Sprawozdanie PROJEKT 1 OMP

Wstęp:

Nazwa zaliczanego przedmiotu: laboratorium z przetwarzania równoległego

Autorzy: Kacper Nawrot 145246, grupa L11, środa 9:45, tygodnie nieparzyste,
Michał Olszewski 144482, grupa L11, środa 9:45, tygodnie nieparzyste

Terminy: wymagany: 11.05.2022,
rzeczywisty: 11.05.2022

Wersja sprawozdania: pierwsza

Opis zadania: Dokonać analizy efektywności algorytmów wyznaczających liczby pierwsze w przedziale <M;N>, napisanych w sposób równoległy oraz sekwencyjny. Wykorzystać metodę pierwiastkowania oraz metodę Sita Eratostenesa.

Adresy email: kacper.nawrot@student.put.poznan.pl
michal.m.olszewski@student.put.poznan.pl

1. Opis wykorzystanego systemu obliczeniowego:

Procesor	Intel Core i7-6700HQ
Liczba procesorów fizycznych (rdzeni)	4
Liczba procesorów logicznych (wątków)	8
Cache L1	4 x 32 KB
Cache L2	4 x 256 KB
Cache L3	6 MB
Bazowa częstotliwość	2.60 GHz
System operacyjny	Ubuntu 20.04.1
Kompilator	gcc (Ubuntu 9.4.0-1ubuntu1-20.04.1) 9.4.0
Oprogramowanie do analizy	Visual Studio Code 1.67 oraz Intel VTune Profiler 2022.1

2. Przygotowane warianty kodów

a) Wersja 01 - dzielenie sekwencyjne (metoda pierwiastkowa)

```
int isPrime(unsigned long x) {
    for(unsigned long i = 2; i * i <= x; i++) {
        if(x % i == 0) {
            return 0;
        }
    }
    return 1;
}

unsigned long countPrimes(char* tab, unsigned long n, unsigned long m) {
    unsigned long counter = 0;
    double t1 = omp_get_wtime();

    for(unsigned long i = m; i <= n ; i++) {
        if(!isPrime(i)) {
            tab[i-m] = '0';
        } else {
            counter++;
        }
    }

    double t2 = omp_get_wtime();
    printf("%.6lf\n", t2 - t1);
    return counter;
}
```

Zdjęcie 1 Sekwencyjny wariant kodu - dzielenie (division_seq.cpp)

Powyższy algorytm poprzez znajdywanie niezerowej reszty z dzielenia przez liczbę ***i***, która jest mniejsza bądź równa pierwiastkowi z maksymalnej wartości, klasyfikuje czy dana liczba jest pierwsza czy też złożona. Czas działania kodu jest dość długi ze względu na pracę na jednym wątku.

b) Wersja 02 - dzielenie równoległe (metoda pierwiastkowa)

```
unsigned long countPrimes(char* tab, unsigned long n, unsigned long m, int threads) {
    unsigned long counter = 0;
    double t1 = omp_get_wtime();

    omp_set_num_threads(threads);
    #pragma omp parallel for reduction (+:counter)
        for(unsigned long i = m; i <= n; i++) {
            if(!isPrime(i)) {
                tab[i-m] = '0';
            } else {
                counter++;
            }
        }

    double t2 = omp_get_wtime();
    printf("%.6lf\n", t2 - t1);
    return counter;
}
```

Zdjęcie 2 Równoległy wariant kodu - dzielenie (division_par.cpp)

Drugi wariant przedstawia dokładnie ten sam kod co wersja 01 z dopisaną dyrektywą:

#pragma omp paralel for reduction (+:counter)

Stosując powyższą dyrektywę, nasz kod będzie wykonywany równolegle. Przyspieszenie w wykonywaniu algorytmu wynika z tego, że każdy procesor logiczny wyznacza wartość ***counter*** lokalnie (zmienna jest prywatna dla każdego wątku) i dopiero po zakończeniu wykonywania pętli, dodaje wartość lokalną do globalnej (redukcja na końcu regionu równoległego - synchronizacja). Dyrektywa przydziela wątkom równą liczbę iteracji (sąsiednich). Dzięki takiemu zabiegowi unikamy zjawiska **wyścigu**[1]. Z kolei **false sharing**[2] występuje, ponieważ przez większość czasu pracy wykonywania programu, wątki muszą aktualizować swoją linię pamięci. Algorytm jest nieefektywny ze względu na to, że nie wykluczamy wielokrotności odnalezionych liczb pierwszych, tylko algorytm sprawdza każdą możliwą wartość.

[1] Sytuacja **wyścigu** pojawia się w oprogramowaniu, gdy prawidłowe działanie programu komputerowego zależy od kolejności lub synchronizacji procesów lub wątków programu. Krytyczne warunki wyścigu powodują nieprawidłowe wykonanie i błędy oprogramowania. Krytyczne warunki wyścigu często zdarzają się, gdy procesy lub wątki zależą od jakiegoś współdzielonego stanu. Operacje na wspólnych stanach są wykonywane w krytycznych sekcjach, które muszą się wzajemnie wykluczać. Nieprzestrzeganie tej zasady może uszkodzić stan współdzielony.

[2] **False sharing** jest zjawiskiem polegającym na unieważnieniu linii pamięci podręcznych procesorów. Sprowadza się to do konieczności aktualizowania danych na unieważnionej linii przez wszystkie wątki.

c) Wersja 03 – sekwencyjna metoda sita Eratostenesa

```
void findPrimes(bool *primes, int m, int n) {
    for (int i = 2; i*i <= n; i++) {
        if (primes[i] == 0) {
            for (int j = i*i; j <= n; j+=i) {
                primes[j] = 1;
            }
        }
    }
}

unsigned long countPrimes(bool *primes, unsigned long m, unsigned long n) {
    int counter = 0;
    for (unsigned long i = m; i <= n; i++) {
        if (primes[i] == 0) {
            counter++;
        }
    }
    return counter;
}
```

Zdjęcie 3 Sekwencyjny wariant kodu - Sito Eratostenesa (sieve_seq.cpp)

Algorytm sekwencyjnego Sita Eratostenesa, którego wyniki są odniesieniem do wersji równoległych. Wykreślanie liczb złożonych występuje wyłącznie na podstawie odnalezionych liczby pierwszych (przykładowo: wartość ***i*** nigdy nie przyjmie 10, ponieważ dziesiątka została odsiana wcześniej przez jej podzielnik – dwójkę). W kodzie występuje jeszcze jedna optymalizacja – pozbywamy się liczb złożonych rozpoczynając od kwadratu danej liczby.

d) Wersja 04 – równoległa, domenowa metoda Sita Eratostenesa

```
void findPrimes(bool *primes, int m, int n, int threads) {
    int sqrtN = floor(sqrt(n));

    if (n > 4) {
        findPrimes(primes, 2, sqrtN, threads);
    } else {
        primes[0] = primes[1] = 1;
        if (n == 4) {
            primes[4] = 1;
        }
        return;
    }

    omp_set_num_threads(threads);
    #pragma omp parallel
    {
        int threadNum = omp_get_thread_num(), allThreads = omp_get_num_threads();
        int minValue = (n / allThreads) * threadNum;
        int maxValue = minValue + n / allThreads - 1;

        if (threadNum == allThreads - 1) {
            maxValue = n;
        }
        if (minValue <= sqrtN) {
            minValue = sqrtN + 1;
        }

        for (int i = 0; i <= sqrtN; i++) {
            if (primes[i] == 1) {
                continue;
            }

            int x = findProduct(minValue, i);
            for (int j = x; j <= maxValue; j += i) {
                primes[j] = 1;
            }
        }
    }
}
```

Zdjęcie 4 Równoległy, domenowy wariant kodu - Sito Eratostenesa (sieve_par_dom.cpp)

Równoległa wersja Sita Eratostenesa wykorzystująca podejście domenowe. Na początku, w sposób rekursywny, wyznaczane są wszystkie liczby pierwsze, mniejsze bądź równe pierwiastkowi z ***n (sqrtN)***. Wątki tworzą nowe przedziały liczb <***minValue,maxValue***>, na podstawie swojego numeru (***omp_get_thread_num()***). Dzięki temu zabiegowi podział jest równomierny. Następnie poszukiwane są liczby pierwsze, również mniejsze bądź równe ***n*** poprzez odrzucanie wielokrotności poprzednio znalezionych liczb pierwszych. False sharing może zajść jedynie w momencie, gdy wątek sprawdza złożoność jakiejś liczby, pobierając również inną liczbę do swojego cache, której wartość może być w tym samym momencie zmieniana przez inny procesor logiczny. Brak wyścigu, synchronizacja na końcu regionu równoległego, bez wpływu na obliczenia.

e) Wersja 05 – równoległa, funkcyjna metoda Sita Eratostenesa

```
void findPrimes(bool *primes, int m, int n, int threads) {
    int sqrtN = floor(sqrt(n));

    if (n > 4) {
        findPrimes(primes, 2, sqrtN, threads);
    } else {
        primes[0] = primes[1] = 1;
        if (n == 4) {
            primes[4] = 1;
        }
        return;
    }

    omp_set_num_threads(threads);
    #pragma omp parallel for
    for (int i = 2; i <= sqrtN; i++) {
        if (primes[i] == 1) {
            continue;
        }

        int j = max(findProduct(sqrtN + 1, i), findProduct(m, i));
        for (j = max(i*i, j); j <= n; j += i) {
            primes[j] = 1;
        }
    }
}
```

Zdjęcie 5 Równoległy, funkcyjny wariant kodu - Sito Eratostenesa (sieve_par_fun.cpp)

Równoległa wersja Sita Eratostenesa wykorzystująca podejście funkcyjne. Na początku, w sposób rekursywny, wyznaczane są wszystkie liczby pierwsze, mniejsze bądź równe pierwiastkowi z ***n*** *(**sqrtN**)*. Następnie poszukiwane są liczby pierwsze, również mniejsze bądź równe ***n***, poprzez odrzucanie wielokrotności poprzednio znalezionych liczb pierwszych. Dyrektywa ***#pragma omp parallel*** rozdzieli liczbę iteracji po równo pomiędzy wątki. Zjawisko **wyścigu** w tym wypadku nie zachodzi, ponieważ wątki modyfikują tylko wartości liczb złożonych. Jedna liczba może zostać zmodyfikowana przez wiele wątków, lecz ostateczny wynik jest taki sam (ciągły **false sharing**). Na końcu pętli następuje synchronizacja, bez efektów na czas przetwarzania.

f) Wersja 06 – równoległa, funkcyjna metoda Sita Eratostenesa wykorzystująca szeregowanie dynamiczne

```
omp_set_num_threads(threads);
#pragma omp parallel for schedule(dynamic)
for (i = 2; i <= sqrtN; i++) {
    if (primes[i] == 1) {
        continue;
    }

    int j = max(findProduct(sqrtN + 1, i), findProduct(m, i));
    for (j = max(i*i, j); j <= n; j += i) {
        primes[j] = 1;
    }
}
```

Zdjęcie 6 Równoległy, funkcyjny wariant kodu - Sito Eratostenesa, wykorzystujący szeregowanie dynamiczne (sieve_par_fun_dynamic.cpp)

Równoległa wersja Sita Eratostenesa wykorzystująca podejście funkcyjne oraz szeregowanie dynamiczne. W stosunku do poprzedniej wersji kodu, jedyną różnicą jest dopisanie ***schedule(dynamic)*** do dyrektywy równoległej. Szeregowanie dynamiczne działa na zasadzie „kto pierwszy ten lepszy", tzn. jeśli wątek jest gotowy do pracy to przydzielana jest mu iteracja (iteracje muszą być przypisywane po jednej na raz do wątków, gdy staną się dostępne, z synchronizacją dla każdego przypisania). Pozwoli nam to na równoważenie obciążenia. Jeśli każda iteracja znacznie różni się od średniego czasu przetwarzania, w przypadku statycznym może wystąpić duża nierównowaga pracy (co jest widoczne w wersji z szeregowaniem statycznym). Dynamiczne szeregowanie wprowadza dodatkowe koszty, ale w tym konkretnym przypadku prowadzi do znacznie lepszego rozkładu pracy. Jeśli chodzi o **false sharing** i **wyścig** to sytuacja się nie zmieniła w stosunku do poprzedniej wersji (brak wyścigu i ciągły false sharing).

3. Prezentacja wyników i omówienie przebiegu eksperymentu obliczeniowo-pomiarowego

Wykonaliśmy dwa eksperymenty:

a) Stworzyć dwa warianty kodu: sekwencyjny oraz równoległy w celu odnalezienia liczb pierwszych z przedziału <M;N> wykorzystując metodę pierwiastkową. Porównać wyniki kodów przy użyciu oprogramowania Intel Vtune.

b) Stworzyć sekwencyjny kod Sita Eratostenesa oraz jego wersje równoległe: domenowe i funkcyjne. Sprawdzić jak wypływa szeregowanie dynamiczne na gorszy wariant kodu równoległego pod względem efektywności wykorzystania procesora. Porównać wyniki kodów przy użyciu oprogramowania Intel Vtune.

Opis Intel Vtune:

Oprogramowanie Intel Vtune wykorzystuje licznik zdarzeń sprzętowych oraz PMU (Performance Monitoring Unit) do pozyskania danych o wykonywanym programie. Liczba PMU jest ograniczona, więc nie są w stanie wyznaczyć wartości dla wszystkich zdarzeń, przez co część z nich jest estymowana. Na końcu oprogramowanie łączy zebrane dane i przedstawia je na metrykach. Wykorzystane przez nas metryki w projekcie to:

- Elapsed Time **[Elapsed]** – czas od początku przetwarzania,
- Clockticks **[Ticks]** – liczba cykli procesora w trakcie przetwarzania,
- Instructions retired **[IR]** – liczba przedziałów alokacji, które zostały zatwierdzone i w pełni wykonane
- Retiring **[R]** – procent przedziałów alokacji, które zostały użyte (nie zaszło ograniczenie front-endu ani back-endu) i wykonane (nie zaszła błędna spekulacja)
- Front-end bound **[F-E]** – ile procent przedziałów alokacji nie zostało wykorzystanych przez ograniczenie części wejściowej procesora,
- Back-end bound **[B-E]** – ile procent przedziałów alokacji nie zostało wykorzystanych przez ograniczenie części wyjściowej procesora,
- Memory bound **[MB]** – ile procent przedziałów alokacji mogło nie zostać wykonane przez zapotrzebowanie na załadowanie albo składowanie instrukcji,
- Core bound **[CB]** – ile procent przedziałów alokacji mogło nie zostać wykonane przez ograniczenia inne niż te związane z pamięcią,
- Effective physical core utilization **[ECPU]** – ile procent fizycznych rdzeni średnio było używanych w trakcie przetwarzania.

Nr	Kod	Min	Max	Wątki	Elapsed[s]	Ticks	IR	R [%]
1	**01**	**5.00E+06**	**1.00E+07**	**1**	**7.677**	**2.55E+10**	**1.10E+10**	**49.6**
2	**01**	**2**	**5.00E+06**	**1**	**4.604**	**1.52E+10**	**6.56E+09**	**49.7**
3	**01**	**2**	**1.00E+07**	**1**	**12.406**	**4.08E+10**	**1.75E+10**	**47.6**
4	02	5.00E+06	1.00E+07	4	2.513	2.82E+10	1.10E+10	49.9
5	02	2	5.00E+06	4	1.794	1.70E+10	6.57E+09	60.2
6	02	2	1.00E+07	4	4.809	4.49E+10	1.76E+10	44.7
7	**02**	**5.00E+06**	**1.00E+07**	**8**	**1.415**	**3.15E+10**	**1.11E+10**	**86.4**
8	**02**	**2**	**5.00E+06**	**8**	**1.023**	**1.89E+10**	**6.70E+09**	**100**
9	**02**	**2**	**1.00E+07**	**8**	**2.738**	**4.92E+10**	**1.76E+10**	**78.1**
10	03	5.00E+08	1.00E+09	1	8.234	2.72E+10	1.37E+10	8.2
11	03	2	5.00E+08	1	4.08	1.35E+10	8.01E+09	9.7
12	03	2	1.00E+09	1	8.734	2.84E+10	1.62E+10	13.7
13	**04**	**5.00E+08**	**1.00E+09**	**4**	**6.702**	**7.92E+10**	**1.39E+10**	**3.4**
14	**04**	**2**	**5.00E+08**	**4**	**3.395**	**3.88E+10**	**8.15E+09**	**8.5**
15	**04**	**2**	**1.00E+09**	**4**	**7.152**	**8.10E+10**	**1.65E+10**	**6.6**
16	04	5.00E+08	1.00E+09	8	6.901	1.50E+11	1.42E+10	1.8
17	04	2	5.00E+08	8	3.482	7.35E+10	8.33E+09	4.4
18	04	2	1.00E+09	8	7.069	1.50E+11	1.66E+10	6.8
19	**05**	**5.00E+08**	**1.00E+09**	**4**	**4.141**	**1.80E+10**	**8.18E+09**	**10.7**
20	**05**	**2**	**5.00E+08**	**4**	**3.857**	**1.60E+10**	**8.03E+09**	**7.1**
21	**05**	**2**	**1.00E+09**	**4**	**8.342**	**3.41E+10**	**1.62E+10**	**13.7**
22	05	5.00E+08	1.00E+09	8	4.089	2.41E+10	8.24E+09	21.5
23	05	2	5.00E+08	8	3.81	2.10E+10	8.11E+09	15.2
24	05	2	1.00E+09	8	7.934	4.36E+10	1.63E+10	14.7
25	**06**	**5.00E+08**	**1.00E+09**	**4**	**2.551**	**2.87E+10**	**8.59E+09**	**10.1**
26	**06**	**2**	**5.00E+08**	**4**	**2.373**	**2.66E+10**	**8.39E+09**	**6.7**
27	**06**	**2**	**1.00E+09**	**4**	**4.967**	**5.58E+10**	**1.70E+10**	**8.2**
28	06	5.00E+08	1.00E+09	8	2.936	6.29E+10	9.11E+09	7.5
29	06	2	5.00E+08	8	2.522	5.41E+10	8.98E+09	10.3
30	06	2	1.00E+09	8	6.237	1.36E+11	1.81E+10	8.4

Tabela 1 Uzyskane wyniki

Legenda:

01 – *division_seq.cpp*
02 – *division_par.cpp*
03 – *sieve_seq.cpp*
04 – *sieve_par_dom.cpp*
05 – *sieve_par_fun.cpp*
06 – *sieve_par_fun_dynamic.cpp*

Nr	Kod	F-E [%]	B-E [%]	MB [%]	CR [%]	ECPU [%]	a [1/s^2]	v [1/s]	PPE
1	**01**	**55.3**	**0**	**0**	**0**	**23.3**	**1.0**	**4.12E+04**	
2	**01**	**57.8**	**0**	**0**	**0**	**24**	**1.0**	**7.57E+04**	
3	**01**	**53.9**	**0**	**0**	**0**	**23.5**	**1.0**	**5.36E+04**	
4	02	62.9	0	0	0	88.5	3.05	1.26E+05	0.7625
5	02	72.4	0	0	0	72.2	2.57	1.94E+05	0.6425
6	02	58.1	0	0	0	74.1	2.58	1.38E+05	0.645
7	**02**	**38.8**	**0**	**0**	**0**	**93**	**5.43**	**2.23E+05**	**0.67875**
8	**02**	**56**	**0**	**0**	**0**	**83.6**	**4.5**	**3.41E+05**	**0.5625**
9	**02**	**38.5**	**0**	**0**	**0**	**85.1**	**4.53**	**2.43E+05**	**0.56625**
10	03	1.3	90.4	44.7	45.7	23.7	1.0	2.97E+06	
11	03	1.8	88.3	43.6	44.8	23.9	1.0	6.46E+06	
12	03	4.9	81.2	39.3	41.9	23.9	1.0	5.82E+06	
13	**04**	**0.4**	**96.3**	**47.9**	**48.3**	**93.3**	**1.23**	**3.65E+06**	**0.3075**
14	**04**	**2.7**	**88.5**	**43.2**	**45.4**	**88**	**1.2**	**7.76E+06**	**0.3**
15	**04**	**2.2**	**91.5**	**44.7**	**46.8**	**74.4**	**1.22**	**7.11E+06**	**0.305**
16	04	2.4	95.7	46.3	49.4	88.7	1.19	3.55E+06	0.14875
17	04	3.3	92.2	44.3	47.9	86.8	1.17	7.57E+06	0.14625
18	04	4.6	88.5	43	45.5	87.2	1.24	7.19E+06	0.155
19	**05**	**6**	**82.7**	**39.4**	**43.2**	**33.7**	**1.99**	**5.91E+06**	**0.4975**
20	**05**	**7.1**	**74.5**	**35.5**	**39**	**30.8**	**1.06**	**6.83E+06**	**0.265**
21	**05**	**3.6**	**82.4**	**39.6**	**42.7**	**30.1**	**1.05**	**6.10E+06**	**0.2625**
22	05	10.5	66.9	29.4	37.5	33.8	2.01	5.99E+06	0.25125
23	05	6	78.5	34.9	43.6	41.8	1.07	6.92E+06	0.13375
24	05	4.3	80.8	36.2	44.6	36	1.1	6.41E+06	0.1375
25	**06**	**4.1**	**85.2**	**39.6**	**45.6**	**88.8**	**3.23**	**9.60E+06**	**0.8075**
26	**06**	**3.8**	**89.2**	**43.2**	**46**	**86.1**	**1.72**	**1.11E+07**	**0.43**
27	**06**	**3.7**	**87.8**	**42.2**	**45.6**	**88.6**	**1.76**	**1.02E+07**	**0.44**
28	06	7.7	84.2	39.3	44.9	88.4	2.80	8.34E+06	0.35
29	06	5.2	84.9	39.6	45.4	88.1	1.62	1.05E+07	0.2025
30	06	6.8	84.3	40.3	44	89.4	1.4	8.15E+06	0.175

Tabela 2 Uzyskane wyniki

Legenda:

01 – *division_seq.cpp*
02 – *division_par.cpp*
03 – *sieve_seq.cpp*
04 – *sieve_par_dom.cpp*
05 – *sieve_par_fun.cpp*
06 – *sieve_par_fun_dynamic.cpp*

Opis numeracji:

1 – 3: **kod 01**, sekwencyjny,
4 – 6: **kod 02**, 4 wątki
7 – 9: **kod 02**, 8 wątków
10 – 12: **kod 03**, sekwencyjny
13 – 15: **kod 04**, 4 wątki
16 – 18: **kod 04**, 8 wątków
19 – 21: **kod 05**, 4 wątki
22 – 24: **kod 05**, 8 wątków
25 – 27: **kod 06**, 4 wątki
28 – 30: **kod 06**, 8 wątków

Testowane przedziały liczbowe:

1 – 3, 4 – 6, 7 – 9:
< **5.00E+06**, **1.00E+07** >
< **2**, **5.00E+06** >
< **2**, **1.00E+07** >

10 – 12, … , 28 – 30 :
< **5.00E+08**, **1.00E+09** >
< **2**, **5.00E+08** >
< **2**, **1.00E+09** >

4. Wnioski

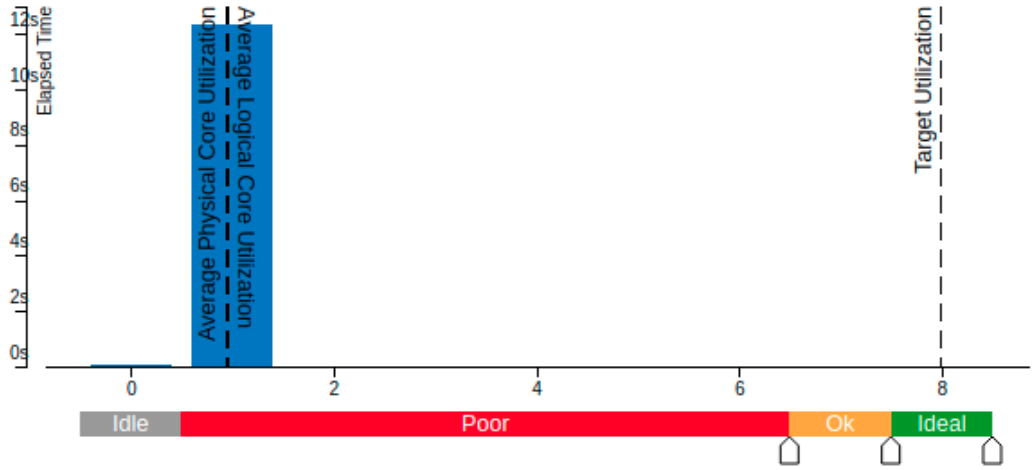
Eksperyment nr 1:

Effective Physical Core Utilization: 23.5% (0.941 out of 4)

Effective Logical Core Utilization: 12.0% (0.958 out of 8)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



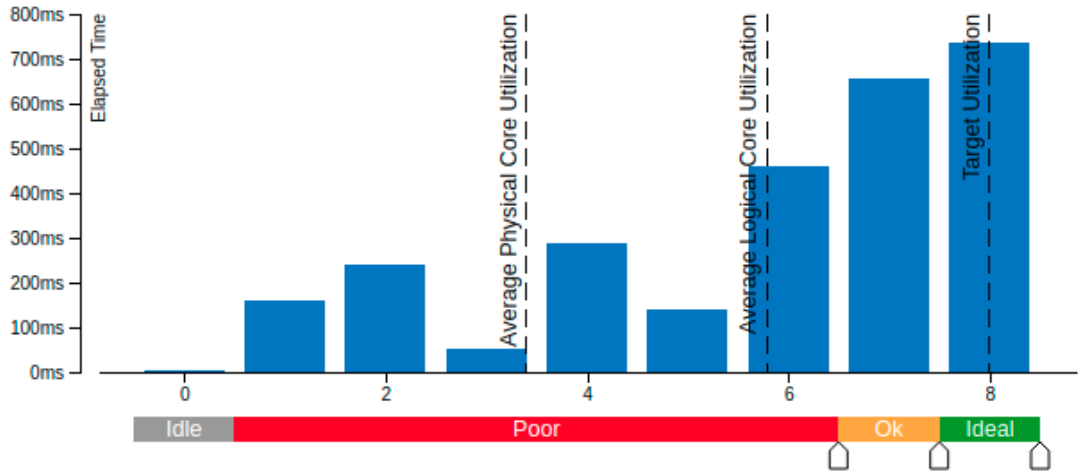
Zdjęcie 7 Zużycie procesora dla dzielenia sekwencyjnego – przedział <2,1+E7> (01)

Effective Physical Core Utilization: 85.1% (3.405 out of 4)

Effective Logical Core Utilization: 72.3% (5.787 out of 8)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



Zdjęcie 8 Zużycie procesora dla dzielenia równoległego – 8 wątków, przedział <2,1+E7> (02)

```
int isPrime(unsigned long x) {
    for(unsigned long i = 2; i * i <= x; i++) {
        if(x % i == 0) {
            return 0;
        }
    }
    return 1;
}
```

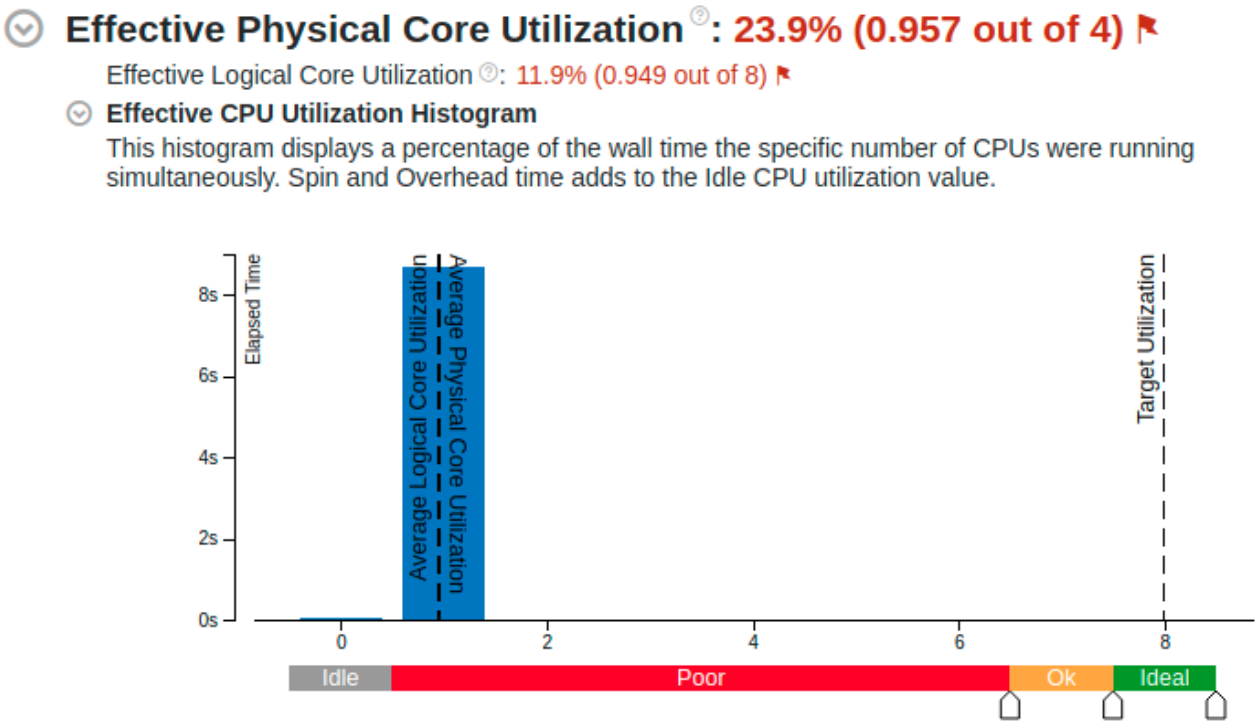
Zdjęcie 9 Mało efektywna funkcja, powód większego czasu przetwarzania na kolejnych wątkach.

W kodzie równoległym możemy zauważyć, że czasy pracy na poszczególnych wątkach rosną. Wynika to z faktu, że skorzystaliśmy z domyślnego szeregowania jakim jest ***schedule(static)***. Wątki, które dostały początkowe wartości iteracji, nie musiały sprawdzać tak dużych przedziałów liczbowych w celu odnalezienia liczb pierwszych. Efektywność programu zostałaby zwiększona, gdyby:

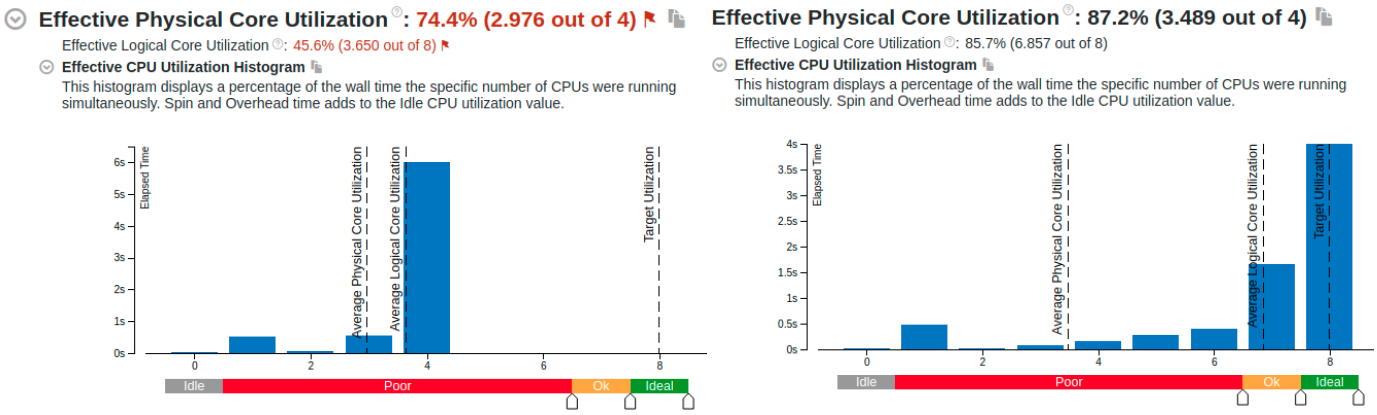
- następowałoby wykreślanie liczb złożonych, aby następne wątki nie musiały sprawdzać tych samych liczb,
- zastosowalibyśmy szeregowanie ***schedule(dynamic)***, w którym wątki gotowe do pracy zgłaszałyby się po kolejne iteracje, więc podział pracy byłby porównywalny.

W tym przypadku wąskim gardłem jest Front-End Bound. Oznacza to, że występują problemy związane z dostarczeniem zadania do Back-End'u. Stan ten wynika z zbyt wielu dostępów do pamięci na cykl. Ostatecznie czas wykonywania kodu równoległego był szybszy, a jego przyspieszenie osiągało wartości pomiędzy **4.5 a 5.5** w zależności od wariantu liczby wątków i przedziału liczb.

Eksperyment nr 2:



Zdjęcie 10 Zużycie procesora dla metody Sita Eratostenesa - kod sekwencyjny, przedział <2,1+E9> (03)



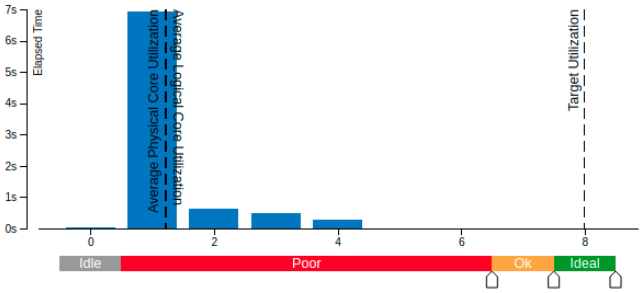
Zdjęcie 11 Zużycie procesora dla metody Sita Eratostenesa - kod równoległy, domenowy, przedział <2,1+E9>, kolejno wątki: 4,8 (04)

Effective Physical Core Utilization: 30.1% (1.205 out of 4)

Effective Logical Core Utilization: 15.3% (1.228 out of 8)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

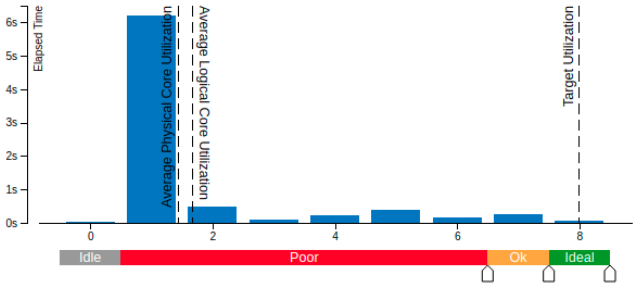


Effective Physical Core Utilization: 36.0% (1.440 out of 4)

Effective Logical Core Utilization: 21.0% (1.680 out of 8)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



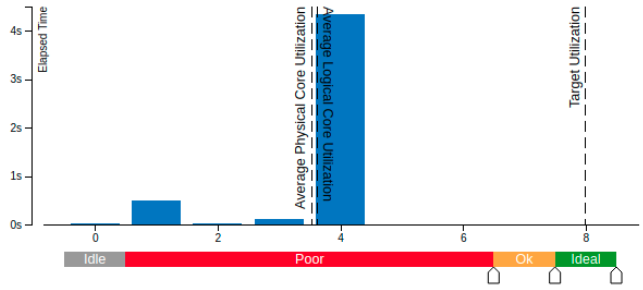
Zdjęcie 12 Zużycie procesora dla metody Sita Eratostenesa - kod równoległy, funkcyjny, przedział <2,1+E9>, kolejno wątki: 4,8 (05)

Effective Physical Core Utilization: 88.6% (3.544 out of 4)

Effective Logical Core Utilization: 45.3% (3.621 out of 8)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

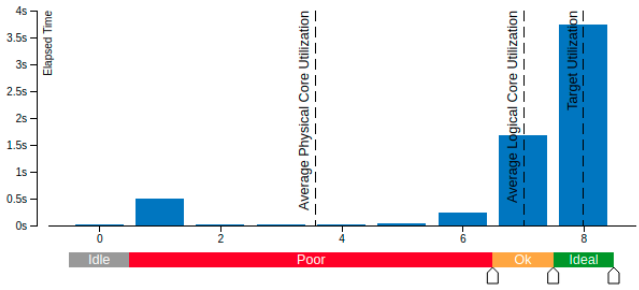


Effective Physical Core Utilization: 89.4% (3.575 out of 4)

Effective Logical Core Utilization: 87.6% (7.011 out of 8)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



Zdjęcie 13 Zużycie procesora dla metody Sita Eratostenesa - kod równoległy, funkcyjny, przedział <2,1+E9>, kolejno wątki: 4,8 (06)

Kod równoległy, funkcyjny wypada niewiele lepiej na tle kodu sekwencyjnego. Powodem takiego zachowania jest, tak jak w przypadku eksperymentu pierwszego – szeregowanie statyczne. Ponownie zauważamy, że jednemu wątkowi przydzielane jest najwięcej obliczeń, podczas gdy reszta ma ich dużo mniej.

Kod domenowy jest lepszy pod względem efektywnego wykorzystania procesora, natomiast pod względem czasu przetwarzania wypada gorzej od kodu funkcyjnego (średnie przysp.: **1.21** – domenowe, **1.38** – funkcyjne). Dzieje się tak, ponieważ procesy zapełniają globalny cache swoimi przedziałami sita, zatem zaczyna występować zjawisko Cache Missu, ponieważ przedziały te są unikalne i nie mają części wspólnych. Dochodzi w ten sposób do sytuacji, gdzie algorytm działa w sposób spowolniony, pomimo pełnego wykorzystania procesorów. Takiego zachowania nie zauważymy z kolei w rozwiązaniu funkcyjnym, gdzie wszystkie wątki pracują na tym samym segmencie i zachodzą tam z kolei Cache Hity. False sharing nie ma dużego wpływu na efektywność obu kodów.

Finalną częścią eksperymentu była poprawa gorzej zrównoleglonego kodu – wersji funkcyjnej. Zmiana szeregowania na dynamiczne znacząco wpłynęła na efektywne wykorzystanie procesora (ze średniej **34.4%** na **88.2%**). Powody, dlaczego zmiana szeregowania tak bardzo wpływa na przetwarzanie zostały przez nas omówione wyżej w sprawozdaniu wielokrotnie. Poprawę równoległości możemy również zauważyć w czasach przetwarzania. Kod funkcyjny z szeregowaniem statycznym osiągał średnie przyspieszenie na poziomie **1.38**, natomiast jego ulepszona wersja uzyskała średni wynik **2.09**.

We wszystkich wariantach równoległego podejścia Sita Eratostenesa zauważamy wysoki poziom Back-end Bound. Jest to spowodowane tym, że dzielenie jest operacją o dużym opóźnieniu i zbyt wiele operacji może być kierowanych do jednego portu wykonawczego (przykładowo, więcej operacji dzielenia docierających do Back-Endu na cykl niż procesory mogą je obsłużyć).

Tabela podsumowująca

Warianty	Nr testu	Kod	Min	Max	Liczba wątków	Elapsed[s]	Przyspieszenie
Dzielenie	7	02	5.00E+06	1.00E+07	8	1.415	5.43
Domenowe	18	04	2	1.00E+09	8	7.069	1.24
Funkcyjne	24	05	2	1.00E+09	8	7.934	3.23