Sprawozdanie z realizacji projektu 1 z laboratorium przetwarzania równoległego	
Imię i nazwisko: **Damian Mielczarek** **Mateusz Matkowski** E-mail: damian.mielczarek@student.put.poznan.pl mateusz.s.matkowski@student.put.poznan.pl	Data wykonania (wysyłki) zadania: **03.05.2022** Termin wymagany (wysyłki) zadania: **04.05.2022**
Termin zajęć laboratoryjnych: **środa- 9:45 tydzień parzysty** Numer grupy dziekańskiej: **L14**	Nr albumu: **145388** **145432**
Temat projektu: **Projekt1_OpenMP**	Wersja sprawozdania: **pierwsza**

1. Wstęp

Analiza efektywności algorytmów przetwarzania równoległego realizowanego na komputerze z procesorem wielordzeniowym z pamięcią współdzieloną, w problemie znajdowania liczb pierwszych metodą „dzielenia” oraz metodą sita Eratostenesa.

2. Opis wykorzystanego systemu obliczeniowego

Procesor	Intel Core i7-11800H
Liczba procesorów fizycznych (rdzeni)	8
Liczba procesorów logicznych (wątków)	16
Cache L1	640 KB
Bazowa częstotliwość	2,3 GHz
System operacyjny	Microsoft Windows 10 Home 64bit
Kompilator	mingw-w64-x86_64-gcc 11.3.0-1
Oprogramowanie do analizy	Visual Studio Code 1.66.2 i Intel VTune Profiler 2022.2

3. Warianty kodów
    a. Dzielenie sekwencyjne

```c
void allToDo(unsigned long n, unsigned long m){
    char *tab=(char*)malloc(n-m+1);
    memset(tab, '1', n-m+1);

    mainProcessing(tab,n,m);

    //printPrimes(tab,n,m);
    free(tab);
}

int main(int arc, char* argv[])
{
    if(arc!=3){
        return 0;
    }
    unsigned long m,n;
    m=atoll(argv[1]);
    n=atoll(argv[2]);

    allToDo(n,m);

    return 0;
}
```

Zdjęcie 1. Wersja kodu sekwencyjnego przy użyciu dzielenia (dzielenie_seq.c).

```c
int isprime(unsigned long x)
{
    for( unsigned long y = 2; y * y <= x; y++)
    {
        if(x % y == 0)
            return 0;
    }
    return 1;
}
void mainProcessing(char* tab, unsigned long n, unsigned long m){
    for(unsigned long i = m; i <= n ; i++)
    {
        if(!isprime(i)){
            tab[i-m] = '0';
        }
    }
}
```

Zdjęcie 2. Wersja kodu sekwencyjnego przy użyciu dzielenia (dzielenie_seq.c).

Jest to pierwsza wersja kodu wykorzystana do eksperymentu. Algorytm znajduje liczby z przedziału <MIN;MAX> poprzez wykrywanie niezerowej reszty z dzielenia przez a ≤ $\sqrt{MAX}$. Algorytm ten jest prosty w implementacji, jednak posiada złożoność obliczeniową rzędu O($\sqrt{n}$), ponieważ rozwiązanie wykonuje się na jednym wątku (działanie sekwencyjne) – rozłożenie pracy na większą ilość wątków (dostępnych w systemie) pozwoliłoby na skrócenie pracy całego programu – co widać w wersji równoległej przedstawionej poniżej.

b. Dzielenie równoległe wersja 1.

```
37  void allToDo(unsigned long n, unsigned long m){
38      char *tab = (char*)malloc(n-m+1);
39      memset(tab, '1', n-m+1);
40
41      #pragma omp parallel num_threads(omp_get_max_threads())
42      {
43          mainProcessing(tab, n, m);
44      }
45
46      //printPrimes(tab,n,m);
47      free(tab);
48  }
49  void mainProcessing(char* tab, unsigned long n, unsigned long m){
50      #pragma omp for schedule(dynamic) nowait
51      for(unsigned long i = m; i <= n; i++)
52      {
53          if(!isprime ( i )){
54              tab[i-m] = '0';
55          }
56      }
57  }
```

Zdjęcie 3. Wersja pierwsza kodu równoległego przy użyciu dzielenia (dzielenie_par.c).

W celu zmniejszenia ilości zdjęć, będziemy ukazywać fragmenty kodu, które uległy zmianie w porównaniu do wersji poprzednich.

Wersja ta jest bardzo podobna do wersji sekwencyjnej, jednak tym razem rozkładamy prace na wszystkie procesory logiczne (lub fizyczne), co pozwala na przyspieszenie obliczeń. Przy użyciu ***#pragma omp parallel*** tworzymy liczbę wątków podaną przez parametr ***num_threads*** (w tym przypadku zostanie utworzone 16 wątków). Korzystając z ***#pragma omp for schedule (dynamic)***, każdy proces otrzymuje jedną – kolejną nie wykonaną jeszcze iteracją z pętli for, kiedy skończy wykonywać poprzednią. Użycie klauzuli ***dynamic*** spowoduje w miarę sprawiedliwy podział pracy na wątki, tzn. na początku wątki mają do wykonania stosunkowo niewiele pracy ze względu na mniejszą liczbę dzielników do sprawdzenia, natomiast z kolejnymi iteracjami liczba dzielników zwiększa się. Wątki otrzymując po jednej liczbie do sprawdzenia, po kolei otrzymują coraz więcej „pracy", stąd określenie „sprawiedliwy podział pracy". W rozwiązaniu nie występuje wyścig (walka o dostęp do jednego adresu przez kilka wątków - co może sprawić potencjalnie różny (zazwyczaj błędny) wynik w zależności od kolejności dostępu do pamięci przez wątki) z tego względu, iż każda iteracja jest wykonywana tylko raz przez dokładnie jeden wątek i po zakończeniu sprawdzania tylko jeden wątek odwołuje się do jednego adresu z tablicy. Natomiast zjawisko false sharing'u[1] występuje - pojawia się ono właściwie przez cały czas przetwarzania programu, ponieważ wątki dostają po jednej iteracji, a co za tym idzie bardzo często ingerują w zapis jednej linii pamięci podręcznej (czyli potrzebne jest wiele synchronizacji) – przez co praca na procesorach w rzeczywistości nie jest zrównoważona. Jedyną synchronizacją w kodzie jest synchronizacja na końcu przetwarzania w linijce 40 (Zdjęcie 3), jednak nie wpływa to znacząco na czas obliczeń. W kolejnej wersji kodu spróbujemy pozbyć się zjawiska false sharing'u.

[1]false sharing- unieważnienie kopi linii pamięci podręcznej procesora - gdy jeden wątek zmieni wartość pod adresem występującym na tej samej linii pamięci, to inne wątki operujące na niej muszą zaktualizować obraz owej linii.

c. Dzielenie równoległe wersja 2.

```c
void allToDo(unsigned long n, unsigned long m){
    char *tab=(char*)malloc(n-m+1);
    memset(tab, '1', n-m+1);

    #pragma omp parallel num_threads(omp_get_max_threads())
    {
        mainProcessing(tab, n, m);
    }

    //printPrimes(tab, n, m);

    free(tab);
}
void mainProcessing(char* tab, unsigned long n, unsigned long m){
    #pragma omp for schedule(dynamic, 640000) nowait
    for(unsigned long i = m; i <= n ; i++)
    {
        if(!isprime(i)){
            tab[i-m]='0';
        }
    }
}
```

Zdjęcie 4. Wersja pierwsza kodu równoległego przy użyciu dzielenia (dzielenie_par_v2.c).

Jest to ulepszona wersja kodu dzielenie_par.c. Podzieliliśmy pracę dla każdego procesora na bloki po 640kB. Liczba ta jest wielkością pamięci cache pierwszego poziomu (dostęp do tego poziomu pamięci jest najszybszy), co uwalnia nas od problemu false sharing'u i znacząco podnosi wydajność programu oraz pozwala to na współbieżną pracę wszystkich wątków bez niepotrzebnej synchronizacji.

d. Sito Eratostenesa – podejście domenowe sekwencyjne

```c
unsigned long * find_primes(unsigned long max){
    char * prime = malloc(max + 1);
    memset(prime, '1', max + 1);
    unsigned long counter = max - 1;
    unsigned long squarter = sqrtl(max);

    for(unsigned long p=2; p<=squarter; p++){
        if(prime[p] == '1'){
            for(unsigned long i=p*p; i<=max; i+=p){
                if(prime[i] == '1'){
                    prime[i] = '0';
                    counter--;
                }
            }
        }
    }

    unsigned long found = 0;
    unsigned long *only_primes = malloc(sizeof(unsigned long) * (counter+1));
    for(unsigned long i = 2; i <= max; i++){
        if(prime[i] == '1'){
            only_primes[found+1] = i;
            found++;
        }
    }
    only_primes[0] = found;
    return only_primes;
}
```

Zdjęcie 5. Funkcja wyznaczająca liczby pierwsze z przedziału (2..$\sqrt{MAX}$) wykorzystywana w kodach (sito_fun_seq.c, sito_fun_par.c, sito_fun_par_v2.c, sito_dom_seq.c, sito_dom_par.c).

Powyższą funkcję wykorzystujemy w kolejnych wersjach sita Eratostenesa.

```
75  void allToDo(unsigned long n, unsigned long m){
76      char *tab = (char*)malloc(n-m+1);
77      memset(tab, '1', n-m+1);
78
79      unsigned long * only_primes = find_primes(sqrtl(n));
80
81      mainProcessing(tab,only_primes, n, m);
82
83      printPrimes(tab,n,m);
84      free(only_primes);
85      free(tab);
86  }
87
88  int main(int arc, char* argv[])
89  {
90      if(arc!=3){
91          return 0;
92      }
93      unsigned long m,n;
94      m = atoll(argv[1]);
95      n = atoll(argv[2]);
96
97      allToDo(n,m);
98
99      return 0;
100 }
```

Zdjęcie 6. Wersja kodu przy użyciu sita Eratostenesa i podejścia domenowego (kod sito_dom_seq.c). Największą zmianą w stosunku do poprzednich kodów jest zmiana w procedury mainProcessing (patrz Zdjęcie 7).

```
52  void mainProcessing(char* tab, unsigned long* only_primes, unsigned long n, unsigned long m){
53      unsigned long chunk_size = 640000; // L1 or L2 cache size (?)
54      unsigned long all_iterations = (n-m) / chunk_size;
55
56      unsigned long beginit;
57      unsigned long finit;
58
59      for(unsigned long k = 0; k <= all_iterations; k++)
60      {
61          beginit = m + k * chunk_size + k;
62          finit = fminl(n, m + (k+1) * chunk_size + k);
63
64          for(unsigned long i=1; i<=only_primes[0]; i++)
65          {
66              for(unsigned long j=only_primes[i]*2; j<=finit; j+=only_primes[i]){
67                  if(j>=beginit){
68                      tab[j-m] = '0';
69                  }
70              }
71          }
72      }
73  }
```

Zdjęcie 7. Procedura mainProcessing w kodzie sekwencyjnym podejścia domenowego sita Eratostenesa (kod sito_dom_seq.c).

W procedurze mainProcessing, chunk_size jest rozmiarem pamięci cache pierwszego poziomu. Dzięki niemu jesteśmy w stanie policzyć liczbę iteracji potrzebną do prawidłowego (i szybkiego) wyznaczenia liczb pierwszych z przedziału m..n (MIN..MAX). W przypadku podejścia sekwencyjnego optymalizacja ta nie wpływa na działanie programu, ponieważ tylko jeden wątek wykonuje owe operacje, co za tym idzie wspomniane wyżej zjawiska false sharing'u oraz wyścigu nie występują. Jednak tak napisany kod jest wersją, która w łatwy sposób pozwoli na płynne przejście do zrównoleglenia rozwiązania.

e. Sito Eratostenesa – podejście domenowe równoległe

```
53  void mainProcessing(char* tab, unsigned long* only_primes, unsigned long n, unsigned long m){
54      unsigned long chunk_size = 640000; // L1 or L2 cache size (?)
55      unsigned long all_iterations = (n-m) / chunk_size;
56
57      unsigned long beginit;
58      unsigned long finit;
59
60      #pragma omp for schedule(guided, 1) nowait
61      for(unsigned long k = 0; k <= all_iterations; k++)
62      {
63          beginit = m + k * chunk_size + k;
64          finit = fminl(n, m + (k+1) * chunk_size + k);
65
66          for(unsigned long i=1; i<=only_primes[0]; i++)
67          {
68              for(unsigned long j=only_primes[i]*2; j<=finit; j+=only_primes[i]){
69                  if(j>=beginit){
70                      tab[j-m] = '0';
71                  }
72              }
73          }
74      }
75  }
76  void allToDo(unsigned long n, unsigned long m){
77      char *tab = (char*)malloc(n-m+1);
78      memset(tab, '1', n-m+1);
79      unsigned long * only_primes = find_primes(sqrtl(n));
80
81      #pragma omp parallel num_threads(omp_get_max_threads())
82      {
83          mainProcessing(tab, only_primes, n, m);
84      }
85
86      //printPrimes(tab,n,m);
87      free(only_primes);
88      free(tab);
89  }
```

Zdjęcie 8. Procedura mainProcessing oraz allToDo w kodzie równoległym podejścia domenowego sita Eratostenesa (kod sito_dom_par.c).

W kodzie równoległym podejścia domenowego sita Eratostenesa obliczana jest liczba iteracji potrzebna do przetworzenia całej tablicy wykreśleń, na podstawie wielkości zbioru MIN..MAX oraz parametru chunk_size, który jest wielkością pamięci cache pierwszego poziomu. Następnie wątkom przydzielane są iteracje przy użyciu klauzuli (***guided,1)***, czyli rozmiar każdego przypisywanego zbioru iteracji jest proporcjonalny do wielkości: ***liczba_nie_przydzielonych_iteracji /liczba_wątków***. Gdy proces zakończy przetwarzanie swojego bloku pracy, dostaje przydział kolejnych iteracji, których liczba zmierza do 1. Ten sposób podziału pracy pozwoli na przydzielenie większych ilości iteracji na początku, kiedy jest najmniej wykreślania (zawsze wykreślamy od 2 do końca aktualnego bloku). Natomiast pod koniec liczba iteracji jest bliska 1, ponieważ jest wtedy dużo więcej wykreślania (od 2 do (prawie) MAX), co pozwala na przyspieszenie przetwarzania i większe zrównoważenie pracy wątków. Wyścig w tym przetwarzaniu nie występuje z tego względu, iż po podziale na iteracje każdy wątek otrzymał różny podzbiór zadań, do których dostępu nie mają inne wątki. Postaraliśmy się, tak jak w poprzedniej wersji kodu (dzielenie_par_v2.c), uniknąć zjawiska false sharing'u poprzez ustawienie parametru chunk_size na wielkość pamięci cache pierwszego poziomu - przez co potencjalnie każdy blok będzie znajdował się na osobnej linii pamięci. Synchronizacja, tak jak w poprzednich kodach równoległych, występuje w linii 84 (Zdjęcie 8), jednak nie wpływa znacząco na czas obliczeń. Jedynym problemem w kodzie jest brak zrównoleglenia dla wyznaczenia początkowej tablicy liczb pierwszych (only_primes), której to zrównoleglenie wyznaczania mogłoby prowadzić do problemów takich jak chociażby false sharing, który wpłynąłby znacząco na czas przetwarzania.

f. Sito Eratostenesa – podejście funkcyjne sekwencyjne

```
53 void mainProcessing(char* tab, unsigned long* only_primes,  unsigned long n, unsigned long m){
54     for(unsigned long i=1; i<=only_primes[0]; i++)
55     {
56         for(unsigned long j=only_primes[i]*2; j<=n; j+=only_primes[i]){
57             if(j>=m){
58                 tab[j-m] = '0';
59             }
60         }
61     }
62 }
63
64 void allToDo(unsigned long n, unsigned long m){
65     char *tab = (char*)malloc(n-m+1);
66     memset(tab, '1', n-m+1);
67     unsigned long * only_primes = find_primes(sqrtl(n));
68     mainProcessing(tab, only_primes, n, m);
69
70     //printPrimes(tab, n, m);
71     free(only_primes);
72     free(tab);
73 }
```

Zdjęcie 9. Procedura mainProcessing oraz allToDo w kodzie sekwencyjnym podejścia funkcyjnego sita Eratostenesa (kod sito_fun_seq.c).

Jest to kod sekwencyjny przygotowany na potrzeby eksperymentu. Wcześniej przygotowana tablica only_primes (liczby pierwsze do $\sqrt{MAX}$ – (funkcja ze Zdjęcia 5)) służy do wykreślenia poszczególnych wielokrotności liczb pierwszych. Wydaje się, iż jest to nieefektywna wersja, ponieważ brak zrównoleglenia (praca na jednym wątku) znacząco spowalnia przetwarzanie.

g. Sito Eratostenesa – podejście funkcyjne równoległe wersja 1.

```
53 void mainProcessing(char* tab,unsigned long* only_primes,  unsigned long n, unsigned long m){
54     #pragma omp for schedule(dynamic,1) nowait
55     for(unsigned long i=1; i>=only_primes[0]; i++)
56     {
57         for(unsigned long j=only_primes[i]*2; j<=n; j+=only_primes[i]){
58             if(j>=m){
59                 tab[j-m] = '0';
60             }
61         }
62     }
63 }
64 void allToDo(unsigned long n, unsigned long m){
65     char *tab = (char*)malloc(n-m+1);
66     memset(tab, '1', n-m+1);
67     unsigned long * only_primes = find_primes(sqrtl(n));
68
69     #pragma omp parallel num_threads(omp_get_max_threads())
70     {
71         mainProcessing(tab,only_primes, n, m);
72     }
73
74     //printPrimes(tab,n,m);
75     free(only_primes);
76     free(tab);
77 }
```

Zdjęcie 10. Procedura mainProcessing oraz allToDo w kodzie równoległym podejścia funkcyjnego sita Eratostenesa (kod sito_fun_par.c).

W tej wersji podejścia funkcyjnego zrównoleglilismy jedynie obszar wykreślania liczb pierwszych. Iteracje dzielimy „równo” na wszystkie wątki poprzez zastosowanie klauzuli ***dynamic***, która to przydziela po jeden iteracji na wcześniej przygotowanym zbiorze liczb pierwszych (only_primes), gdy tylko wątek zakończy iterację poprzednią. Klauzula ta zrównoważy pracę na wszystkich wątkach, ponieważ każdy wątek dostanie jakąś iterację do wykonania. W rozwiązaniu tym występuje problem wyścigu, lecz nie wpływa to na poprawność i prędkość przetwarzania (nic złego nie dzieje się jeśli dwa wątki próbują wpisać pod adres wartość ‘0’). Problemem w tej wersji jest zjawisko false sharing’u – procesy otrzymując po jednej iteracji muszą często synchronizować się z pamięcią podręczna, a co za tym idzie spowalnia to przetwarzanie. Dodatkową synchronizacją (jednak nie wpływającą znacząco na czas przetwarzania) jest synchronizacja w linii 72 (Zdjęcie 10).

h. Sito Eratostenesa – podejście funkcyjne równoległe wersja 2.

```
53  void mainProcessing(char* tab,unsigned long* only_primes,  unsigned long n, unsigned long m){
54      #pragma omp for schedule(guided,1) nowait
55      for(unsigned long i=1; i>=only_primes[0]; i++)
56      {
57          for(unsigned long j=only_primes[i]*2; j<=n; j+=only_primes[i]){
58              if(j>=m){
59                  tab[j-m] = '0';
60              }
61          }
62      }
63  }
```

Zdjęcie 11. Procedura mainProcessing w kodzie równoległym podejścia funkcyjnego sita Eratostenesa (kod sito_fun_par_v2.c).

Kolejną wersją kodu jest wersja z klauzulą ***guided***, która rozdziela pracę na procesy w sposób opisany w podpunkcie „e”. Zatem pierwszy wątek dostanie „najwięcej pracy”, to oznacza, że musi wykreślić więcej liczb złożonych. Równocześnie ostatnie przydziały są mniejsze oraz jest mniej liczb do wykreślania, przez co prawdopodobnie ostatnie procesy skończą działanie szybciej, a co za tym idzie, problem false sharing’u zostanie zminimalizowany – czyli potencjalnie przetwarzanie wykona się szybciej. Problem wyścigu jest taki sam jak w poprzednim punkcie.

4. Eksperyment obliczeniowo-pomiarowy

Numer porządkowy	Nazwa pliku (.c)	Rozmiar instancji	Liczba procesorów
1.	dzielenie_seq	<2; 750.000.000>	1
2.	dzielenie_seq	<2; 375.000.000>	1
3.	dzielenie_seq	<375.000.000; 750.000.000>	1
4.	dzielenie_par	<2; 750.000.000>	8
5.	dzielenie_par	<2; 375.000.000>	8
6.	dzielenie_par	<375.000.000; 750.000.000>	8
7.	dzielenie_par	<2; 750.000.000>	16
8.	dzielenie_par	<2; 375.000.000>	16
9.	dzielenie_par	<375.000.000; 750.000.000>	16
10.	dzielenie_par_v2	<2; 750.000.000>	8
11.	dzielenie_par_v2	<2; 375.000.000>	8
12.	dzielenie_par_v2	<375.000.000; 750.000.000>	8
13.	dzielenie_par_v2	<2; 750.000.000>	16
14.	dzielenie_par_v2	<2; 375.000.000>	16
15.	dzielenie_par_v2	<375.000.000; 750.000.000>	16

16.	sito_fun_seq	<2; 750.000.000>	1
17.	sito_fun_seq	<2; 375.000.000>	1
18.	sito_fun_seq	<375.000.000; 750.000.000>	1
19.	sito_fun_par	<2; 750.000.000>	8
20.	sito_fun_par	<2; 375.000.000>	8
21.	sito_fun_par	<375.000.000; 750.000.000>	8
22.	sito_fun_par	<2; 750.000.000>	16
23.	sito_fun_par	<2; 375.000.000>	16
24.	sito_fun_par	<375.000.000; 750.000.000>	16
25.	sito_fun_par _v2	<2; 750.000.000>	8
26.	sito_fun_par _v2	<2; 375.000.000>	8
27.	sito_fun_par _v2	<375.000.000; 750.000.000>	8
28.	sito_fun_par _v2	<2; 750.000.000>	16
29.	sito_fun_par _v2	<2; 375.000.000>	16
30.	sito_fun_par _v2	<375.000.000; 750.000.000>	16
31.	sito_dom_seq	<2; 750.000.000>	1
32.	sito_dom_seq	<2; 375.000.000>	1
33.	sito_dom_seq	<375.000.000; 750.000.000>	1
34.	sito_dom_par	<2; 750.000.000>	8
35.	sito_dom_par	<2; 375.000.000>	8
36.	sito_dom_par	<375.000.000; 750.000.000>	8
37.	sito_dom_par	<2; 750.000.000>	16
38.	sito_dom_par	<2; 375.000.000>	16
39.	sito_dom_par	<375.000.000; 750.000.000>	16

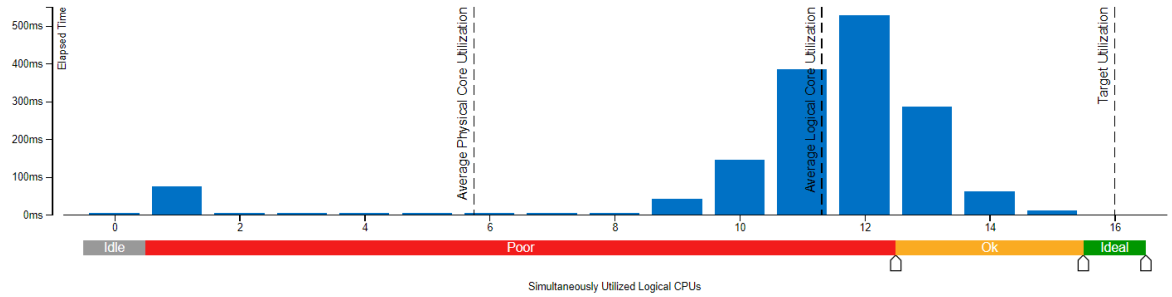
Tabela 1. Zbiorcza informacja na temat kodu - instancji testowej - liczby użytych procesorów. W celu lepszej orientacji w następnej tabeli będziemy odwoływać się jedynie do numerów porządkowych.

Numer porządkowy	Elapsed time [s]	Instruction retired	Clockticks	Retiring [%]	Front-end bound [%]	Back-end bound [%]	Memory bound [%]	Core bound [%]	Effective physical core utilization [%]	Acceleration	Processing speed [1/s]	Parallel processing efficiency
1.	1194,048	8,4E+12	5,09E+12	36,5	2,7	60,5	0,1	60,3	12,0	---	6,28E+05	---
2.	446,502	3,10E+12	1,88E+12	36,4	2,7	60,1	0,2	60,0	12,0	---	8,40E+05	---
3.	761,584	5,30E+12	3,21E+12	36,6	2,7	60,5	0,2	60,4	12,0	---	4,92E+05	---
4.	208,109	8,80E+12	5,98E+12	36,0	5,6	57,4	4,1	53,3	76,2	51,10	3,60E+06	6,39
5.	90,330	3,32E+12	2,37E+12	34,7	6,8	57,2	6,5	50,6	68,4	22,18	4,15E+06	2,77
6.	117,351	5,48E+12	3,61E+12	37,0	4,8	57,6	2,3	55,3	82,3	28,81	3,20E+06	3,60
7.	225,675	9,29E+12	1,03E+12	38,5	16,6	41,7	8,4	33,3	70,5	55,41	3,32E+06	3,46
8.	108,361	3,60E+12	4,19E+12	36,2	17,6	42,9	12,5	30,4	61,4	26,60	3,46E+06	1,66
9.	118,041	5,70E+12	6,07E+12	40,6	15,9	40,6	5,7	34,9	77,6	29,00	3,18E+06	1,81
10.	179,447	8,41E+12	5,80E+12	37,4	5,0	56,8	0,2	56,6	81,9	44,06	4,18E+06	5,51
11.	67,086	3,10E+12	2,16E+12	37,4	5,2	56,7	0,2	56,5	80,3	16,47	5,59E+06	2,06
12.	113,097	5,30E+12	3,65E+12	37,4	5,0	56,6	0,2	56,4	81,4	27,77	3,32E+06	3,47
13.	153,529	8,41E+12	9,54E+12	38,8	16,4	42,0	0,2	41,8	92,8	37,69	4,89E+06	2,36
14.	57,050	3,10E+12	3,50E+12	38,9	16,3	41,9	0,2	41,7	91,6	14,01	6,57E+06	0,88
15.	96,854	5,31E+12	6,01E+12	38,7	16,3	42,1	0,2	42,0	93,1	23,78	3,87E+06	1,49
16.	7,752	1,63E+10	3,25E+10	8,4	1,5	91,2	39,0	52,3	12,2	---	9,67E+07	---
17.	4,073	7,95E+09	1,64E+10	9,0	1,8	96,0	40,4	55,6	11,7	---	9,21E+07	---
18.	4,184	1,40E+10	1,74E+10	11,0	2,0	88,0	37,0	51,0	11,4	---	8,96E+07	---
19.	2,678	1,81E+10	8,04E+10	5,6	2,0	91,8	41,1	50,7	65,5	0,66	2,80E+08	0,08
20.	1,230	8,90E+09	3,66E+10	6,0	2,4	88,9	39,3	49,6	58,5	0,32	3,05E+08	0,04
21.	1,613	1,39E+10	4,85E+10	6,5	3,7	92,6	42,6	50,0	63,7	0,40	2,32E+08	0,05
22.	2,894	1,82E+10	1,35E+11	4,9	2,9	92,5	42,0	50,4	70,6	0,71	2,59E+08	0,04
23.	1,242	8,94E+09	5,88E+10	5,4	3,4	88,1	41,7	46,4	68,2	0,35	3,02E+08	0,02
24.	1,543	1,39E+10	7,30E+10	5,9	4,1	89,6	41,1	48,5	71,9	0,38	2,43E+08	0,02
25.	0,335	3,85E+12	1,65E+12	53,0	14,6	76,9	28,4	48,6	16,8	0,08	2,24E+09	0,01
26.	0,158	1,94E+12	8,23E+08	41,1	6,6	34,9	12,7	22,2	12,8	0,04	2,37E+09	0,005
27.	0,233	2.88E+12	1,06E+12	43,9	22,2	32,6	14,7	17,9	10,9	0,06	1,61E+09	0,007
28.	0,323	3,88E+12	2,26E+12	40,1	8,1	55,3	13,7	41,6	16,4	0,08	2,32E+09	0,005
29.	0,171	1,96E+12	1,28E+12	33,1	6,0	50,8	13,5	37,4	17,8	0,04	2,19E+09	0,003
30.	0,230	2,89E+12	1,40E+12	40,2	20,6	39,1	15,7	23,4	15,6	0,06	1,63E+09	0,004
31.	294,288	5,70E+12	1,24E+12	58,5	40,4	0,7	0,1	0,6	11,8	---	2,55E+06	---
32.	72,787	1,41E+12	3,06E+11	58,4	40,0	0,8	0,1	0,7	11,7	---	5,15E+06	---
33.	220,249	4,27E+12	9,26E+11	58,5	40,4	0,7	0,1	0,6	11,7	---	1,70E+06	---
34.	57,863	5,70E+12	1,83E+12	50,4	51,5	0,8	0,0	0,8	81,0	14,21	1,30E+07	1,78
35.	14,378	1,41E+12	4,52E+11	50,4	50,3	0,9	0,0	0,9	77,4	3,53	2,61E+07	0,44
36.	43,578	4,27E+12	1,38E+12	50,8	51,8	0,8	0,0	0,8	79,8	10,70	8,61E+06	1,34
37.	39,072	5,70E+12	2,34E+12	58,9	45,5	0,4	0,0	0,4	89,9	9,60	1,92E+07	0,60
38.	9,598	1,41E+12	5,80E+11	59,1	41,5	0,4	0,0	0,4	91,0	2,36	3,90E+07	0,15
39.	28,857	4,27E+12	1,77E+12	59,2	41,7	0,3	0,0	0,3	91,6	7,08	1,30E+07	0,44

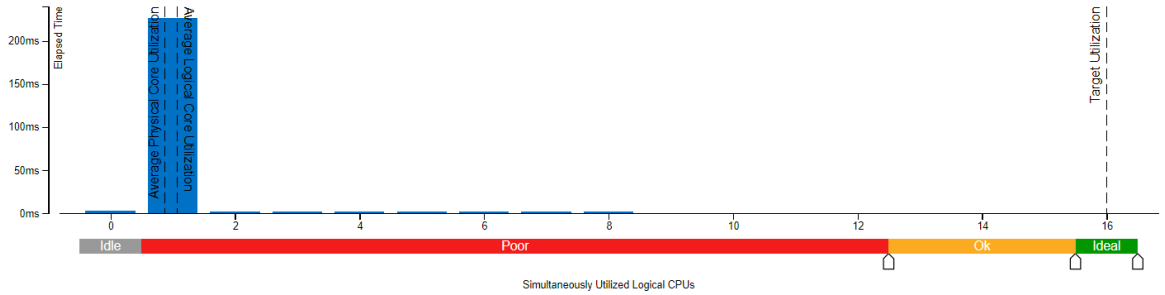
Tabela 2. Zbiorcza informacja na temat badanych parametrów wymienionych w zadaniu przy użyciu narzędzia VTune.

5. Wnioski

Z powyższej tabeli wynika, iż najlepszym algorytmem dla problemu znajdowania liczb pierwszych jest wersja sito_fun_par_v2.c (numery 25-30). Uniknięcie false sharing'u przy użyciu klauzuli guided znacząco zwiększa efektywność przetwarzania. Efektywność nie skaluje się ze wzrostem liczby procesorów. To może być spowodowane zbyt małą instancją dla tego problemu – nie można było ustawić większego przedziału, ponieważ i tak przy takich wartościach początkowe kody np. dzielenie_seq.c (numery 1-3) wykonują się stanowczo za długo i nie bylibyśmy w stanie porównać ich w tabeli. Kolejnym podejściem, które radzi sobie równie dobrze (jednak 10 razy gorzej), jest podejście z klauzulą dynamic (sito_fun_par.c (numery 20-24)) jednak istnieje zauważalna różnica dzięki zjawisku false sharing'u. Klauzula guided zmniejsza zrównoleglenie (wyjaśnienie w podpunkcie „g"). Natomiast klauzula dynamic zwiększa zrównoleglenie, jednak kosztem wolniejszej pracy wątków (false sharing). Podejście funkcyjne sekwencyjne jest trzecim najlepszym algorytmem, jednak w porównaniu do wersji kodów z klauzulą dynamic jest 4 razy wolniejsza (oczywiście przez brak zrównoleglenia). Podejścia „dzielenie równoległe", podejście domenowe oraz pozostałe podejścia sekwencyjne nie są efektywne – obliczenia zajmują zbyt wiele czasu. Algorytm domenowy równoległy radzi sobie całkiem dobrze (około 2x gorzej od podejścia funkcyjnego sekwencyjnego) tylko dla mniejszego przedziału (2..MAX/2) (numer 38).

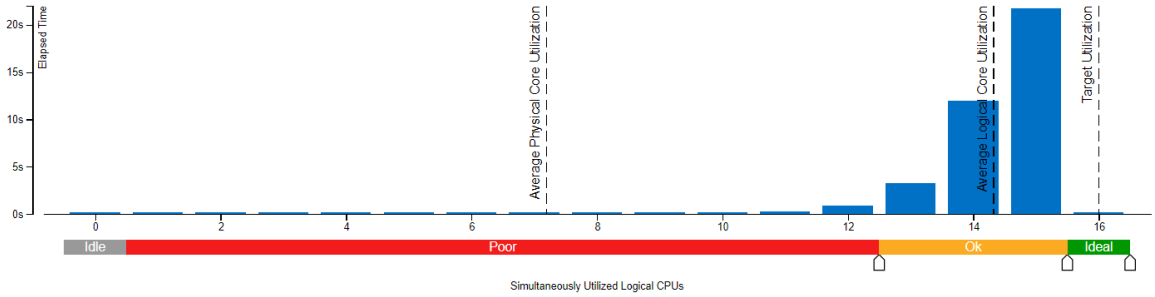


Histogram 1. Przykładowy histogram efektywnego wykorzystania rdzeni fizycznych procesora dla podejścia funkcyjnego z klauzula dynamic.



Histogram 1. Przykładowy histogram efektywnego wykorzystania rdzeni fizycznych procesora dla podejścia funkcyjnego z klauzula guided.

Najlepsze jakości zrównoleglenia przetwarzania można zauważyć w podejściu domenowym (numery 34-39), gdyż najdłuższy czas jest poświęcony na równoczesne wykorzystanie wszystkich wątków. Dobrym wykorzystaniem zrównoleglenia cechują się również kody „dzielenie równoległe" (numery 4-15). Dane z tabeli są dobrą wizualizacją, że zrównoważone wykorzystanie rdzeni fizycznych i logicznych nie zawsze doprowadza do obliczeń o najkrótszym czasie. Natomiast najgorszym pod tym względem okazał się kod sito_fun_par_v2.c – powód podany jest na początku punktu 5.



Histogram 3. Przykładowy histogram efektywnego wykorzystania rdzeni fizycznych procesora dla podejścia domenowego.

Głównym ograniczeniem efektywnościowym w rozwiązaniach (w których? – wyjaśniliśmy w punkcie 3 sprawozdania) jest zjawisko false sharing'u. Wpływa ono znacząco na czasy poszczególnych obliczeń, przykładowo podejście sita - równoległe funkcyjne (sito_fun_par.c) jest 10x razy wolniejsze od jego ulepszonej wersji (sito_fun_par_v2), w której pojawianie się tego zjawiska zostało zminimalizowane (kosztem zrównoleglenia) – numery w tabeli odpowiednio 20-24 oraz 25-30.

Podejścia dzielenia równoległe są mało efektywne ze względu na ilość sprawdzanych dzielników. Pomimo, iż wykonują się równolegle, to i tak jest to mało efektywne. Ratunkiem, gdy nie mamy innego kodu pod ręką, może być użycie wielkości chunk o wartości wielkości pamięci cache poziomu pierwszego (numery 4-9 oraz 10-15 ukazują przyspieszenie o około 18%).

Najlepsze podejście równoległe opisaliśmy powyżej, natomiast najgorszym podejściem równoległym okazało się podejście domenowe (najlepszy wynik numer 38). Przewaga podejścia funkcyjnego jest niezależna od liczby użytych procesorów i rozmiaru testowanej instancji. Przyczyną takich wyników jest to, że w każdej iteracji wykreślamy liczby od 2 do końca bloku (Zdjęcie 8- linia kodu: 68), co jak pokazuje eksperyment nie jest najlepszym podejściem (podejście sekwencyjne funkcyjne działa około 2x lepiej i jest niezależne od rozmiaru instancji).

Odwołując się do miary „retiring", najmniejszym procentowym udziałem wykorzystanych zasobów procesora cechują się warianty równoległe sita w podejściu funkcyjnym. Wąskim gardłem dla tych podejść okazała się wartość udziału procentowego w ograniczeniu efektywności przetwarzania części wyjściowej procesora (Back-end bound) (numery 19-30). Natomiast podejście domenowe równoległe oraz dzielenie równoległe cechują się około 10x większym wykorzystaniem zasobów procesora, a wąskim gardłem dla nich okazał się udział procentowy w ograniczeniu efektywności przetwarzania części wejściowej procesora (Front-end bound).

Numer porządkowy	Elapsed time [s]	Acceleration	Effective physical core utilization [%]
2.	446,502	---	---
14.	57,050	37,69	91,6
17.	4,073	---	---
26.	0,158	---	---
32.	72,787	---	---
38.	9,598	---	91,0

Tabela 3. Tabela podsumowująca. Parametr „---„ nie jest kluczowa dla wyniku.