Sprawozdanie z zadania na Przetwarzanie Równoległe

Projekt 1

Sebastian Michoń 136770, Marcin Zatorski 136834

1 Wstęp

1. Sebastian Michoń 136770: grupa dziekańska L1
2. Marcin Zatorski 136834: grupa dziekańska L10
3. Wymagany termin oddania sprawozdania: 27.04.2020r.
4. Rzeczywisty termin oddania sprawozdania: 27.04.2020r.
5. Wersja I sprawozdania
6. Adresy mailowe: sebastian.michon@student.put.poznan.pl, marcin.r.zatorski@student.put.poznan.pl

2 Wykorzystywany system równoległy

1. Kompilator: gcc 7.5.0
2. System operacyjny: Ubuntu 18.04
3. Procesor Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz - 4 rdzenie, 2 wątki na 1 rdzeń: 8 procesorów logicznych i 4 fizyczne
4.

3 Użyte kody

1. 01_erasto_single.cpp - Kod sekwencyjny, standardowe sito erastotenesa działające w $O(r * loglog(r))$, z podwójną optymalizacją: do odsiewania używane są tylko liczby pierwsze, ponadto odsiew rozpoczyna się od kwadratu danej liczby - jest to poprawne, ponieważ jeśli liczba nie jest pierwsza to jej najniższy dzielnik wyższy od 1 jest mniejszy równy jej pierwiastkowi.

Listing 1: Sito Erastotenesa

```
for (i=2;i*i<=n;i++){
        if (res[i]==0){
                for (j=i*i;j<=n;j+=i) res[j]=1;
        }
}
```

Celem kodu jest jedynie pokazanie koncepcji; nie zachodzi wyścig ani nie ma synchronizacji, ponieważ jest jeden wątek.

2. 02_most_primitive.cpp - Kod sekwencyjny, który szuka dzielnika liczby pośród liczb mniejszych równych jej pierwiastkowi. Rozwiązanie to dizała w złożoności $O((r-l)*\sqrt{r})$. Sprawdzałem podzielność także dla liczb, które nie są pierwsze, aby nie używać żadnych tablic poza tablicą znalezionych liczb pierwszych - celem jest pokazanie kodu wykorzystującego w jak najmniejszym stopniu tablice.

Listing 2: Rozwiązanie pierwiastkowe

```
for (i=2;i<=n;i++){
        for (j=2;j*j<=i;j++){
                if (i%j==0) {
                        res[i]=1;
                        break;
                }
        }
}
```

3. 03_erasto_functional_static_schedule.cpp - kod równoległy, koncepcja sita, podejście funkcyjne. Funkcja najpierw wyznacza rekursywnie wszystkie liczby pierwsze mniejsze równe pierwiastkowi z docelowego rozmiaru sita, następnie sama poszukuje liczb pierwszych mniejszych równych zadanej liczbie. Kluczowa część algorytmu wygląda tak: (gdzie res[i]==0 oznacza liczbę pierwszą):

Listing 3: Sito funkcyjne ze static schedulingiem

```
#pragma omp parallel for
        for (i=0;i<=sq;i++){
                if (res[i]==1) continue;
                for (int j=i*i; j<=n; j+=i) res[j]=1;
        }
```

Gdzie res[i]==0 oznacza liczbę pierwszą, a *sq* oznacza $\lfloor(\sqrt{n})\rfloor$. Własności tego kodu:

(a) Dyrektywa powyżej tworzy zbiór wątków, którym przydziela w przybliżeniu równy podzbiór iteracji do wykonania; co istotne, przy tak sformułowanym kodzie wątek będzie wykonywał kolejne iteracje - na przykład 1. wątek może wykonać je dla i=2, 3, 5, 7, a 2. wątek dla 11, 13, 17, 19. Jest to nieefektywne, ponieważ procesy dostaną taką samą ilość liczb, którymi będą odsiewać liczby sita, a proces, który dostanie liczby najmniejsze wykona najwięcej operacji: w powyższym przypadku, 1. wątek wykona nieco mniej niż $\frac{n}{2}+\frac{n}{3}+\frac{n}{5}+\frac{n}{7}$ operacji oznaczenia liczby (gdzie n to rozmiar sita - "nieco mniej" wynika z tego, że nie odznaczam liczb $x<\sqrt{n}$)), a 2. wątek $\frac{n}{11}+\frac{n}{13}+\frac{n}{17}+\frac{n}{19}$ - czyli dużo mniej.

(b) Nie zachodzi wyścig (rozumiany jako zależność działania programu od kolejności wykonywania wątków), ponieważ wątki modyfikują tylko część tablicy, która nie jest używana do znajdowania liczb pierwszych, ponadto jeśli zmieniam wartość jakiegoś elementu tablicy, w której zaznaczam liczby pierwsze, to mogę tylko oznaczyć liczbę jako pierwszą; co za tym idzie, jeśli liczba zostanie oznaczona przez kilka wątków jako liczba pierwsza, to niezależnie od tego, który z nich oznaczy ją jako pierwszy, który później nie ma żadnego znaczenia z punktu widzenia występowania wyścigu.

(c) Synchronizacja zachodzi tylko na końcu pętli for, nie powinna mieć istotnego wpływu na czas obliczeń - wywołania rekursywne wykonają się relatywnie szybko, bo suma rozmiarów sit, które będą w nich wypełniane jest nie większa niż $2*\sqrt{n}$ - można to pokazać przez $\sqrt[2]{n}+\sqrt[4]{n}+..+k \leq \sqrt{n}+\frac{\sqrt{n}}{2}+\frac{\sqrt{n}}{4}+..+\frac{\sqrt{n}}{2^{\lfloor log_2(n)\rfloor}} \leq 2*\sqrt{n}$, gdzie $k \leq 4$ - warunek początkowy rekursji, zaś czas wykonania całego sita i tak jest ograniczony przez czas wykonania najwolniejszego procesu w pierwszym wywołaniu funcji (nierekursywnym).

(d) False sharing zachodzi w szczególnym przypadku, gdy sprawdzam pod kątem pierwszości liczbę x, razem z nią ściągając do cache część ciągu, która może się zmieniać, ponieważ $|x - sqrtn| < 64$ (64 bajty to rozmiar linii pamięci, a rozmiar typu bool to 1 bajt). Może on jednak zajść nie więcej niż $64 * log_2(n)$ razy, bo $\sqrt[log_2(n)]{n} \leq 4$ - jest to liczba o kilka rzędów wielkości mniejsza niż n, zatem (co pokaże później VTune profiler) false sharing nie będzie istotnie wpływał na czas przetwarzania.

4 Tablica wynków: kody od Pi2 do Pi6 w 3 wersjach

Kod	Wątki	Rzeczywisty czas obliczeń	Czas użycia procesorów	Przysp.	Pi
./pi_s.c	1	11.106908	11.106908	-	3.141592653590
./pi2.c	2	9.854029	19.660201	1.12714383	1.629332922363
./pi2.c	4	6.201361	22.991912	1.79104361	0.606812628416
./pi2.c	8	4.855977	37.927782	2.28726536	0.407415018512
./pi3.c	2	32.231695	61.883320	.344595839	3.141592653590
./pi3.c	4	68.859282	269.248873	.161298632	3.141592653590
./pi3.c	8	102.148402	721.992069	.108733056	3.141592653590
./pi4.c	2	5.644930	11.230873	1.96759003	3.141592653590
./pi4.c	4	2.862752	11.442671	3.87980097	3.141592653590
./pi4.c	8	1.489600	11.794959	7.45630236	3.141592653590
./pi5.c	2	5.616025	11.189766	1.97771697	3.141592653590
./pi5.c	4	2.865750	11.450181	3.87574212	3.141592653590
./pi5.c	8	1.493857	11.804175	7.43505435	3.141592653590
./pi6.c	2	6.310614	12.557454	1.76003602	3.141592653590
./pi6.c	4	4.567762	15.864548	2.43158640	3.141592653590
./pi6.c	8	3.467091	25.773811	3.20352364	3.141592653590

gdzie Czas użycia procesorów jest sumaryczny, a Przysp. to skrót od przyspieszenia kodu równoległego względem sekwencyjnego.

1. Kod pi2.c różni się od kodu sekwencyjnego dodaniem dyrektywy

   ```
   #pragma omp paralel for
   ```

   dla pętli obliczającej pi. Zarówno x jak i *sum* są współdzielone przez wątki; błędny wynik wynika z wyścigu w dostępie do danych (który zachodzi np. jeśli jeden wątek zmieni wartość x-a w momencie w którym 2. wątek dodaję wartość zależną od x-a do sumy - jeśli 1. zapisze dane przed 2. wątkiem, ten 2. będzie korzystał z niepoprawnych danych), który zachodzi, ponieważ nie ma sychronizacji w dostępie do zmiennych współdzielonych. Sumaryczne czasy użycia procesorów są wyższe niż dla kodu sekwencyjnego, ponieważ nie zachodzi czasowa lokalność odwołań - dane muszą być zmieniane w pamięci po każdej zmianie sumy, ponadto jeśli x został zmieniony pomiędzy zapisem x-a a zmianą sumy przez wątek, x też zostanie zmieniony, choć nie powinien (jeśli algorytm ma działać poprawnie).

2. Kod pi3.c ma lokalną dla wątku zmienną x i zapis do sumy następuje z użyciem dyrektywy:

   ```
   #pragma omp atomic
   ```

   Co za tym idzie, zmienna współdzielona *sum* jest uaktualniana w punkcie czasu przez dokładnie 1 wątek i nie zachodzi wyścig w dostępie do danych. Kod nie jest efektywny, ponieważ wykonuje 10^9 razy operację założenia i zdjęcia zamka(blokady) na współdzielonej zmiennej

sum (przy czym najpewniej zachodzą jakieś optymalizacje, np. zamek jest zwalniany w momencie, gdy inny wątek chce zapisać do sumy, a nie bezpośrednio po jej aktualizacji - to by wyjaśniało dlaczego kod używający 2 wątków jest szybszy niż ten używający 8 wątków).

3. Kod pi4.c różni się od pi3.c tym, że każdy wątek zapisuje i odczytuje lokalną dla wątku zmienną, po zakończeniu obliczeń zmienia atomowo sumę; nie zachodzi wyścig, ponieważ zmienne x i $s2$ (suma dla pojedynczego wątku) są lokalne dla wątku, a zmienna *sum* jest aktualizowana w punkcie czasu przez co najwyżej jeden wątek, ponieważ jest aktualizowana atomowo. Lokalne zmienne x i $s2$ znajdują się w pamięci procesorów, które na nich operują i nie są zmieniane i odczytywane przez inne wątki, co implikuje czasową lokalność odwołań i efektywność: Sumaryczny czas użycia procesorów jest prawie taki sam jak kodu sekwencyjnego, użycie większej liczby procesorów prowadzi do przyspieszenia rzeczywistego wykonania kodu tyle razy, ile jest procesorów logicznych.

4. Kod pi5.c jest podobny do kodu pi4.c, ale zamiast lokalnych zmiennych sumy jest tam:

```
#pragma omp parallel for private(x) reduction(+:sum)
```

   co działa nieomal dokładnie w ten sam sposób co pi4.c, ponieważ dyrektywa tworzy zmienną lokalną dla wątku, która jest aktualizowana zamiast współdzielonej *sum* w pętli i dodawana do zmiennej globalnej dla wątków dopiero po zakończeniu działania w pętli przez wątek. Co za tym idzie, czasowo ten algorytm funkcjonuje prawie tak samo jak pi4.c.

5. Kod pi6.c różni się od pi4.c tym, że zamiast pojedynczych zmiennych lokalnych do sumowania została użyta tablica typu double[] taka, że każdy wątek aktualizuje tylko 1 indeks - co za tym idzie, zachodzi zjawisko false sharingu - zapisy do tablicy w 1 wątku powodują konieczność zmiany całej linii pamięci w innych pamięciach podręcznych procesorów, ponieważ te mają w swoich liniach pamięci własne kopie kolejnych wartości tej tablicy. Zmniejsza to przestrzenną lokalność odwołań i efektywność kodu, sumaryczny czas użycia procesorów rośnie wraz ze zwiększeniem liczby wątków - a co za tym idzie, więcej razy zachodzi pobieranie linii do pamięci podręcznych procesora, ponieważ więcej wątków aktualizuje i odczytuje dane z tablicy.

5 Długość linii

Kod pi7.c wypisywał najniższe wartości czasu (niecałe 5.6 s) dla 2 wątków w iteracjach k takich, że

$$k \equiv 7 \bmod 8$$

Przy czym w k-tej iteracji używałem wartości tablicy o indeksach k i $k+1$, a iteracje indeksowałem od 0: co za tym idzie, długość linii pamięci podręcznej procesora, którego używałem to 64 bajty, ponieważ na moim systemie rozmiar chara to 1 bajt, a:

$$sizeof(double) == 8$$

Czyli rozmiar typu double to 8 rozmiarów chara czyli 8 bajtów. Co za tym idzie, rozmiar linii pamięci w moim procesorze to $8 * 8 = 64$ bajty

6 Wyjaśnienie, trudności

1. W 0. iteracji korzystam z indeksów (0, 1) tablicy *double*[]; w 1. z indeksów (1, 2), ... w 7. z indeksów (7, 8) i tak dalej.

2. W 25 pierwszych iteracjach najniższe wyniki czasowe uzyskuje w 7., 15. i 23. iteracji. Można to logicznie uzasadnić, jeśli linia pamięci podręcznej procesora ma 8 bajtów: w 0., 1., ... 6. iteracji 2 procesory nadpisują wzajemnie własne linie pamięci podręcznej, bo pobierają za każdym razem kolejne 8 doubli z tablicy *double*[] (0...7) - co za tym idzie, zmiana w tej tablicy wymusza zmianę w linii pamięci podręcznej 2. procesora.

3. W 7. iteracji 1. procesor ma w swojej linii pamięci elementy tablicy *double*[] o indeksach 0...7, 2. procesor ma w swojej linii pamięci elementy tablicy *double*[] o indeksach 8...15. 2 procesory nie nadpisują swoich linii pamięci, co przyspiesza działanie programu. Analogicznie, w 15. iteracji 1. procesor ma w swojej linii pamięci elementy tablicy *double*[] o indeksach 8...15, 2. procesor - indeksy 16...23. To zjawisko powtarza się cyklicznie co 8 bajtów. Nie widzę innego sensownego uzasadnienia, dlaczego właśnie 7., 15. i 23. i kolejne co ósme iteracje dawały najlepsze rezultaty, dlatego przyjmuje powyższe rozumowanie za co najmniej zasadne, szczególnie zważając na to, że linia pamięci podręcznej mojego procesora ma 64 bajty zgodnie z jej dokumentacją.

4. Gdy zamiast kolejnych wartości tablicy używałem wartości tablicy o indeksach, których różnica wynosi 20 dla 2 wątków, wyniki czasowe iteracji były takie same jak dla co 8. iteracji używając kolejnych elementów tablicy, co wzmacnia tezę o tym, że przestrzenna lokalność odwołań ma wpływ na długość obliczeń i to, że linia pamięci faktycznie ma 8 bajtów.

5. Jedyną trudnością były zdarzające się iteracje, w których procesor delikatnie zwalniał, zapewne w wyniku procesów w tle, po wyeliminowaniu których otrzymałem powyższe rezultaty. Różnice pomiędzy 7., 15., 23. ... a pozostałymi iteracjami były rzędu 0.15-0.4 sekundy, dlatego zasadnym było albo wydłużenie pętli, albo wyeliminowanie zbędnych procesów w tle.