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.

```
1  for (i=2;i*i<=n;i++){
2    if (res[i]==0){
3      for (j=i*i;j<=n;j+=i) res[j]=1;
4    }
5  }
6
```

Listing 1: Sito Erastotenesa

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.

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

```

Listing 2: Rozwiązanie pierwiastkowe

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):

```
#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;
  }

```

Listing 3: Sito funkcyjne ze static schedulingiem

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 pętli 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. 04_erasto_functional_handmade_scheduling.cpp - kod równoległy, koncepcja sita, podejście funkcyjne. Kod jest analogiczny do poprzedniego, ale iteracje są ręcznie przypisywane do każdego wątku - Najpierw wyliczam sumę $sum = \frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + ... + \frac{n}{np}$, gdzie np to najwyższa liczba pierwsza $np \leq \sqrt{n}$ i średnią liczbę operacji które mają być wykonywane przez proces: $psum = \frac{sum}{proc\_number}$, gdzie $proc\_number$ to liczba procesów. Następnie każdemu procesowi przydzielam w osobnej tablicy liczby pierwsze, którymi będzie skreślał elementy tablicy w ten sposób, że:

   (a) i-ty proces dostanie i-tą liczbę pierwszą

   (b) następnie proces będzie dobierał sobie wolne liczby pierwsze począwszy od np do momentu, w którym jego szacowana liczba operacji nie przekroczy $psum$

   Kod, który wykonuje takie operacje jest przedstawiony tutaj:

```
22  int beg=0, end=sq;
23  double partsum=0;
24  for (i=0;i<proc;i++){
25    ij[i]=0;
26    partsum=0;
27    for (j=beg;j<=end;j++){
28      if (res[j]==0) {
29        squarez[i][ij[i]]=j, partsum+=1.0/j, ij[i]++;
30        break;
31      }
32    }
33    beg=j+1;
34    if (partsum<part){
35      for (j=end;j>=beg;j--){
36        if (res[j]==0) {
37          squarez[i][ij[i]]=j, partsum+=1.0/j, ij[i]++;
38        }
39        if (partsum>=part) break;
40      }
41      end=j-1;
42    }
43  }
44
```

Listing 4: Sito funkcyjne z ręcznym schedulingiem

   Dzięki powyższemu rozwiążę problem przedstawiony w części (a) poprzedniego rozwiązania - procesy będą względnie równo dzielić się pracą bez dodatkowego narzutu związanego z synchronizacją, przy czym procesy, które wezmą najniższe liczby pierwsze (2, 3, czasem 5) nadal wykonają największą pracę, ponieważ dla $n < 500000000 \wedge psum = 8$ zachodzi $\frac{n}{2} \geq psum$) - Analogicznie dla 3. Pozostałe części rozwiązania - (b), (c), (d) są identyczne dla tego kodu

5. 05_erasto_functional_dynamic_schedule.cpp - działa tak jak kod (3), ale używa innej dyrektywy:

```
45  #pragma omp parallel for schedule(dynamic)
46    for (i=0;i<=sq;i++){
```

```
      if (res[i]==1) continue;
      for (int j=i*i; j<=n; j+=i) res[j]=1;
    }

```

Listing 5: Sito funkcyjne z dynamic schedulingiem

Dzięki zmianie dyrektywy na schedule(dynamic) wątek po zakończeniu swojej pracy wykonuje część (blok) pracy innego wątku zamiast niego - dzięki temu wątki będą się dzieliły równo pracą, natomiast w porównaniu z kodem (3) dojdzie narzut związany z koniecznością synchronizacji wątków. Rozwiązanie zadania (3) używało domyślnej klauzuli schedule(static).

6. 07_erasto_domain.cpp - kod równoległy, koncepcja sita, podejście domenowe. Każdy wątek, znając swój numer, używając całej tablicy do $\sqrt{n}$ włącznie oznacza wszystkie liczby podzielne przez daną liczbę pierwszą większe niż $\sqrt{n}$.

```
  #pragma omp parallel
  {
    int left=(n/thr)*omp_get_thread_num(), i, j, based_left;
    int right=left+n/thr-1;
    if (omp_get_thread_num()==thr-1) right=n;
    if (left<=sq) left=sq+1;

    for (i=0;i<=sq;i++){
      if (res[i]==0){
        based_left=left-left%i+((left%i==0)?0:i);
        for (j=based_left;j<=right;j+=i){
          res[j]=1;
        }
      }
    }
  }

```

Listing 6: Sito funkcyjne z dynamic schedulingiem

Kod ten ma 3 zasadnicze różnice w porównaniu z kodem (3) w kontekście współbieżności:

(a) Dyrektywa tworzy wątki, które podzielą się nieomal równomiernie pracą - ponieważ każdy wątek musi użyć każdej liczby pierwszej z części domkniętej sita do oznaczenia przedziału z części otwartej sita o wielkości (prawie) równej dla każdego wątku.

(b) False sharing zachodzi w szczególnym przypadku, gdy sprawdzam pod kątem pierwszości liczbę x - albo ją odnaczam jako pierwszą, razem z nią ściągając do cache liczbę y w części otwartej używanej przez inny wątek, która może się zmieniać, ponieważ $|x - y| < 64$ (64 bajty to rozmiar linii pamięci, a rozmiar typu bool na systemie, na którym zaszło testowanie to 1 bajt). Może on jednak zajść nie więcej niż $(8 + 1) * 64 * log_2(n)$ razy, bo $\sqrt[log_2(n)]{n} \leq log_2(n)$, a liczba wątków to co najwyżej 8, dodatkowe +1 wynika z wątku używającego podciągu obok tablicy z wyznaczonymi pierwszymi - 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.

(c) Wątki będą modyfikowały współdzielone L2 i L3 cache - co za tym idzie, często będą zachodziły cache-missy, ponieważ L2 i L3 cache będą często zmieniały dane - w praktyce każdy wątek będzie modyfikował zupełnie inne części tablicy, które będą stale się zmieniać (inaczej niż np. w przypadku, w którym 1 wątek modyfikuje co 2. element tablicy).

7. 08_sqrt_functional.cpp - kod równoległy, koncepcja dzielenia, podejście funkcyjne. Każdy wątek znając swój numer wyznacza 2 liczby: left i right, oznaczające przedział, z którego będę brał liczby i sprawdzał podzielność danej liczby przez jedną z nich

```
int sq=floor(sqrt(n)), vv;
vv=(sq-2)/thr;
omp_set_num_threads(thr);
#pragma omp parallel
{
  int wisdom=omp_get_thread_num(), j, i;
  int v1=2+vv*wisdom, v2=2+vv*(wisdom+1)-1;
  if (wisdom==thr-1) v2=sq;

  for (i=2;i<=n;i++){
    for (j=v1;j*j<=i && j<=v2;j++){
      if (i%j==0) {
      res[i]=1;
      break;
    }
  }
}

```

Listing 7: Sito funkcyjne z dynamic schedulingiem

Własności tego kodu:

(a) Wszystkie wątki dostaną względnie równy zbiór liczb pierwszych do sprawdzenia - to wynika ze sposobu wyznaczenia lewego i prawego końca przedziału

(b) W kodzie tym nie zachodzą wyścigi, bo wyznaczanie dzielnika w 1 wątku i ewentualne oznaczenie liczby jako złożona jest niezależne od działania innych wątków.

(c) Jedyna synchronizacja zajdzie na końcu pętli for.

(d) False sharing może zajść dla całej tablicy liczb pierwszych, ponieważ wątki ją modyfikują niezależnie od siebie.

Fundamentalnym problemem algorytmu jest nonsensowny algorytm - każda liczba jest dzielona przez liczby do pierwiastka z niej niezależnie od tego, czy znalazłem jej dzielnik w innym wątku. To zapewnia mniejszy narzut związany z synchronizacją, jednocześnie generując problemy związane z nieefektywnością.

8. 09_sqrt_domain.cpp - kod równoległy, koncepcja dzielenia, podejście domenowe. Każdy wątek używa całego zbioru otwartego do odznaczania własnego podzbioru zbioru otwartego wyznaczanego przez dyrektywę #pragma omp parallel for

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

```

Listing 8: Sito funkcyjne z dynamic schedulingiem

Własności tego kodu:

(a) Wszystkie wątki dostaną względnie równy podzbiór zbioru liczb do sprawdzenia o podobnym rozkładzie najniższych dzielników.

(b) W kodzie tym nie zachodzą wyścigi, ponieważ każdy wątek szuka dzielników innych liczb (i ewentualnie oznacza je jako pierwsze - jest to jedyna modyfikacja współdzielonej zmiennej przez wątek).

(c) Jedyna synchronizacja zajdzie na końcu pętli for.

(d) False sharing może zachodzić tylko na stykach podzbiorów zbioru otwartego modyfikowanych przez 2 wątki.

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.