *‘sum’*. Każdy wątek zapisuje wartość do *‘sum’*, co prowadzi do wyścigów danych.

Wynik obliczeń jest niepoprawny ze względu na wyścigi danych w przypadku współdzielonej zmiennej *‘sum’*. Kilka wątków równocześnie próbuje aktualizować wartość sum, co prowadzi do utraty danych i nieprzewidywalnych wyników.

Liczba dostępów do pamięci wzrasta ze względu na współdzielenie zmiennej *‘sum’* między wątkami głównie z powodu potrzeby synchronizacji i koordynacji dostępu do tej zmiennej. Gdy wiele wątków równocześnie próbuje odczytywać i aktualizować wartość zmiennej *‘sum’*, muszą one komunikować się ze sobą w celu zapewnienia spójności danych. Proces ten wymaga częstych dostępów do pamięci, ponieważ wątki muszą czytać i zapisywać dane w pamięci współdzielonej.

2. Wersja trzecia kodu – PI3

Wyniki obliczeń tym razem są poprawne, ponieważ operacje zapisu do ‘*sum*’ są wykonywane atomowo za pomocą dyrektywy ‘*atomic*’.

W związku w powyższym liczba dostępów do pamięci wzrosła w tej wersji kodu w porównaniu do poprzedniej, ponieważ zastosowanie dyrektywy ‘*atomic*’ wymaga dodatkowych operacji synchronizacyjnych, co prowadzi do większej ilości dostępów do pamięci.

Możemy także zaobserwować, że zwiększenie liczby wątków wprowadziło dodatkowy narzut synchronizacyjny i komunikacyjny, co zwiększyło czas przetwarzania. Ma to sens, ponieważ dyrektywa ‘*atomic*’ wymusza na wątkach koordynację dostępu do zmiennej współdzielonej.

3. Wersja czwarta kodu – PI4

Każdy wątek używa własnej prywatnej zmiennej *‘sum_private’* do obliczenia sumy częściowej. Zapewnia to, że każdy wątek niezależnie przetwarza swoją część danych.

Po zakończeniu pracy wszystkich wątków, ich wyniki są atomowo dodawane do zmiennej globalnej *‘pi_total’*, zapewniając poprawne scalenie wyników.

Liczba dostępów do pamięci jest nadal dość wysoka ze względu na konieczność synchronizacji i komunikacji między wątkami podczas dodawania wyników do zmiennej globalnej. Jednakże, zastosowanie zmiennych prywatnych zmniejsza konkurencję o dostęp do tej zmiennej, co zdecydowanie zmniejsza narzut synchronizacyjny w porównaniu do poprzedniej wersji kodu.

4. Wersja piąta kodu – PI5

Używając klauzuli *‘reduction(+:pi)’* przy dyrektywie *‘parallel for’*, każdy wątek automatycznie sumuje swoje lokalne sumy częściowe do zmiennej pi, co eliminuje potrzebę ręcznego scalania wyników.

Wykorzystanie klauzuli *‘reduction’* znacznie zmniejsza liczbę dostępów do pamięci w porównaniu do użycia dyrektywy *‘atomic’* lub zmiennej globalnej do zapisu wyników. Wartości są scalane lokalnie w każdym wątku, co minimalizuje konkurencję o dostęp do pamięci współdzielonej.

5. Wersja szósta kodu – PI6

Wyniki obliczeń są poprawne, ale czas przetwarzania jest wyższy niż w przypadku poprzednich wersji kodu. Spowodowane jest to występowaniem zjawiska *‘false sharing’*, co oznacza, że różne wątki modyfikują sąsiednie elementy tablicy. Prowadzi do unieważniania kopii linii pamięci podręcznej, co z kolei zwiększa czas przetwarzania.

Przetwarzanie wersji PI6 cechuje się gorszą wydajnością niż wersje PI4 i PI5 z powodu ww. zjawiska i konieczności sprowadzania do procesora aktualnych wersji linii danych pamięci podręcznej.

6. Wersja siódma kodu – PI7

W tej wersji dla każdej iteracji wykonujemy obliczenia na dwóch sąsiednich słowach tablicy. Przykładowo, w pierwszej iteracji oba wątki pracują na elementach tablicy tab[0] i tab[1], w drugiej iteracji na tab[1] i tab[2], itd. Dzięki temu, mamy kontrolę nad używanymi danymi i możemy zaobserwować, jak czas przetwarzania różni się między kolejnymi iteracjami.

Dzięki temu, że każdy wątek pracuje na dwóch sąsiednich słowach tablicy, ale nie na tych samych słowach, zapobiegamy występowaniu zjawiska *‘false sharingu’*. Słowa, na których operuje każdy wątek, są od siebie oddalone o co najmniej jedno słowo tablicy. W rezultacie, mimo że wątki mogą dzielić tę samą linię pamięci podręcznej procesora, to nie następuje konflikt między nimi, ponieważ operują na różnych danych.

Dlatego, w każdej iteracji, czas przetwarzania jest cyklicznie krótszy co 8 iterację, ponieważ wątki operują na danych, które znajdują się na różnych liniach pamięci podręcznej procesora, co minimalizuje występowanie false sharingu.

```
0.  0.185000 sekund
1.  0.190000 sekund
2.  0.187000 sekund
3.  0.182000 sekund
4.  0.204000 sekund
5.  0.028000 sekund
6.  0.142000 sekund
7.  0.165000 sekund
8.  0.139000 sekund
9.  0.146000 sekund
10.  0.204000 sekund
11.  0.170000 sekund
12.  0.163000 sekund
13.  0.026000 sekund
14.  0.170000 sekund
15.  0.201000 sekund
16.  0.184000 sekund
17.  0.188000 sekund
18.  0.174000 sekund
19.  0.160000 sekund
20.  0.206000 sekund
21.  0.025000 sekund
22.  0.178000 sekund
23.  0.185000 sekund
24.  0.183000 sekund
25.  0.167000 sekund
26.  0.148000 sekund
27.  0.196000 sekund
28.  0.206000 sekund
29.  0.036000 sekund
30.  0.166000 sekund
31.  0.198000 sekund
32.  0.202000 sekund
33.  0.203000 sekund
34.  0.169000 sekund
35.  0.164000 sekund
36.  0.165000 sekund
37.  0.057000 sekund
38.  0.181000 sekund
39.  0.156000 sekund
40.  0.167000 sekund
41.  0.187000 sekund
42.  0.170000 sekund
43.  0.185000 sekund
44.  0.163000 sekund
45.  0.027000 sekund
46.  0.189000 sekund
47.  0.139000 sekund
48.  0.164000 sekund
49.  0.194000 sekund
```