Sprawozdanie z zadania na Przetwarzanie Równoległe

Projekt 1

Sebastian Michoń 136770, Marcin Zatorski 136834

1 Wstęp

1. Sebastian Michoń 136770: grupa dziekańska L1 - grupa poniedziałkowa, 8:00
2. Marcin Zatorski 136834: grupa dziekańska L10 - grupa środowa, 13:30
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
7. Celem realizowanego zadanie jest współbieżne wyznaczanie liczb pierwszych i badanie wydajności kodu w zależności od rozwiązania - domenowego i funkcyjnego.

2 Wykorzystywany system równoległy

1. Kompilator: icpc (Intel C++ compiler) 19.1.1.217
2. Do zdobycia informacji na temat przetwarzania używałem Intel VTune'a
3. System operacyjny: Ubuntu 18.04
4. Procesor Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz - 4 rdzenie, 2 wątki na 1 rdzeń: 8 procesorów logicznych i 4 fizyczne. Do pomiarów został wyłączony HyperThreading, co za tym idzie używałem co najwyżej 4 procesorów logicznych. Linia pamięci podręcznej procesora ma 64 bajty. Pamięc cache ma:
   (a) L1: 256 kilobajtów
   (b) L2: 1024 kilobajty
   (c) L3: 8192 kilobajty

   Pamięc L2 i L3 jest dostępna dla wszystkich procesorów, L1 jest lokalna dla procesora.

3 Używane oznaczenia

1. l, r oznaczają kolejno lewy i prawy koniec przedziału, w którym mają zostać wyznaczone liczby pierwsze.
2. **Domknięta część sita** to taka część elementów zrównoleglonego sita, w której liczby oznaczono jako pierwsze/złożone i do której nie zostanie dokonany zapis w trakcie wykonywania sita. W szczególności, aby wypełnić równolegle sito o rozmiarze n, jego część domknięta musi zawierać wszystkie liczby z przedziału $< 0; \lfloor\sqrt{n}\rfloor >$.

3. **Otwarta część sita** to taka część elementów zrównoleglonego sita, w której liczby nadal mogą zostać odznaczone jako złożone.

4. W kodach jeśli res[i]==1 to i jest liczbą złożoną.

5. Część kodów w sprawozdaniu została uproszczona względem oryginału - tutaj moim celem jest pokazanie koncepcji.

4 Użyte kody

1. 01_erasto_single.cpp - Kod sekwencyjny, standardowe sito erastotenesa działające w $O(r * loglog(r))$, z podwójną optymalizacją: do odznaczania liczb złożonych używane są tylko liczby pierwsze (np. nie używam 6, aby odznaczyć 36, 42.. etc., ponieważ te zostały już odznaczone przez każdy pierwszy dzielnik 6 - czyli 2 i 3), ponadto odsiew rozpoczyna się od kwadratu danej liczby - jest to poprawne, ponieważ jeśli liczba x nie jest pierwsza to jej najniższy dzielnik $d > 1$ spełnia $d \leq \sqrt{x}$.

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

```

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 d liczby x pośród liczb mniejszych równych jej pierwiastkowi: $d \leq \sqrt{x}$. Rozwiązanie to działa 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, co za tym idzie mającego niewielkie narzuty związane z dostępem do pamięci.

```
for (i=l;i<=r;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 $p \leq \sqrt{n}$, gdzie n to docelowy rozmiar sita, następnie sama poszukuje liczb pierwszych $p \leq n$. Kluczowa część algorytmu wygląda tak:

```
#pragma omp parallel for
for (i=0;i<=sq;i++){
  if (res[i]==1) continue;
  int sv=sq+1;
  int j=sv-sv%i+((sv%i==0)?0:i);
```

```
    for (j=max(i*i, j); j<=n; j+=i) res[j]=1;
  }

```

Listing 3: Sito funkcyjne ze static schedulingiem

Gdzie sq oznacza $\lfloor(\sqrt{n})\rfloor$, a sv to najmniejsza liczba większa równa $sq+1$ i i^2 podzielna przez i. Własności tego kodu:

(a) Dyrektywa powyżej tworzy zbiór wątków, którym przydziela w przybliżeniu równy podzbiór części domkniętej sita; 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 z otwartej części sita, a proces, który dostanie najmniejsze liczby 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 otwartą sita, ponadto jeśli zmieniam wartość jakiegoś elementu tablicy, w której zaznaczam liczby pierwsze, to mogę tylko oznaczyć liczbę jako złożoną; co za tym idzie, jeśli liczba zostanie oznaczona przez kilka wątków jako liczba złożona, to niezależnie od tego, który z nich oznaczy ją jako pierwszy, który później nie zajdzie wyścig. Nie zmieniam w pętli żadnej zmiennej globalnej poza tablicą pierwszości.

(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 może zajść, gdy aktualizuję liczbę znajdującą się w cache po modyfikacji zmiennej w pobliżu w tablicy pierwszości przez inny wątek. Taka sytuacja może zajść przez cały czas działania sita, ponieważ wszystkie wątki mogą aktualizować całą tablicę pierwszości; także w szczególnym przypadku, gdy sprawdzam pod kątem pierwszości liczbę x z części domkniętej sita, razem z nią ściągając do cache fragment części otwartej sita, ponieważ $|x - \lfloor\sqrt{n}\rfloor| < 64$ (64 bajty to rozmiar linii pamięci, a rozmiar typu bool na używanym sprzęcie to 1 bajt).

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 $sum = \frac{1}{2}+\frac{1}{3}+\frac{1}{5}+\frac{1}{7}+...+\frac{1}{np}$, gdzie np to najwyższa liczba pierwsza $np \leq \sqrt{n}$ i średnią liczbę operacji zmniejszoną n-krotnie, które mają być wykonywane przez proces: $partsum = \frac{sum}{proc}$, gdize $proc$ 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 nieprzydzielone liczby pierwsze począwszy od np do momentu, w którym jego szacowana liczba operacji nie przekroczy $psum$

Kod odpowiadający za przydział liczb pierwszych do tablicy procesu:

```
  for (i=0;i<=sq;i++){
    if (res[i]==0) summa+=1.0/i;
  }
  part=summa/proc;

  int beg=0, end=sq;
  double partsum=0;
  for (i=0;i<proc;i++){
    ij[i]=0;
    partsum=0;
    for (j=beg;j<=end;j++){
      if (res[j]==0) {
        squarez[i][ij[i]]=j, partsum+=1.0/j, ij[i]++;
        break;
      }
    }
    beg=j+1;
    if (partsum<part){
      for (j=end;j>=beg;j--){
        if (res[j]==0) {
          squarez[i][ij[i]]=j, partsum+=1.0/j, ij[i]++;
        }
        if (partsum>=part) break;
      }
      end=j-1;
    }
  }

```

Listing 4: Ręczny scheduling sita funkcyjnego

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 wykonałyby największą pracę, ponieważ dla $n < 500000000 \wedge proc = 8$ nadal zachodzi $\frac{1}{2} \geq psum$) - Analogicznie dla 3. Problemem z tą heurystyką jest nieuwzględnienie cache missów: odznaczanie liczb złożonych za pomocą liczby 2 będzie miało dużo rzadziej cache missa niż odznacznie liczb złożonych za pomocą za pomocą liczby 8387, a następnie kolejnych 10 liczb pierwszych, za każdym razem wymieniając dane w cache. 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:

```
  #pragma omp parallel for schedule(dynamic)
    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 pierwszości do $\sqrt{n}$ włącznie oznacza wszystkie liczby

podzielne przez daną liczbę pierwszą większe niż $\sqrt{n}$, które należą do przedziału unikalnego dla danego procesu, wyznaczonego za pomocą jego numeru.

```
#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 złożoną, 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 $(4 + 1) * 64 * log_2(n)$ razy, bo $\sqrt[log_2(n)]{n} \leq log_2(n)$, a liczba wątków to co najwyżej 4, dodatkowe +1 wynika z wątku używającego podciągu obok tablicy z części domkniętej sita - jest to liczba o kilka rzędów wielkości mniejsza niż n, zatem (co pokaże później VTune profiler) false sharing nie będzie prawie wcale 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), a nie zmieszczą się one w L1 cache (mogącej pomieścić 32KB danych).

7. 08_erasto_super_domain.cpp - usunięty został problem częstych cache missów przez to, że wątek aktualizuje w 1 pętli co najwyżej 32000 kolejne elementy sita plus kilka dodatkowych; W moim algorytmie utrzymuję tablicę liczb pierwszych modyfikowaną przed stworzeniem wątków wartościami $x \leq \sqrt{n}$; Każdy wątek aktualizuje kolejne liczby pierwsze od poprzedniej oznaczonej albo startowej, dopóki nie przekroczy $left + 32000$ albo prawej granicy podzbioru, który wolno mi modyfikować. ($left$ to początek iteracji), wtedy jest to jego ostatnia iteracja w tym batchu; przechodzę do kolejnej liczby powtarzając proces; w końcu, po przejściu całej tablicy liczb pierwszych, wracam do 1. liczby pierwszej i aktualizuje $left+ = 32000$. Powtarzam proces dopóki każda liczba pierwsza nie przekroczy prawej granicy zadanego ciągu.

```
ineo=0;
for (int vi=0;vi<=sq;vi++){
  if (res[vi]==0) {
    for (ite=0;ite<10;ite++) dp[ite][vi]=0;
    neoprimez[ineo]=vi, ineo++;
```

```
    }
  }

  #pragma omp parallel
  {
    int thnum=omp_get_thread_num(), allth=omp_get_num_threads();
    int left=a+((n-a)/allth)*thnum, i, ji, j, finished=0, curfun=left;
    int right=left+(n-a)/allth-1;
    if (thnum==allth-1) right=n;
    if (left<=sq) left=sq+1;

    while (finished<ineo){
      curfun=curfun+outer;
      for (ji=0;ji<ineo;ji++){
        i=neoprimez[ji];
        if (dp[thnum][i]==0) dp[thnum][i]=max(modal(left, i), modal(a, i));
        for (j=dp[thnum][i]; j<=right;j+=i){
          if (j>=curfun) break;
          p[j]=1;
        }
        if (j>right) finished++;
        dp[thnum][i]=j;
      }
    }
  }
```

Listing 7: Sito funkcyjne z dynamic schedulingiem

Jedyną fundamentalną różnicą pomiędzy tym kodem a zwykłym sitem domenowym jest unikanie cache missów i dążenie do trzymania w L1 cache jak największej części aktualnie używanego sita. Poza tym złożoność może się zwiększyć, a przynajmniej nie jestem w stanie dowieść standardowej złożoności sita. To rozwiązanie pozwala efektywnie zrównoleglić zadanie uzyskując CPI=0.331 i czas przetwarzania 0.4s dla $l = 1, r = 10^9$ na 4 rdzeniach, co pokażą wyniki VTune'a.

8. 09_sqrt_domain.cpp - kod równoległy, koncepcja dzielenia, podejście domenowe (dodatkowy kod do pokazania współbieżności bez dzielenia pamięci). Każdy wątek używa liczb $y \leq \sqrt{n}$ do odznaczania własnego podzbioru N 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.

Kod ten się bardzo efektywnie zrównolegla, ale i tak jest wiele wolniejszy od sit z powodu gorszej złożoności obliczeniowej.

5 Wprowadzenie do rezultatów pomiarów

1. Wykonałem 3 eksperymenty z kodem:

   (a) Pierwszym było stworzenie podstawowego sita i rozwiązania pierwisatkowego. Rozwiązanie pierwiastkowe pokazałem od razu w wersji zrównoleglonej, jej konstrukcja ograniczała się bowiem do dodania jednej dyrektywy do kodu sekwencyjnego. Sito erastotenesa sekwencyjne było podstawą do dalszych obliczeń.

   (b) Drugi eksperyment to sprawdzenie, jak w zależności od przydziału zadań i synchronizacji procesy używające sita w ramach koncepcji funkcyjnej będą działały, co będzie je ograniczało i które będzie efektywniejsze - ręcznie napisane, używające prostej heurystyki czy używające dyrektywy z dynamicznym przydziałem procesów do zadań.

   (c) Trzecim, kluczowym eksperymentem było pokazanie, jak uzyskać eleganckie, bardzo szybkie sito w wariancie domenowym wykorzystując dobrodziejstwa Cache L1, jej własności i proste ulepszenie sita pod kątem zwiększenia liczby trafień do pamięci. Porównywałem je ze standardowym sitem domenowym.

2. VTune Profiler używa licznika zdarzeń sprzętowych do zdobycia informacji na temat przetwarzania, po czym (po przetwarzaniu) łączy te informacje w metryki np. CPI. Używa do zdobywania informacji PMU (performance monitoring unit) - ich liczba jest ograniczona, a sumują one informację o jednym, konkretnym zdarzeniu, przez co część zdarzeń jest raczej estymowana niż deterministycznie obliczana. Pojedynczy PMU wyliczający jakąś konkretną statystykę, wiedząc o zajściu zdarzenia inkrementuje rejestr; gdy rejestr przyjmie wartość progu próbkowania, jest on łączony z instrukcją tak, aby dowiedzieć się, po ilu zdarzeniach globalnych zaszło tyle zdarzeń danego typu (co istotne, aby statystyka/estymacja była zasadna, musi zajść odpowiednia liczba zdarzeń globalnych). Metryki, które zostały użyte w tym sprawozdaniu, to:

   (a) Clockticks - Liczba cykli procesora w trakcie przetwarzania.

   (b) Instructions retired - liczba przedziałów alokacji, które zostały zatwierdzone i w pełni wykonane.

   (c) Retiring - procent przedziałów alokacji, które zostały użyte (nie zaszło ograniczenie front-endu ani back-endu) i wykonane (nie zaszła błędna spekulacja).

   (d) Front-end bound - ile procent przedziałów alokacji nie zostało wykorzystanych przez ograniczenie części wejściowej procesora, albo inaczej: jak często back-end mógł przyjąć jakąś instrukcję, ale nie otrzymał jej od front-endu. Front-end odpowiada za przyniesienie instrukcji (fetch), zdekodowanie jej i przekazanie do back-endu.

   (e) Back-end bound - ile procent przedziałów alokacji nie zostało wykorzystanych przez ograniczenie części wyjściowej procesora, albo incaczej: jak często back-end nie przyjmuje instrukcji od front-endu, ponieważ nie ma zasobów na ich przetworzenie. Składają się na to: Core Bound i Memory bound.

(f) Memory bound - ile procent przedziałów alokacji mogło nie zostać wykonane przez zapotrzebowanie na załadowanie albo składowanie instrukcji - czyli narzut związany z dostępem do pamięci, której albo nie ma w cache, albo jest zabrudzona.

(g) Core bound - ile procent przedziałów alokacji mogło nie zostać wykonane przez ograniczenia inne niż te związane z pamięcią - między innymi dzielenie czy operacje arytmetyczne na liczbach zmiennoprzecinkowych.

(h) Effective physical core utilization - ile procent fizycznych rdzeni średnio było używanych w trakcie przetwarzania.

(i) Metryki L1, L2, L3 bound, a także DRAM bound - procentowe dane, w ilu cyklach procesora zaszła sytuacja, w której program mimo posiadania odpowiednich danych w cache/DRAMie nie mógł przyjąć operacji.

Wykorzystywanym trybem pracy było Microarchitecture Exploration.

6 Tablica wynków: kody sit i pierwiastkowe

Oznaczenia i skróty:

1. name - nazwa kodu, pierwsz cyfry jego nazwy i skrót.
2. left, right - przedział, dla którego wykonano program.
3. T - liczba wątków
4. Elapsed - czas, który upłynął od początku przetwarzania
5. Ticks - liczba cykli procesora w trakcie wykonywania kodu.
6. IR - Instructions Retired
7. R - Retired
8. FEB, BEB - Front-end bound, Back-end Bound
9. MB, CB - Memory Bound, Core Bound
10. L1, L2, L3 - L1 Bound, L2 Bound, L3 Bound
11. DRB, DTB - DRAM Bound, DTLB Store Overhead
12. ECPU - Effective CPU Utilization
13. Div - przyspieszenie przetwarzania równoległego
14. Eff - efektywność przetwarzania równoległego
15. Avg - liczba przetestowanych liczb w jednostce czasu.

name	left	right	T	Elapsed	Ticks	IR	R	FEB	BEB	MB	CB	L1	L2	L3	DRB	DTB	ECPU	Div	Eff	Avg
03_efss	2.0	1.00E+09	4.0	10.199	6.08E+10	1.60E+10	6.8	0.4	92.4	75.5	16.9	9.7	0.3	17.5	0.0	36.3	34.3	728.5	182.12	9.80E+07
04_efhs	2.0	1.00E+09	4.0	8.802	1.20E+11	1.43E+10	2.8	0.9	96.1	78.7	17.4	0.0	0.0	35.5	0.0	43.2	80.5	628.71	157.18	1.14E+08
05_efds	2.0	1.00E+09	4.0	6.417	1.03E+11	1.47E+10	3.3	0.9	95.5	78.5	17.0	2.7	0.0	25.5	0.0	40.8	96.0	458.36	114.59	1.56E+08
08_esd	2.0	1.00E+09	4.0	0.419	6.47E+09	1.81E+10	47.2	9.1	33.1	19.5	13.6	14.6	0.4	1.1	0.0	0.7	92.5	29.93	7.48	2.39E+09
07_ed	2.0	1.00E+09	4.0	9.204	1.47E+11	2.11E+10	3.9	0.6	95.3	73.3	22.1	6.5	0.0	24.5	0.0	35.6	94.8	657.43	164.36	1.09E+08
08_esd	2.0	1.00E+09	1.0	1.494	6.30E+09	1.79E+10	49.8	14.8	25.1	21.5	3.6	26.4	2.0	1.4	0.0	1.7	24.2	106.71	106.71	6.69E+08
08_esd	2.0	1.00E+09	2.0	0.769	6.33E+09	1.79E+10	49.4	12.0	28.8	20.2	8.7	19.5	0.9	1.1	0.0	1.2	47.4	54.93	27.46	1.30E+09
08_esd	2.0	1.00E+09	4.0	0.412	6.35E+09	1.79E+10	51.7	9.7	28.1	17.0	11.1	15.4	0.7	0.0	0.7	0.7	92.4	29.43	7.36	2.43E+09
07_ed	2.0	1.00E+09	1.0	11.113	4.72E+10	1.91E+10	8.9	0.8	90.1	72.5	17.6	21.2	0.9	0.0	9.9	33.3	24.2	793.79	793.79	9.00E+07
07_ed	2.0	1.00E+09	2.0	9.315	7.88E+10	1.92E+10	5.8	0.4	93.5	73.4	20.1	22.4	0.3	7.6	0.0	33.0	48.4	665.36	332.68	1.07E+08
07_ed	2.0	1.00E+09	4.0	9.266	1.48E+11	2.11E+10	4.1	0.7	95.0	72.7	22.3	7.6	0.0	23.4	0.0	34.6	94.8	661.86	165.46	1.08E+08
08_esd	2.0	1.00E+07	1.0	0.019	4.80E+07	1.60E+08	0.0	20.8	79.2	79.2	0.0	41.7	0.0	0.0	0.0	0.0	23.2	1.36	1.36	5.26E+08
08_esd	2.0	1.00E+07	2.0	0.014	5.60E+07	1.64E+08	0.0	0.0	100.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	34.9	1.0	0.5	7.14E+08
08_esd	2.0	1.00E+07	4.0	0.014	6.40E+07	1.68E+08	0.0	0.0	100.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	53.0	1.0	0.25	7.14E+08
07_ed	2.0	1.00E+07	1.0	0.053	1.72E+08	1.80E+08	0.0	0.0	100.0	92.9	7.1	0.0	0.0	0.0	0.0	0.0	23.7	3.79	3.79	1.89E+08
07_ed	2.0	1.00E+07	2.0	0.037	2.16E+08	2.00E+08	0.0	0.0	100.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0	41.6	2.64	1.32	2.70E+08
07_ed	2.0	1.00E+07	4.0	0.025	2.52E+08	2.08E+08	0.0	0.0	100.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	75.1	1.79	0.45	4.00E+08
01_es	2.0	1.00E+09	1.0	10.429	4.43E+10	1.36E+10	7.1	0.5	92.2	77.0	15.3	18.2	1.5	1.1	7.9	32.1	24.2	744.93	744.93	9.59E+07

7 Wnioski

1. Rozwiązanie oparte na wyznaczaniu dzielników mniejszych równych od $\sqrt{n}$ bardzo efektywnie się zrównolegla - procesy dzielą się pracą bardzo równo, przez większość czasu działania programu wykonuje się 7-8 wątków. Zwiększenie liczby procesów k-krotnie powoduje zmniejszenie czasu przetwarzania prawie k-krotnie, przy założeniu, że nowy wątek jest wykonywany przez inny procesor logiczny. Nie zachodzi prawie wcale False Sharing, ponieważ jedyna współdzielona zmienna to tablica liczb pierwszych. Wątki nie czytają współdzielonej pamięci, co powoduje, że praktycznie nie zachodzą cache missy. Wąskim gardłem rozwiązania jest Front-End Bound - co oznacza, że są większe problemy z dostarczeniem zadania do wykonania do Back-endu niż z jego wykonaniem; a także intensywne dzielenie, zwiększające Core-Bound. Algorytm pomimo zrównoleglenia nadal jest wolniejszy od sita, ponieważ ma większą złożoność - dla $n = 10^9$ ponad 1000-krotnie wolniejszy.

2. Sito erastotenesa w podejściu funkcyjnym było w stanie uzyskać prawie 2-krotne przyspieszenie względem sekwencyjnego sita używając dynamic schedulingu. Co ciekawe, rozwiązania używające handmade schedulingu jest około 1.5 razy wolniejsze od rozwiązania używającego static schedulingu, pomimo tego, że ma prawie dwukrotnie wyższy współczynnik "effective physcial core utilization" - powodem jest to, że bardzo często zachodzi

3. W podejściu