Grupa lab.
5

Data ćwiczenia
30.10.2023

Laboratorium 5

Imie i nazwisko
Łukasz Sawina

Obliczania sumy

Program prezentuje 3 sposoby na obliczenie sumy: pierwszy sekwencyjny, gdzie pętla przechodzi po wszystkich elementach tablicy i sumuje je do zmiennej, drugi wykorzystuje wątki do obliczania sumy z wykorzystaniem sekcji krytycznej (jedna zmienna globalna, do której są dodawane wyniki poszczególnych wątków) oraz trzecia wykorzystująca wątki bez sekcji krytycznej (dodatkowa tablica przechowująca wyniki poszczególnych wątków, które na końcu są sumowane w głównym programie).

Tab.1 Wyniki pomiarów czasu dla sumy liczonej sekwencyjnie, jednostka [s]

Sekwencyjnie			
O3		-g	
1000	1000000	1000	1000000
0,000002	0,119	0,000003	0,230
0,000002	0,117	0,000004	0,231
0,000004	0,120	0,000011	0,232
0,000002	0,121	0,000003	0,230
0,000004	0,119	0,000004	0,232
Średnia			
0,0000028	0,119	0,000005	0,231

Jak widać z powyższej tabeli wynik dla małego rozmiaru tablicy jest o wiele szybszy niż dla dużego, aż 42 500 razy szybszy, jest to spowodowane z oczywistych względów, ilość elementów jest aż 1000 krotnie większa. Dodatkowo różnica między wersją zoptymalizowaną a niezoptymalizowaną również się pojawia i wynosi ok. 2 krotności wyniku.

Tab. 2 Wyniki pomiarów czasu dla sumy liczonej równolegle, jednostka [s]

LW.	Równolegle				Równolegle			
	O3				g			
	1000		1000000		1000		1000000	
	mutex	no mutex	mutex	no mutex	mutex	no mutex	mutex	no mutex
2	0,000329	0,000089	0,0586	0,0581	0,000313	0,000047	0,1192	0,126
2	0,000312	0,000057	0,0586	0,0594	0,000320	0,000083	0,1206	0,118
2	0,000422	0,000080	0,0588	0,0590	0,000444	0,000079	0,1189	0,120
2	0,000322	0,000092	0,0586	0,0585	0,000296	0,000085	0,1189	0,117
2	0,000461	0,000082	0,0586	0,0590	0,000308	0,000091	0,1181	0,119
4	0,000657	0,000086	0,0353	0,0365	0,000303	0,000092	0,0635	0,080
4	0,000540	0,000088	0,0317	0,0327	0,000436	0,000084	0,0644	0,064
4	0,000507	0,000137	0,0314	0,0329	0,000468	0,000083	0,0640	0,064
4	0,000432	0,000109	0,0316	0,0314	0,000308	0,000087	0,0638	0,064
4	0,000346	0,000083	0,0313	0,0330	0,000446	0,000077	0,0638	0,064
8	0,000659	0,000195	0,0268	0,0283	0,000499	0,000200	0,0582	0,057
8	0,000510	0,000203	0,0294	0,0291	0,000512	0,000214	0,0577	0,055
8	0,000550	0,000215	0,0292	0,0293	0,000639	0,000208	0,0566	0,055
8	0,000597	0,000300	0,0285	0,0290	0,000606	0,000209	0,0599	0,056
8	0,000503	0,000226	0,0290	0,0279	0,000547	0,000173	0,0571	0,057
Średnia								
2	0,000369	0,000080	0,0586	0,0588	0,000336	0,000077	0,119	0,120
4	0,000496	0,000101	0,0323	0,0333	0,000392	0,000085	0,064	0,067
8	0,000564	0,000228	0,0286	0,0287	0,000561	0,000201	0,058	0,056

Jak widać z wykorzystaniem wątków do liczenia sumy otrzymujemy pewną poprawę czasu dla dużej tablicy, jednak dla mniejszej wręcz przeciwnie, operacja jest wykonywana dłużej. Fakt ten jest oczywisty, ponieważ przy wykorzystaniu wątków pomimo liczenia samej sumy dochodzi nam operacja tworzenia oraz oczekiwania na wątki, która zajmuje pewien czas. W wersji z sekcjami krytycznymi dodatkowo pojawia się blokowanie sekcji i oczekiwanie na jej zwolnienie, co jeszcze dodatkowo dokłada czas.

Tab. 3 Zestawienie średnich wartości dla sumy tablicy 1000 elementów, jednostka [s]

	mutex		no mutex		Sekwencyjne	
	O3	g	O3	g	O3	g
2	0,000369	0,000336	0,000080	0,000077	2,8E-06	0,000005
4	0,000496	0,000392	0,000101	0,000085		
8	0,000564	0,000561	0,000228	0,000201		

Przy zestawieniu średnich czasów dla tablicy 1000 elementów dokładnie widać, że wersja z wykorzystaniem wątków jest mniej korzystna, ponieważ wersja z mutexem jest ok. 50 razy wolniejsza od wersji sekwencyjnej. Bardzo dobrze to pokazuje, że wykorzsytanie wielu wątków do relatywnie małych obliczeń nie zawsze jest opłacalne, ponieważ nie zaoszczędzimy aż tyle czasu, aby zniwelować czas tworzenia wątków. Dodatkowo można zauważyć, że wzrost ilości wątków wcale nie skraca czasu, wręcz przeciwnie. Jest to znowu spowodowane dodatkowym czasem na tworzenie dodatkowych wątków.

Tab. 4 Zestawienie średnich wartości dla sumy tablicy 1 000 000 elementów, jednostka [s]

	mutex		no mutex		Sekwencyjne	
	O3	g	O3	g	O3	g
2	0,0586	0,119146	0,058783	0,120089		
4	0,0323	0,063896	0,033308	0,067020	0,119163	0,230874
8	0,0286	0,057883	0,028731	0,055860		

Dopiero przy większych tablicach widać zauważalną różnicę, ok. 2 krotne przyspieszenie liczenia sumy przy wykorzystaniu wątków. Przy tak dużej tablicy czas tworzenia wątków jest niwelowany przez zysk liczenia równoległego. W tym przypadku można również zauważyć poprawę czasów przy wykorzystaniu większej ilości wątków.

Obliczanie całki

Program służy do obliczenia całki z sin(x) w przedziale [0,PI], który liczy całkę numerycznie wykorzystując metodę trapezów. Przy uruchomieniu programu wymagane jest podanie wartości dx, z czego przy testach maksymalna wartość jaką można podać to 0.00000001, ponieważ przy jeszcze mniejszej wartości zwracane przez program to 0. Jest to najpewniej spowodowane błędami w precyzji liczb zmiennoprzecinkowych.

```
lukasz@lukasz-ThinkPad:~/C_programs/ProgramowanieRownolegle-AGH/Lab_5/Zad_2$ ./obliczanie_calki

Program obliczania całki z funkcji (sinus) metodą trapezów.

Podaj wysokość pojedynczego trapezu:  0.000001

Poczatek obliczeń sekwencyjnych
Obliczona liczba trapezów: N = 3141593, dx_adjust = 0.000001

Koniec obliczen sekwencyjnych
        Czas wykonania 0.073362.        Obliczona całka = 1.999999999999758

Poczatek obliczeń równoległych (zrównoleglenie pętli)
Obliczona liczba trapezów: N = 3141593, dx_adjust = 0.000001

Wątek 0: my_start 0, my_end 1570797, my_stride 1

Wątek 1: my_start 1570797, my_end 3141593, my_stride 1

Koniec obliczen  równoległych (zrównoleglenie pętli)
        Czas wykonania 0.038976.        Obliczona całka = 1.999999999999815

Poczatek obliczeń równoległych (dekompozycja obszaru)
0.000000
2.000000

Wątek 1: a_local 2.000000, b_local 3.141593, dx 0.000001
Obliczona liczba trapezów: N = 1141593, dx_adjust = 0.000001

Wątek 0: a_local 0.000000, b_local 2.000000, dx 0.000001
Obliczona liczba trapezów: N = 2000000, dx_adjust = 0.000001

Koniec obliczen  równoległych (dekompozycja obszaru)
        Czas wykonania 0.036194.        Obliczona całka = 1.999999999999725
```

Przy wywołaniu programu z wartością dx = 0.000001 otrzymujemy wyniki zbliżone do siebie z błędem mniejszym niż 10^{-12}.

Jak widać program zwraca trzy wyniki, są to trzy różne sposoby na wykonanie tych samych obliczeń. Pierwsze z nich podobnie jak w przypadku sumy jest wykonywane sekwencyjnie, całość jest liczona w jednej pętli. Naszym zadaniem było uzupełnienie programu o liczenie całki wykorzystując do tego wiele wątków. W moim przypadku to 2 wątków.

Dekompozycja pętli

W pierwszej kolejności chcieliśmy wykonać zwrównoleglenie pętli, wykonując to w dwóch różnych wariantach.

Cykliczne

Do funkcji liczącej całkę zostały przekazane wartości początku, końca, dx oraz ilość wątków. Te wartości zostały przypisane do zmiennych globalnych.

```
// tworzenie struktur danych do obsługi wielowątkowości
l_w_global = l_w;
int indeksy[l_w];
for(i=0;i<l_w;i++) indeksy[i]=i;

a_global = a;
b_global = b;
dx_global = dx_adjust
```

Jak widac zmienne zostały przypisane do globalnych zmiennych, dodatkowo utworzona została tablica indexów z numerami wątków.

W funkcji wątku następnie zostały one przypisane do zmiennych lokalnych

```
 int my_id = *( (int *) arg_wsk );

double a = a_global, b = b_global, dx = dx_global;
int N = ceil((b-a)/dx), l_w = l_w_global;

// dekompozycja cykliczna
int my_start = my_id;
int my_end = N;
int my_stride = l_w
```

Jak widać ustawione zostały wartości *my_start*, *my_end*, *my_stride*. Są to wartości, które będą odpowiadały za działanie pętli, zauważyć można, że początek oraz koniec pętli będą takie same jak w wersji sekwencyjnej. Różnica jest dopiero przy skoku, każdy wątek będzie przeskakiwał o *l_w* wartości dalej, czyli cyklicznie przechodził na kolejne elementy.

Następnie wykonywana jest pętla licząca całkę

```
int i;
double calka = 0.0;
for(i=my_start; i<my_end; i+=my_stride){
   double x1 = a + i*dx;
   calka += 0.5*dx*(funkcja(x1)+funkcja(x1+dx));
}

pthread_mutex_lock( &muteks );
calka_global += calka;
pthread_mutex_unlock( &muteks );
```

Jak widać każdy wątek będzie wykonywał liczenie wartości tylko dla określonych wartości i po obliczeniu dla wszystkich swoich skoków, wartość zostanie bezpiecznie zapisana do wartości globalnej.

Blokowe

W tej wersji zmiany pojawiają się dopiero przy określaniu wartości dla pętli. Teraz każdy wątek nie będzie kolejno brał elementów i je liczył, tylko każdy watek będzie miał własne miejsce startu oraz końca. Cała długość zostanie podzielona na bloki dla każdego wątku.

```
float j = ceil( (float)N/l_w);
int my_start = j*(my_id);
int my_end = j*(my_id+1);
int my_stride = 1;
if(my_end > N) my_end = N
```

Najpierw obliczany jest rozmiar bloku dla każdego wątku i kolejno obliczany początek oraz koniec. W tym przypadku *my_stride* ma wartość 1, ponieważ wątek musi przejść wszystkie iteracje w swoim bloku.

Przy liczeniu rozmiaru bloku wykorzystane jest zaokrąglanie w górę, jest to spowodowane tym, aby zabezpieczyć ostatni wątek przed dodatkowymi operacjami. Lepiej aby każdy wątek wykonywał dodatkowo jedną iterację niż ostatni wątek wykonał dodatkowo *l_w* operacji.

Na końcu można zauważyć, że w przypadku gdyby nasz przedział był niepodzielny przez ilość wątków, sprawdzamy czy wartość *my_end* wątku nie wychodzi poza zakres całkowania. W sytuacji gdy wychodzi jest to odpowiednio poprawiane.

Wykonanie pętli jest identyczne jak w poprzedniej wersji dlatego pomijam ten fragment kodu. Na końcu również wartości z każdego bloku zostają bezpieczenie przypisane do globalej zmiennej z wynikiem.

Dekompozycja obczaru

Kolejno zamiast zrównoleglać pętle, chcemy aby nasz wątek otrzymywał informację tylko o obszarze, który ma scałkował. W tym przypadku wątek nie wie jak wygląda cały obszar, zna tylko obszar przeznaczony dla niego.

Do tego programu potrzebna jest nam struktura danych dzięki, której prześlemy do wątku informację o przedziale jaki ma całkować

```
typedef struct CalkowaniePodobszar {
   int ID;
   double a;
   double b;
   double dx;
} CalkowaniePodobszar
```

W naszej strukturze przekazywany jest początek, koniec, długość fragmentu całkowania oraz ID wątku. (w strukturze nie ma zmiennej na wynik, ponieważ wykorzystałem zwracanie wyniku przez pthread_exit() jako opcję pozyskiwania wyniku).

Przy tworzeniu wątku obliczane są wartości poszczególnych zmiennych a następnie przesyłane są do wątku jako parametry.

```
int N = ceil((b-a)/l_w);
// tworzenie wątków
for(i=0; i<l_w; i++ )
{
   double my_start = a+N*(i);
   double my_end = a+N*(i+1);
   if(my_end > b) my_end = b;

   printf("%lf\n", my_start);

   podobszary[i].a = my_start;
   podobszary[i].b = my_end;
   podobszary[i].dx = dx;
   podobszary[i].ID = i;

   pthread_create( &watki[i], NULL, calka_podobszar_w, &podobszary[i]);
```

Obliczanie początku oraz końca przedziału jest identyczne jak dla dekompozycji blokowej, wszystkie podobszary zapisane są w dynamicznej tablicy, która na końcu jest uwalniana.

```
// rozpakowanie danych przesłanych do wątku
CalkowaniePodobszar* podobszar = (CalkowaniePodobszar *) arg_wsk;
double a_local = podobszar->a, b_local = podobszar->b, dx=podobszar->dx;
int my_id = podobszar->ID;
```

Przekazane wartości są następnie przypisywane do zmiennych lokalnych i na nich wykonywana jest ta sama funkcja co dla całkowania sekwencyjnego, jednak w tym przypadku obliczenia zostały zoptymalizowane.

```
int i;
double* calka = malloc(sizeof(double));
*calka = 0.0;
for(i=0; i<N; i++){
   double x1 = a_local + i*dx_adjust;
   *calka += (funkcja(x1)+funkcja(x1+dx_adjust));
}

*calka *= 0.5*dx_adjust;
pthread_exit((void*)calka);
```

Jak widać przed zwróceniem wyniku przy pomocy *pthread_exit()* wartość całki jest mnożona przez *0.5*dx_adjust.* W tej wersji aby zmniejszyć ilość mnożenia w programie, sama pętla liczy tylko sumy, a dopiero na końcu wynik jest mnożony przez tą wartość. W ten sposób zmniejszamy ilość mnożenia z N do 1, a jak wiemy mnożenie nie jest najszybszą operacją do wykonywania.

Dodatkowo wartość całki jest utworzona nie jako zmienna lokalna, a jako wskaźnik, jest to spowodowane tym, że naszą wartość będziemy chcieli zwrócić do głównego wątku, dlatego nie możemy jej zwrócić jako zwykła wartość, a jako adres.

```
// oczekiwanie na zakończenie pracy wątków
void* calka_podobszaru;
for(i=0; i<l_w; i++ ) {
   pthread_join( watki[i], &calka_podobszaru );
   calka_suma_local += *(double*)calka_podobszaru;
   free(calka_podobszaru);
```

Na końcu głównej funkcji wartości zwrócone z wątków są zapisywane do wskaźnika i następnie rzutowane na wartości double i zapisywane w zmiennej lokalnej przechowującej wynik całki. Ważne aby po każdym odczytaniu wartości zwolnić naszą pamięć.

W przypadku tego programu mogą pojawiać się różnice w wynikach z sekwencyjnym, jest to spowodowane obliczaniem wartości *dx_adjust*, która jest obliczana dla innych wielkości obszarów.

Wnioski

Wykorzystanie watków do przyspieszenia pewnych obliczeń jest bardzo przydatną umiejętnością, jednak jak pokazuje pierwsze zadanie z sumą, warto sprawdzić czy takie działanie będzie miało jakiś sens, ponieważ gdy obliczeń i danych będzie mało, będzie to nieopłacalne, przez dodatkowe straty czasu na tworzenie wątków, jednak przy długich, dużych obliczeniach taka operacja jak najbardziej jest przydatna.

Dodatkowo ważne jest, aby odpowiednio zadbać o podział naszych obliczeń. Przede wszystkim chcemy, aby były one jak najbardziej równo podzielone między wątkami. Z tego powodu do liczenia podziału blokowego wykonuje się zaokrąglanie w górę, gwarantuje to, że wszystkie wątki będą miały podobną ilość operacji, jedynie ostani może mieć troszkę mniej, ale jest to zawsze lepsze wyjście, niż aby czekać aż ostatni wątek wykona kilkanaście dodatkowych obliczeń.

Sposób w jaki wartości zostaną przekazane do wątków też jest ważne, najbezpieczniejszą wersją jest dekompozycja obszaru, czyli zamiast przekazywać całego obszaru do funkcji wątku i tam liczyć sobie jakie przedziały ma wykonywać, bezpieczniejsze jest przekazanie mu bezpośrednio przedziału jaki ma obliczyć. Pomijamy wtedy wykorzystanie wartości globalnych, które mogą być nieumyślnie zmienione i program nie będzie liczył prawidłowo. Dodatkowo w wersji z dekompozycją obszaru pomijamy wykorzystanie mutexów, wartości są zwracane bezpośrednio do głównego wątku i tam sumowane, nie ma potrzeby tworzenia globalnej wartości wyniku i zabezpieczania jej przy pomocy mutexów.