Sprawozdanie z projektu i eksperymentu obliczeniowego

Błażej Krzyżanek 136749, Maciej A. Czyżewski 136698

21 maja 2020

1 Wstęp

1. **Nazwa zaliczanego przedmiotu:**
   laboratorium z przetwarzania równoległego
2. **Imiona i nazwiska autorów sprawozdania, numery indeksów, numer grupy dziekańskiej i termin zajęć laboratoryjnych:**
   Błażej Krzyżanek 136749 I1 poniedziałki 8:00 „pod kreską”
   Maciej A. Czyżewski 136698 I1 środa 9:45 „pod kreska”
3. **Wymagany termin oddania sprawozdania:**
   27 Apr. 2020
4. **Wersja:**
   Pierwsza.
   Skrypty (mirror): `https://gist.github.com/maciejczyzewski/b800171f1e4ef4d7d090682e704c76ba`
5. **Krótki opis treści realizowanego zadania:**
   Analiza efektywności algorytmów przetwarzania równoległego realizowanego w komputerze z procesorem wielordzeniowym z pamięcią współdzieloną, na przykładzie problemu znajdowania liczb pierwszych.
6. **Adres email kontaktowy do autorów sprawozdania:**
   blazej.krzyzanek@student.put.poznan.pl
   maciej.czyzewski@student.put.poznan.pl

2 Opis wykorzystanego systemu obliczeniowego

System operacyjny	Microsoft Windows 10 Professional 64bit
Kompilator	Intel C++ Compiler 19.1
Oprogramowanie do analizy	Intel® Parallel Studio XE Cluster Edition for Windows
Procesor	Intel i7-4800MQ
Liczba rdzeni	4
Liczba wątków	8
Cache	6MB
Bazowa częstotliwość	2,70 Ghz

Tabela 1: Dane techniczne

3 Warianty kodu

3.1 Dzielenie – wersja sekwencyjna $D1$

Przedział: $< 2; 10^8 >$**; 1 wątek**	
Algorytm	$D1_{seq}$
czas [s]	135.31
instructions retired	4.66×10^{11}
clockticks	4.87×10^{11}
retiring	47.6%
front-end bound	31.7%
back-end bound	20.2%
memory bound	0.1%
core bound	20.3%
Efektywne wykorzystanie rdzeni fizycznych procesora	22.9%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.005
prędkość przetwarzania $[\frac{1}{s}]$	7.4×10^5
efektywność przetwarzania	0.005

Tabela 2: Porównanie algorytmów

Pierwszą wersją kodu do eksperymentu była prosta implementacja sekwencyjnego algorytmu znajdowania liczb pierwszych z przedziału $< a, n >$ poprzez wykrywanie niezerowej reszty z dzielenia przez $k \leqslant \sqrt{n}$. Algorytm ten cechuje się prostotą implementacji, jednak znaczną złożonością obliczeniową $O(\sqrt{n})$. Wyniki przedstawiono w tabeli 3.

Przedział	**czas [s]**
$< 2; 10^8 >$	135
$< 2; 0.5 \cdot 10^8 >$	50.24
$< 0.5 \cdot 10^8; 10^8 >$	84.65

Tabela 3: Wyniki algorytmu sekwencyjnego

3.2 Dzielenie – wersja równoległa z dynamicznym podziałem przedziału przeszukiwania $D2$

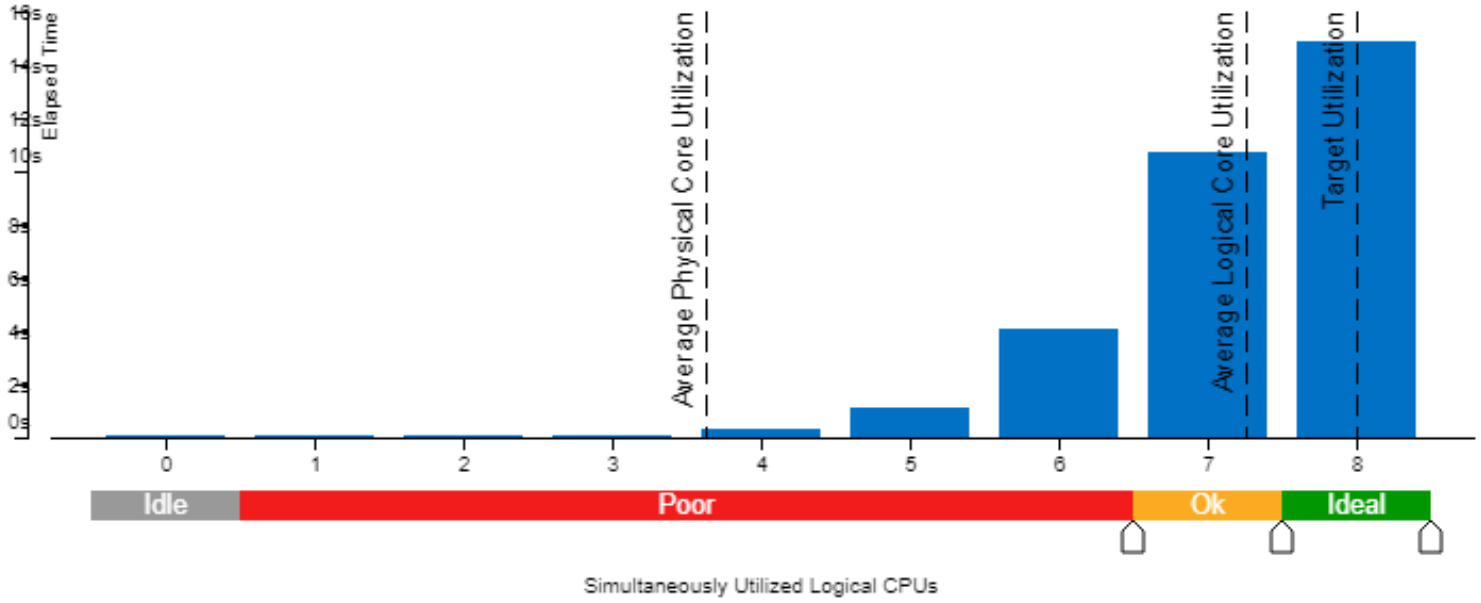
Przedział: $< 2; 10^8 >$**; 8 wątków**	
Algorytm	$D2$
czas [s]	29.99
instructions retired	4.68×10^{11}
clockticks	7.73×10^{11}
retiring	54.3%
front-end bound	24.2%
back-end bound	21.1%
memory bound	0.2%
core bound	21.0%
Efektywne wykorzystanie rdzeni fizycznych procesora	90.7%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.025
prędkość przetwarzania $[\frac{1}{s}]$	3.3×10^6
efektywność przetwarzania	0.003

Tabela 4: Porównanie algorytmów

Kolejnym krokiem w celu usprawnienia algorytmu było zrównoleglenie obliczeń poprzez podział zadanego przedziału $< a, n >$ na fragmenty o długości zmniejszającej się w czasie, poczynając od $\frac{liczba\_nieprzydzielonych\_iteracji}{liczba\_wtkw}$ i następnie wywołania pierwotnego algorytmu dla każdego z tych fragmentów. Po przeanalizowaniu wszystkich liczb z zadanego przedziału, algorytm miał finalnie scalić całą tablicę. Algorytm nie gwarantuje wystąpienia liczb w jakimkolwiek porządku. Wyniki zanotowano w tabeli 5.

Przedział	**czas [s], 8 wątków**	**czas [s], 4 wątki**	**czas [s], 2 wątki**
$< 2; 10^8 >$	29.79	39.76	67.57
$< 2; 0.5 \cdot 10^8 >$	10.18	14.58	25.21
$< 0.5 \cdot 10^8; 10^8 >$	18.57	24.05	42.89

Tabela 5: Wyniki algorytmu równoległego



Porównując czasy przetwarzania algorytmu w wersji sekwencyjnej z zrównolegloną, można zaobserwować nawet kilkukrotne zwiększenie prędkości. Mimo wywołania algorytmu na 8 wątkach, nie udało się jednak uzyskać ośmiokrotnego przyśpieszenia względem wersji jednowątkowej – w najlepszym przypadku przyśpieszenie wyniosło ok. 500%. Analizując sposób wykorzystania procesora przez uruchomiony algorytm przy użyciu oprogramowania Intel VTune Profiler 2020, zaobserwowano wykorzystanie logicznych procesorów na poziomie 93.7%.

3.3 Sito Eratostenesa – wersja z sekwencyjnym wyznaczaniem początkowych liczb pierwszych $S1$

Przedział: $< 2; 10^8 >$**; 8 wątków**	
Algorytm	S1
czas [s]	3.11
instructions retired	1.06×10^{10}
clockticks	1.65×10^{10}
retiring	26.3%
front-end bound	18.5%
back-end bound	51.9%
memory bound	24.5%
core bound	27.4%
Efektywne wykorzystanie rdzeni fizycznych procesora	23.2%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.244
prędkość przetwarzania $[\frac{1}{s}]$	3.2×10^7
efektywność przetwarzania	0.030

Tabela 6: Porównanie algorytmów

Algorytm wyznaczania liczb pierwszych poprzez wielokrotne dzielenie ze względu na swoją złożoność obliczeniową jest wysoce nieefektywny, znany jest inny algorytm dający znacznie lepsze rezultaty – sito Eratostenesa. Pierwsza wersja użytego algorytmu oblicza standardowym, sekwencyjnym algorytmem sita wszystkie liczby pierwsze z przedziału $< 2; \sqrt{n} >$, a następnie z tablicy „wykreśla" wszystkie wielokrotności tych liczb mniejsze od n, używając do tego zadania równoległej pętli `for` uruchomionej jako `guided`. Po wyznaczeniu wielokrotności, algorytm w jednym wątku zapisuje wszystkie indeksy niewykreślonych elementów początkowej tablicy. Wyniki przedstawiono w tabeli 7, uwzględniono też uruchomienie z przydziałem jednego wątku. Zmieniając algorytm na sito Eratostenesa uzyskano 20-krotne zwiększenie wydajności. Na podstawie wyników z tabeli 7 można przypuszczać, że implementacja w praktyce działa jednowątkowo, tak też wynika z analizy VTune Profiler – średnio wykorzystywane było 1.362 z 8 logicznych rdzeni.

Przedział	**czas [s], 8 wątków**	**czas [s], 4 wątki**	**czas [s], 2 wątki**	**czas [s], 1 wątek**
$< 2; 10^8 >$	2.86	2.70	2.71	2.71
$< 2; 0.5 \cdot 10^8 >$	1.36	1.29	1.31	1.29
$< 0.5 \cdot 10^8; 10^8 >$	2.71	2.66	2.62	2.64

Tabela 7: Wyniki algorytmu sekwencyjnego

3.4 Sito – podejście domenowe

W tej wersji każdy wątek dostanie cały vector liczb pierwszych z zakresu sqrt(N) - oraz zakres na przedziale (shift, N) ustalony wartością block_size. Kolejne podejścia będą się rożnić kolejnością/zbalansowaniem/metodą działania danego wątku na jego zakresie.

3.4.1 Przygotowanie eksperymentu

```
void sieve(int N) {
  omp_set_dynamic(0); // FIXME
  omp_set_num_threads(THREAD_NUM);

  primes_vec[primes_size++] = 2;
  for (uint64 i = 3; i <= sqrt(N); i += 2)
    if (is_prime(i))
      primes_vec[primes_size++] = i;

  int shift = primes_vec[primes_size - 1], block_size;
  block_size = (N - shift) / (THREAD_NUM) + 1;
  bench.T1();

  // __fn_1(block_size, primes_size, shift);
  // __fn_2(block_size, primes_size, shift);
  // __fn_3(block_size, primes_size, shift);
  // __fn_3_fastest(block_size, primes_size, shift);

  bench.T2();
  rep(i, shift, N) if (sieve_vec[i] == 0) primes_vec[primes_size++] = i;
}
```

3.4.2 $FN1_D$: zapis sterowany

Przedział: $< 2; 10^8 >$**; 8 wątków**	
Algorytm	$FN1_D$
czas [s]	1.01
instructions retired	4.84×10^9
clockticks	1.92×10^{10}
retiring	11.9%
front-end bound	8.5%
back-end bound	77.7%
memory bound	38.5%
core bound	39.2%
Efektywne wykorzystanie rdzeni fizycznych procesora	59.4%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.752
prędkość przetwarzania $[\frac{1}{s}]$	9.9×10^7
efektywność przetwarzania	0.094

Tabela 8: Porównanie algorytmów

```
1  void __fn_1(int block_size, int primes_size, int shift) {
2  #pragma omp parallel num_threads(THREAD_NUM) default(none)  \
3      firstprivate(block_size, primes_size, shift)            \
4      shared(sieve_vec, primes_vec)
5    rep(i, 0, primes_size) {
6      int p = primes_vec[i];
7      rep(idx, 0, THREAD_NUM) {
8        int _j = shift + idx * block_size;
9  #pragma omp for schedule(guided) nowait
10       for (int j = -fast_mod(_j, p); j <= block_size; j += p)
11         sieve_vec[_j + j] = 1;
12     }
13   }
14 }
```

W tej wersji, problemem jest prędkość zapisu (czekanie na I/O). Dlatego w eksperymencie sprobujemy naprawic ten blad poprzez dodanie *#pragma omp for schedule* przy pętli która wykonuje najwięcej operacji tego typu. To podejście jest jednak nieskuteczne. W tym wypadku w przetwarzaniu rownogleglym bedzie uczestniczylo 8 threadow. Procesom zadania sa przydzialane zgodnie z trybem 'guided' (linia 9) - czyli zakresy zadan dla threadow beda sie zmniejszac eksponencjalnie. Tryb ten zostal wybrany empirycznie poprzez sprawdzenie innych mozliwych trybow oraz wielkosci blokow (dla static oraz dynamic). W tym skrypcie tak jak i wszystkich pozostalych ponizej bedziemy wspoldzielic dwie zmienne dyrektywa *shared* (linia 4) - sieve_vec (tablica z ktorej wykreslamy liczby z zakresu 0 do N) oraz primes_vec (czyli wektor w ktorym przedtrzymujemy liczby pierwsze). Przekazujemy rowniez poprzez klazure *firstprivate* (linia 3) parametry takie jak: block_size - jaki podzakres wystepuje na watek, primes_size - ile mamy liczb pierwszych aktualnie, shift - ile w masce mamy juz wypelnione i mozemy pominac. W tym kodzie nie wystepuje wyscig (sytuacja w ktorej dwa procesy nadpisuja stan tworzac nieprawidlowy wynik poniewaz operowaly na nieaktualnych/niesynchronizowanych stanach[1]), poniewaz stan w sieve_vec moze byc jedynie zmieniony (linia 11) na zaznaczony (wiec podczas konfliktu, i tak uzyskamy porzadany wynik) - jednak w tej sytuacji kazdy operuje na swoim przedziale wiec nie ma nawet takiej mozliwosci. Synchronizacja wystepuja raz po wykonaniu sekcji **omp parallel** (linia 14). A w calym procesie bedzie to ten jedyny raz poniewaz liczby pierwsze z zakresu od 0 do sqrt(N) preprocessujemy sekwencyjnie (naiwny fork&join jest bardziej kosztowny dla malych N).

[1] *przyklad:* 1A: Odczytaj zmienną V, 1B: Odczytaj zmienną V, 2A: Dodaj 1 do zmiennej V, 2B: Dodaj 1 do zmiennej V, 3A: Zapisz wartość w zmiennej V, 3B: Zapisz wartość w zmiennej V, jeśli instrukcja 1B zostanie wykonana pomiędzy 1A i 3A to uzyskamy nie prawidlowy stan. A chcielismy zwiekszyc ta wartosc o 1 niezaleznie dwoma procesami czy sumarycznie o 2, a w rezultacie mamy zwiekszone tylko o 1.

3.4.3 $FN2_D$: wykorzystanie scheduler-a z OpenMP

Przedział: $< 2; 10^8 >$**; 8 wątków**	
Algorytm	$FN2_D$
czas [s]	0.94
instructions retired	4.27×10^9
clockticks	1.69×10^{10}
retiring	11.6%
front-end bound	9.4%
back-end bound	77.6%
memory bound	46.4%
core bound	31.3%
Efektywne wykorzystanie rdzeni fizycznych procesora	55.0%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.809
prędkość przetwarzania $[\frac{1}{s}]$	1.06×10^8
efektywność przetwarzania	0.101

Tabela 9: Porównanie algorytmów

```
1  void __fn_2(int block_size, int primes_size, int shift) {
2  #pragma omp parallel for num_threads(THREAD_NUM) default(none) \
3      firstprivate(block_size, primes_size, shift)                \
4      shared(sieve_vec, primes_vec)                               \
5      collapse(2)
6    rep(i, 0, primes_size) {
7      rep(idx, 0, THREAD_NUM) {
8        int p = primes_vec[i];
9        int _j = shift + idx * block_size;
10 #pragma omp taskloop
11       for (int j = -fast_mod(_j, p); j <= block_size; j += p)
12         sieve_vec[_j + j] = 1;
13     }
14   }
15 }
```

Cel: zbalansowac tym razem zewnetrzne petle odpowiedzialne za ktora liczbe pierwsza przetwarzamy W tym wypadku w przetwarzaniu rownogleglym bedzie uczestniczylo 8 threadow. Procesom zadania sa przydzialane zgodnie z trybem (niewprost) ‘runtime‘ (linia 2) - czyli w moim wypadku ‘auto‘. Tryb ten zostal wybrany empirycznie poprzez sprawdzenie innych mozliwych trybow oraz wielkosci blokow (dla static oraz dynamic). Okazalo sie ze dobranie tych wartosci jest nie trywialne, dlatego zostalismy przy trybie automatycznym. Dodatkowo uzylismy `collapse(2)` aby dac jeszcze wieksza kontrole OpenMP nad rozdzieleniem zadan. Zgodnie z zaleceniem dokumentacji OpenMP ciezka pod wzgledem I/O petle (linia 11) dalismy jako *#pragma omp taskloop*, w praktyce nic to nie zmienia (teorytycznie petla powinna byc podzielona na taski ktore moga swobodnie wykonywac). Wyscigu podobnie jak poprzednio nie mamy, a wyniki beda poprawne poniewaz liczby pierwsze zczytujemy po synchronizacji (linia 15) wiec nie musimy dbac o kolejnosc przetwarzania (czyli jaka liczba pierwsza najpierw).

3.4.4 $FN3_D$: sami decydujemy o przedziałach

Przedział: $< 2; 10^8 >$**; 8 wątków**	
Algorytm	$FN3_D$
czas [s]	0.98
instructions retired	5.50×10^9
clockticks	1.73×10^{10}
retiring	17.6%
front-end bound	11.0%
back-end bound	70.7%
memory bound	33.7%
core bound	37.0%
Efektywne wykorzystanie rdzeni fizycznych procesora	59.5%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.776
prędkość przetwarzania $[\frac{1}{s}]$	1.02×10^8
efektywność przetwarzania	0.096

Tabela 10: Porównanie algorytmów

```
1  void __fn_3(int block_size, int primes_size, int shift) {
2  #pragma omp parallel num_threads(THREAD_NUM) default(none) \
3      firstprivate(block_size, primes_size, shift)           \
4      shared(sieve_vec, primes_vec)
5    rep(i, 0, primes_size) {
6      int idx = omp_get_thread_num();
7      int p = primes_vec[i];
8      int _j = shift + idx * block_size;
9      for (int j = -fast_mod(_j, p); j <= block_size; j += p)
10       sieve_vec[_j + j] = 1;
11   }
12 }
```

Cel: manualnie (czyli bez wbudowanego schedulera dokonac podzialu zakresu dla threadow). Dokonamy tego poprzez parametr block_size - sami decydujemy o tym jaki przedział należy do danego wątku. W tym wypadku w przetwarzaniu rownogleglym bedzie uczestniczylo 8 threadow. Przydzial odbywa sie poprzez pomnozenie (linia 8) identyfikatora watku (0, ilosc watkow) * rozmiar

bloku przedzialu ktory jest tak naprawde wartoscia: zakres/ilosc watkow. A nastepnie wypelenie bloku od (_j) wykresleniami (linia 9 - warunek petli). Wazny jest tez start bloku, nie mozemy zaczac od zera, musimy kontynuowac sekwencje wielokrotonsci danej liczby pierwszej - dlatego uzywamy funkcji –`fast_mod(_j, p)`. Spodziewane jest polepszenie wzgledem automatycznego schedulera - poniewaz robimy to pod ten problem a nie ogolny. Identyfikator watku uzyskamy poprzez funkcje `omp_get_thread_num()`. Stwierdzenia na temat synchronizacji jak i popranowsci takie jak w poprzedniej wersji skryptu.

3.4.5 $FN3_{D512}$: więcej wątków niż procesorów logicznych

Przedział: $< 2; 10^8 >$**; 512 wątków**	
Algorytm	FN_{D512}
czas [s]	**0.09**
instructions retired	4.57×10^9
clockticks	2.51×10^{10}
retiring	9.0%
front-end bound	7.1%
back-end bound	83.7%
memory bound	75.1%
core bound	8.6%
Efektywne wykorzystanie rdzeni fizycznych procesora	**82.7**%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ **jednowątkowo**	8.444
prędkość przetwarzania $[\frac{1}{s}]$	1.1×10^9
efektywność przetwarzania	1.056

Tabela 11: Porównanie algorytmów

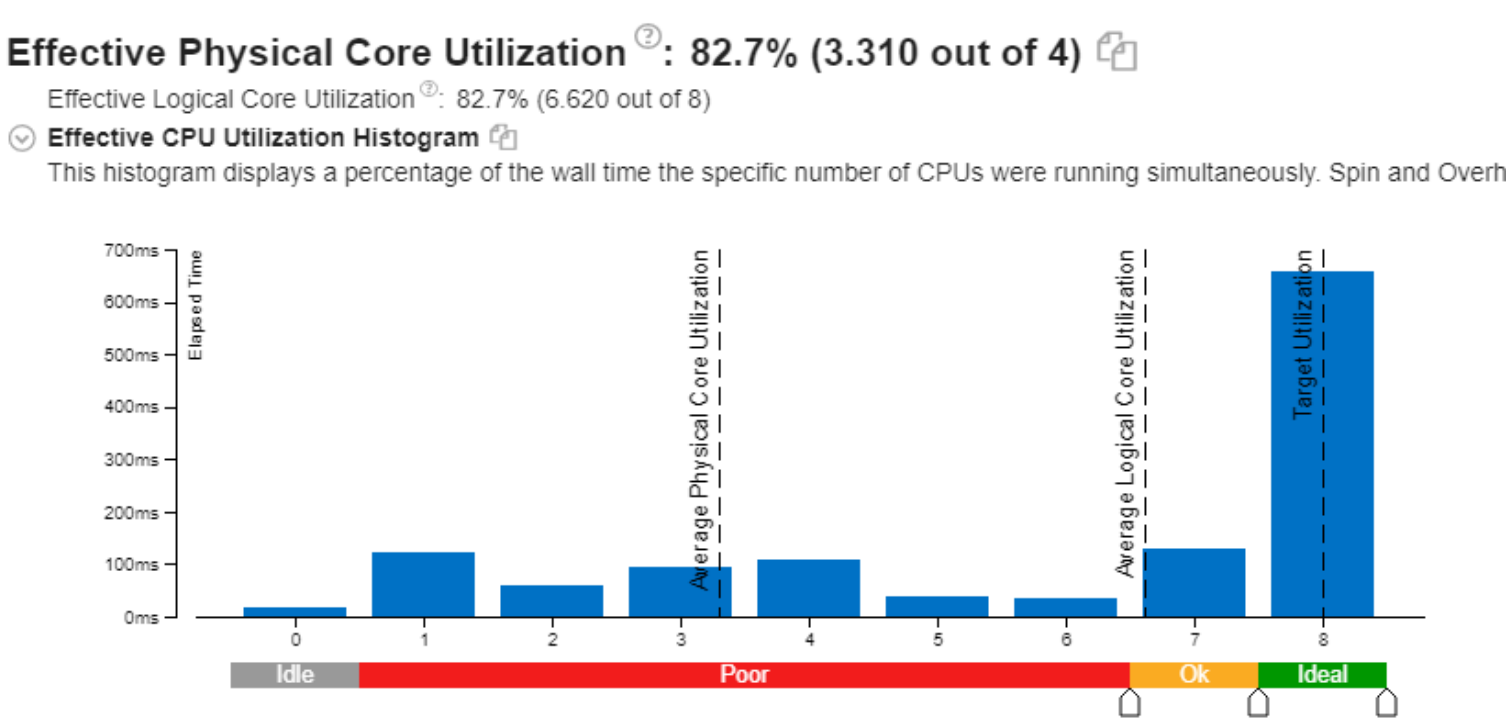
```
1  #define THREAD_NUM 512
2  void __fn_3_fastest(int block_size, int primes_size, int shift) {
3  #pragma omp parallel num_threads(THREAD_NUM) default(none)  \
4      firstprivate(block_size, primes_size, shift)            \
5      shared(sieve_vec, primes_vec)
6    rep(i, 0, primes_size) {
7      int idx = omp_get_thread_num();
8      int p = primes_vec[i];
9      int _j = shift + idx * block_size;
10     for (int j = -fast_mod(_j, p); j <= block_size; j += p)
11       sieve_vec[_j + j] = 1;
12   }
13 }
```

Ważna uwaga! Aby poniższy kod działał prawidłowo musi zachodzić: `THREAD_NUM < sqrt(N)`.

Ta wersja działa szybko ponieważ lepiej wykorzystujemy I/O, w zapisie/odczycie danych. Pojedynczy thread jak coś przetwarza często czeka na I/O (u nas linia 11, wykreslenie liczby), w ten sposób niwelujemy czas tego biernego czekania. Mamy więcej przełączeń pomiędzy kontekstami, ale w tym przypadku nam się to opłaca (widac to po clocktick-sach - czyli wiecej bylo operacji przy porownywalnym back-end bound). Naturalnie mamy zwiekszenie pamieci ze wzgledu na potrzebe przechowywania kontekstu wielu wywlaczszczonych threadow (memory bound). Dlatego w tym wypadku odpalamy poprzednia wersje programu dla duzej ilosc watkow przy podziale manualnym (czyli dokonanym poprzez block_size). (uwaga: po ujednoliceniu kod jest taki sam jak $FN3_D$)

	czas [s]						
Przedział	**512 wątków**	**128 wątków**	**64 wątki**	**8 wątków**	**4 wątki**	**2 wątki**	**1 wątek**
$< 2; 10^8 >$	0.089	0.113	0.384	0.798	0.794	0.809	1.01

Tabela 12: Wyniki algorytmu $FN3_{D512}$



3.5 Sito – podejście funkcyjne

W tej wersji każdy thread będzie operował na fragmencie vector-a liczb pierwszych (shift, primes_size) ustalony wartością block_size - oraz współdzielony dostęp do całego zakresu maski. Kolejne podejścia będą się różnić kolejnością/zbalansowaniem/metodą działania danego threadu na jego zakresie.

3.5.1 Przygotowanie eksperymentu

```
1  void sieve(int N) {
2    omp_set_dynamic(0); // FIXME
3    omp_set_num_threads(THREAD_NUM);
4
5    primes_vec[primes_size++] = 2;
6    for (uint64 i = 3; i <= sqrt(N) * 1.01; i += 2)
7      if (is_prime(i))
8        primes_vec[primes_size++] = i;
9
10   int shift = 0, block_size;
11   block_size = (primes_size - shift) / (THREAD_NUM);
12   bench.T1();
13
14   // __fn_1(block_size, shift, N);
15   // __fn_2(block_size, shift, N);
16   // __fn_3(block_size, shift, N);
17
18   bench.T2();
19   rep(i, primes_vec[primes_size - 1] + 1, N) if (sieve_vec[i] == 0)
20       primes_vec[primes_size++] = i;
21 }
```

3.5.2 $FN1_F$: zapis sterowany

Przedział: $< 2; 10^8 >$**; 8 wątków**	
Algorytm	$FN1_F$
czas [s]	1.28
instructions retired	4.44×10^9
clockticks	1.76×10^{10}
retiring	12.8%
front-end bound	8.0%
back-end bound	78.1%
memory bound	38.7%
core bound	39.4%
Efektywne wykorzystanie rdzeni fizycznych procesora	68.6%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.594
prędkość przetwarzania $[\frac{1}{s}]$	7.8×10^7
efektywność przetwarzania	0.074

Tabela 13: Porównanie algorytmów

```
1  void __fn_1(int block_size, int shift, int N) {
2  #pragma omp parallel num_threads(THREAD_NUM) default(none)  \
3      firstprivate(block_size, shift, N)                      \
4      shared(sieve_vec, primes_vec)
5    {
6      rep(idx, 0, THREAD_NUM) {
7        rep(i, shift + (idx * block_size),
8               shift + (idx + 1) * block_size) {
9          int p = primes_vec[i];
10 #pragma omp for schedule(guided) nowait
11         for (int j = p * p; j <= N; j += p)
12           sieve_vec[j] = 1;
13       }
14     }
15   }
16 }
```

Ta implementacja jest skolerowana do optymalizacji/podejscia $FN1_D$ metody domenowej (wyjasnie tylko roznice). W tej wersji, problemem jest prędkośc zapisu (czekanie na I/O). Dlatego dodany został *#pragma omp for schedule* przy pętli która wykonuje najwięcej takich operacji. To podejście jest jednak nieskuteczne. Tym razem kazdy watek dysponuje calym zakresem. Co sprawia ze I/O bedzie jeszcze troche wolniejsze (duze przeskoki w pamieci pomiedzy sasiadujacymi watkami - potrzeba ciaglego sprowadzania linii).

3.5.3 $FN2_F$: dany blok

Przedział: $< 2; 10^8 >$**; 8 wątków**	
Algorytm	$FN2_F$
czas [s]	1.16
instructions retired	2.46×10^9
clockticks	1.47×10^{10}
retiring	9.2%
front-end bound	5.1%
back-end bound	85.8%
memory bound	42.7%
core bound	43.0%
Efektywne wykorzystanie rdzeni fizycznych procesora	58.3%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.655
prędkość przetwarzania $[\frac{1}{s}]$	8.6×10^7
efektywność przetwarzania	0.082

Tabela 14: Porównanie algorytmów

```
1  void __fn_2(int block_size, int shift, int N) {
2  #pragma omp parallel for num_threads(THREAD_NUM) default(none) \
3      firstprivate(block_size, shift, N)                          \
4      shared(sieve_vec, primes_vec)                               \
5          collapse(2)
6    rep(idx, 0, THREAD_NUM) {
7      rep(i, 0, block_size) {
8        int p = primes_vec[shift + idx * block_size + i];
9  #pragma omp taskloop
10       for (int j = p * p; j <= N; j += p)
11         sieve_vec[j] = 1;
12     }
13   }
14 }
```

Ta implementacja jest skolerowana do optymalizacji/podejscia $FN2_D$ metody domenowej (wyjasnie tylko roznice). W tej wersji wykorzystujemy wbudowany w bibliotekę scheduler. Próbowaliśmy znaleźć odpowiednie parametry: guided, dynamic, static, auto. Zmieniać wielkość bloku. Niestety wszystkie wersje zachowywały się tak samo. Petle nalezy zaprezentowac w wersji kanonicznej dlatego nalezalo wyelimionwac z $FN2_D$:linia 7-8 odwolanie do *idx*. Dokonalismy tego w linii 8.

3.5.4 $FN3_F$: sami decydujemy o przedziałach

Przedział: $< 2; 10^8 >$; 8 wątków	
Algorytm	$FN3_F$
czas [s]	0.98
instructions retired	5.17×10^9
clockticks	9.48×10^9
retiring	23.9%
front-end bound	14.7%
back-end bound	60.7%
memory bound	29.1%
core bound	31.7%
Efektywne wykorzystanie rdzeni fizycznych procesora	37.1%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.776
prędkość przetwarzania $[\frac{1}{s}]$	1.02×10^8
efektywność przetwarzania	0.096

Tabela 15: Porównanie algorytmów

```
1  void __fn_3(int block_size, int shift, int N) {
2  #pragma omp parallel num_threads(THREAD_NUM) default(none)  \
3      firstprivate(block_size, shift, N)                      \
4      shared(sieve_vec, primes_vec)
5    {
6      int idx = omp_get_thread_num();
7      rep(i, shift + (idx * block_size),
8             shift + (idx + 1) * block_size) {
9        int p = primes_vec[i];
10       for (int j = p * p; j <= N; j += p)
11         sieve_vec[j] = 1;
12     }
13   }
14 }
```

Ta implementacja jest skolerowana do optymalizacji/podejscia $FN3_D$ metody domenowej (wyjasnie tylko roznice). W tym wypadku poprzez parametr block_size sami decydujemy o tym jaki przedział należy do danego wętku.

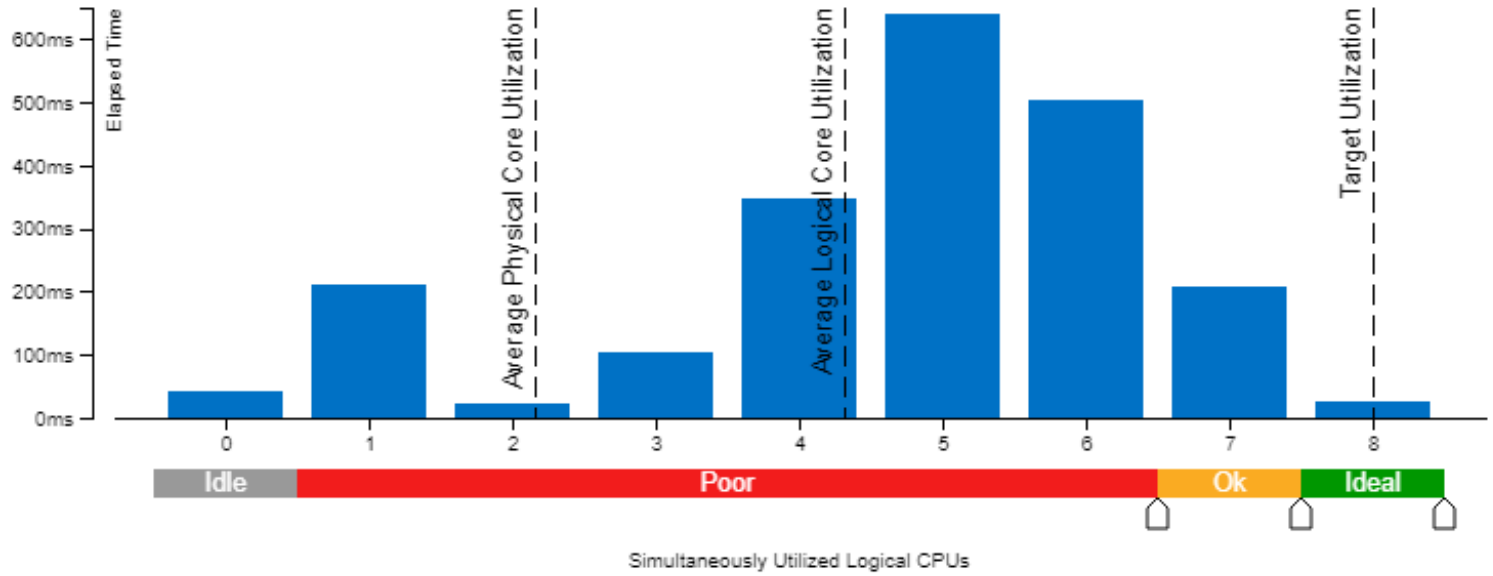
3.6 Sito – taskloop

Przedział: $< 2; 10^8 >$; 8 wątków	
Algorytm	$FN_{Taskloop}$
czas [s]	0.56
instructions retired	3.71×10^9
clockticks	3.47×10^{10}
retiring	5.8%
front-end bound	3.4%
back-end bound	90.4%
memory bound	48.2%
core bound	42.1%
Efektywne wykorzystanie rdzeni fizycznych procesora	54.0%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ jednowątkowo	0.487
prędkość przetwarzania $[\frac{1}{s}]$	6.4×10^7
efektywność przetwarzania	0.061

Tabela 16: Porównanie algorytmów

```
1  void sieve(int N) {
2    sieve_vec[0] = sieve_vec[1] = 1;
3    int _N2 = int(sqrt(N)) / 2;
4
5  #pragma omp parallel num_threads(THREAD_NUM) default(none)
6      firstprivate(_N2, N) shared(sieve_vec)
7    {
8  #pragma omp for schedule(guided) nowait
9      for (int j = 2 * 2; j < N; j += 2)
10       sieve_vec[j] = 1;
11
12 #pragma omp for schedule(dynamic, 2) nowait
13     for (int i = 3; i < _N2; i++) {
14       int x = 2 * i - 3;
15       if (sieve_vec[x] == 0) {
16 #pragma omp taskloop simd nogroup num_tasks(512)
17         for (int j = 0; j < (N - x * x) / (x * 2) + 1; j++) {
18           sieve_vec[x * x + j * x * 2] = 1;
19         }
20       }
21     }
22   }
23 }
```

Powyzej prezentujemy podejście hybrydowe. Nie zapisujemy liczb pierwszych które później rozdzielamy aby każdy watek przetwarzał jakąś część. Ten skrypt ma parallelizm w dwóch kierunkach - 1) kolejne liczby są oceniane czy powinny zostać wykreślone 2) gdy coś ma zostać wykreślone zostaje dodane jako *task* do scheduler-a (a wiec może się wykonać na zmianę z zewnętrznymi iteracjami). W eksperymencie wyszło ze jest to skuteczna technika aby zaimplementować najszybszą wersję jedno/dwu wątkowa. Jest to spowodowane mniejszym obciążeniem pamięci (nie musimy trzymać wektora liczb pierwszych) oraz wykorzystujemy wektoryzację operacji poprzez SIMD oraz poprzez `num_tasks(512)` poprawiamy analogicznie jak w $FN3_{D512}$ - I/O. Synchronizacja nastepuje w linii 22, nowait w linii 12 nic nie robi przez to, natomiast ten w linii 8 dziala prawidlowo bo jakies watki moga szybciej skonczyc ta petle.



4 Wyniki

Kody ktore byly testowane zostaly juz omowione w sekcjach powyzej, szczegolowy wyniki, miary jakosc i ocena znajduja sie w tabeli nizej w raporcie (obracona tabela na cala strone) - jednostki standardowe zaznaczone w pierwszej kolumnie. Dane zebrane przez *Intel Parallel Studio XE* sa bardzo cenne poniewaz pozwalaja okreslic jakos danego algorytmu oraz znalesc potencjalne miejsca na usprawnienie. Zalecana technika jest nieoptymalizowanie kodu w ciemno, tylko najpierw zbieranie danych wydajnioswiocy, doglebna analiza, obranie kierunku dziala lub decyzja o zakonczeniu procesu optymalizacja (tak jak w naszym przypadku dla wersji FN_{D512} - ktora spelnia nasze wymagania oferujac bardzo szybkie rownolegle wyliczenie liczb pierwszych w zadanym wykresie - ta wersja nalezo by teraz porownac z szybkoscia implementacji sit z roznych bibliotek open source). *Intel Parallel Studio XE* zbiera statystyki poprzez zmodyfikowanie badanego kodu (robiac swoje wstawki, wtedy pewne fragmenty kontu inwokuja funkcje z Vtune w odpowiednich momentach lub odstepach) - zasada jest podobna do dzialania ptrace a ta technika nazywa sie "targeted injection".

Wszystkie wersje z sitem maja problem z prędkością zapisu do współdzielonej maski (tak wynika ze statystyk). Najtrudniejszym zadaniem w napisaniu szybkiego sita jest dobre zarządzanie pamięcią (a doklanie I/O wait). Podejście z alokowaniem pamięci w sekcji **omp parallel** nie daje żadnych usprawnień. Przykład takiej implementacji na dole (nie prezentujemy statystyk dla niej ponieważ pokrywają się z `__fn_3`).

```
void __fn_3_local_sieve_vec(int block_size, int primes_size, int shift) {
#pragma omp parallel num_threads(THREAD_NUM) default(none)                      \
    firstprivate(block_size, primes_size, shift) shared(sieve_vec, primes_vec)
  {
    uint_fast8_t *local_sieve_vec =
        (uint_fast8_t *)calloc(sizeof(uint_fast8_t), (block_size + 2) + 1);

    int idx = omp_get_thread_num();
    int _j = shift + idx * block_size;

    rep(i, 0, primes_size) {
      int p = primes_vec[i];
      for (int j = p - fast_mod(_j, p); j <= block_size + 2; j += p)
        local_sieve_vec[j] = 1;
    }

    memcpy(&sieve_vec[_j], &local_sieve_vec[0],
           (block_size + 2) * sizeof(local_sieve_vec[0]));
  }
}
```

Dlatego wersja `__f3_fastest` ($FN3_{D512}$) ktora rozwiazuje problem I/O - jest wydajniosiowo 10 razy lepsza niz srednia z predkosci przetwarzania pozostalych programow. Druga najlepsza wersja to $FN_{Taskloop}$ - oba podejscia wyrozniaja sie od pozostalych czasem przetwarzania zakresu od 2 do 10^8. Inne programy pod wzgledem wydajniosowiym nie zbyt sie roznia od siebie - jest to spowodowane tym ze zysk z optymalizacji w poszczegolnych przypadkach jest mniejszy niz strata na I/O lub innych wadach zlej rozplonwajej struktury pamieci. W przypadku $D2$ - mamy „efektywne wykorzystanie rdzeni fizycznych procesora" na poziomie 90%, jednak nie doprowadz do obliczen w krotszym czasie. Drugie najlepsze pod wzgledem tej miary to $FN3_{D512}$ - 83%, przeklada sie to na 8 krotne przyspieszenie tego programu (na architekturze ktora ma 8 rdzieni logicznych) wzgledem wersji sekwencyjnej (czy

tez jak w naszym wypadku $FN_{Taskloop}$ odpalony na jednym watku). Najgorsze okazalo sie podejscie z “Dzieleniem”, czyli wersja co nie uzywa sita. Przyczyna slabego wyniku jest poprostu slaba zlozonosc algorytmu odpowiadajacego algorytmu sekwencyjnego ktorego nie da sie optymalnie przelozyc na wersje rownolegla (nie tak jak sito). Problemem tego algorytmu jest przedewszystkim brak utylizacji pamieci oraz madrego zapamietywania wykonanej uprzednio pracy. Zaden algorytm oprocz $FN3$ nie jest skalowalny - wynika do z wczesniej juz wspomnienego problemu z czekaniem na I/O, ktory jest dominujacym problemem.

Przedział: $< 2; 10^8 >$**; 8 wątków (oprócz eksperymentu** $FN3_{D512}$ i $D1_{seq}$)											
Algorytm	$D1_{seq}$	$D2$	S1	$FN1_D$	$FN2_D$	$FN3_D$	$FN1_F$	$FN2_F$	$FN3_F$	FN_{D512}	$FN_{Taskloop}$
czas [s]	135.31	29.99	3.11	1.01	0.94	0.98	1.28	1.16	0.98	**0.09**	0.56
instructions retired	4.66×10^{11}	4.68×10^{11}	1.06×10^{10}	4.84×10^9	4.27×10^9	5.50×10^9	4.44×10^9	2.46×10^9	5.17×10^9	4.57×10^9	3.71×10^9
clockticks	4.87×10^{11}	7.73×10^{11}	1.65×10^{10}	1.92×10^{10}	1.69×10^{10}	1.73×10^{10}	1.76×10^{10}	1.47×10^{10}	9.48×10^9	2.51×10^{10}	3.47×10^{10}
retiring	47.6%	54.3%	26.3%	11.9%	11.6%	17.6%	12.8%	9.2%	23.9%	9.0%	5.8%
front-end bound	31.7%	24.2%	18.5%	8.5%	9.4%	11.0%	8.0%	5.1%	14.7%	7.1%	3.4%
back-end bound	20.2%	21.1%	51.9%	77.7%	77.6%	70.7%	78.1%	85.8%	60.7%	83.7%	90.4%
memory bound	0.1%	0.2%	24.5%	38.5%	46.4%	33.7%	38.7%	42.7%	29.1%	75.1%	48.2%
core bound	20.3%	21.0%	27.4%	39.2%	31.3%	37.0%	39.4%	43.0%	31.7%	8.6%	42.1%
Efektywne wykorzystanie rdzeni fizycznych procesora	22.9%	90.7%	23.2%	59.4%	55.0%	59.5%	68.6%	58.3%	37.1%	**82.7**%	54.0%
przyspieszenie przetwarzania względem $FN_{Taskloop}$ **jednowątkowo**	0.005	0.025	0.244	0.752	0.809	0.776	0.594	0.655	0.776	8.444	0.487
prędkość przetwarzania $[\frac{1}{s}]$	7.4×10^5	3.3×10^6	3.2×10^7	9.9×10^7	1.06×10^8	1.02×10^8	7.8×10^7	8.6×10^7	1.02×10^8	1.1×10^9	6.4×10^7
efektywność przetwarzania	0.005	0.003	0.030	0.094	0.101	0.096	0.074	0.082	0.096	1.056	0.061

Tabela 17: Porównanie algorytmów

5 Program testujący poprawność

Poniżej znajduje się program który używając biblioteki **primesieve** weryfikuje nasze wyniki (z eksperymentów) z prawidłowymi. W ten sposób mamy pewność, ze nasze programy znajdują prawidłowe liczby (oraz wielokrotnie odpalalismy aby sprawdzic czy wynik jest stabilny). Flagi `-fsanitize=undefined,address` infomowaly nas o mozliwych naruszeniach pamieci.

```python
import subprocess
from primesieve import *
import sys, IPython.core.ultratb

sys.excepthook = IPython.core.ultratb.ColorTB()

# GCC = "g++-9 -Wall -Wconversion -Wfatal-errors -g -std=c++17 \
# -fsanitize=undefined,address -Wno-builtin-macro-redefined"
GCC = "g++-9 -Wall -Wconversion -Wfatal-errors \
-Wno-builtin-macro-redefined -std=c++17"
CMD = GCC + " -O3 -fopenmp {}.cc && ./a.out {} {} {}"

def run(ver, M, N, only_count=False):
    global CMD
    if not only_count:
        only_count = "p"

    print("==>", CMD.format(ver, M, N, only_count))
    proc = subprocess.Popen(
        [CMD.format(ver, M, N, only_count)], stdout=subprocess.PIPE, shell=True
    )
    (out, err) = proc.communicate()

    data = {"vec": [], "count": -1, "time_clock": 0, "time_wall": 0}

    for row in str(out.decode("utf-8")).split("\n"):
        print("|", row)
        if row.startswith("[SIEVE]"):
            continue
        elif row.startswith("[COUNT]"):
            data["count"] = int(row.replace("[COUNT]", ""))
        elif row.startswith("<time.h>"):
            data["time_clock"] = float(row.replace("<time.h> time=", ""))
        elif row.startswith(" <omp.h>"):
            data["time_wall"] = float(row.replace(" <omp.h> time=", ""))
        else:
            data["vec"] += map(int, filter(None, row.split(" ")))

    return data

################################################################################

program_name = sys.argv[1]
print(f"--> [{program_name}]")

M, N = 0, 10 ** 8
data = run(program_name, M, N, only_count=True)

count = count_primes(N) - count_primes(M)
print(f"{count} == {data['count']}")
assert count == data["count"]
```

Uruchamianie odbywa się następująco:

```
$python3 verify.py FN_D512
```