Projekt 1 OMP

Wersja pierwsza

Przetwarzanie Równoległe - Laboratorium

studia dzienne kierunek Informatyka rok a. 2021/2022

Grupa: L15; Godzina: Środa 8:00 parzyste

Autorzy: Maciej Łuczak 145366, Michał Linde 145408

maciej.ar.luczak@student.put.poznan.pl

michal.linde@student.put.poznan.pl

Terminy oddania sprawozdania z projektu

Wymagany: 4 maja 2022 Rzeczywisty: 3 maja 2022

Krótki opis treści zadania

Zakres projektu obejmuje analizę problemu i opracowanie kodów do znajdowania liczby pierwszych w podanym przedziale <M..N> oraz ponad to analizę efektywności przetwarzania równoległego przygotowanych kodów realizowanych na komputerze z procesorem wielordzeniowym z współdzieloną pamięcią. W ramach projektu zaimplementowano różne wersje algorytmów znajdowania liczb pierwszych, sekwencyjne oraz równoległe w tym analizę ich efektywności.

Algorytmy sekwencyjnego znajdowania liczb pierwszych wykorzystują metodę testowania każdej liczby z wskazanego do badań przedziału i sprawdza czy jest podzielna przez liczby mniejsze. Do odsiania, nieinteresujących liczb złożonych została wykorzystana metoda Sita która, usuwa z badanego zbioru liczby będące liczbami złożonymi. Równoległe podejście przekształca algorytm sekwencyjny tak aby uzyskać różne warianty zrównoleglenia realizując podejście domenowe oraz funkcyjne. Zadanie realizowane równolegle z wykorzystaniem podejścia domenowego wykreśla wszystkie wymagane wielokrotności podzbioru liczb pierwszych, natomiast, podejście funkcyjne wykreśla wielokrotności mających ograniczony zakres wartości, ale będących wielokrotnościami wymaganych liczb pierwszych. Analiza efektywności opiera się na analizie oceny efektywności przeprowadzonej z wykorzystaniem oprogramowania Intel VTune Profiler oraz zaprezentowaniu otrzymanych wyników.

1. Prezentacja wykorzystanych technologii

Procesor	8th Generation Intel® Core™ i5 Processors
Liczba procesorów fizycznych	4
Liczba procesorów logicznych	8
Oznaczenie procesora	i5-8300H
Wielkość pamięci podręcznych	8 MB Intel® Smart Cache
System operacyjnego	Ubuntu
IDE	Visual Studio Code
Oprogramowanie do oceny efektywności	Intel® VTune™ Profiler

Tabela1. Wykaz wykorzystywanego systemu obliczeniowego do realizacji projektu

2. Prezentacja przygotowanych wariantów kodu

2.1 Algorytmy sekwencyjne

Algorytmy sekwencyjnego znajdowania liczb pierwszych wykorzystują metodę testowania każdej liczby z wskazanego do badań przedziału, sprawdzając czy jest podzielna przez liczby mniejsze. Do odsiania, nieinteresujących liczb złożonych została wykorzystana metoda Sita która, usuwa z badanego zbioru liczby będące liczbami złożonymi.

2.1.1 Sekwencyjny – metoda dzielenie: *SEQ_D*

```
19  int main()
20  {
21      int sum = 0;
22      long primeNumbers[MAX - MIN];
23      long indice = 0;
24      int start;
25
26      if (MIN < 2)
27      {
28          start = 2;
29      }
30      else {
31          start = MIN;
32      }
33
34      auto chronostart = chrono::steady_clock::now();
35      for (int i = start; i <= MAX; i++)
36      {
37          if (isprime(i))
38          {
39              primeNumbers[indice] = i;
40              indice++;
41              sum++;
42          }
43      }
44      auto chronoend = chrono::steady_clock::now();
45      return 0;
46  }
```

Rysunek 1. Kod Funkcji int main() algorytmu sekwencyjnego z dzielenie.

```
10  int isprime(int x)
11  {
12      for (int y = 2; y * y <= x; y++)
13      {
14          if (x % y == 0)
15              return 0;
16      }
17      return 1;
18  }
```

Rysunek 2. Kod funkcji int isprime(int x) algorytmu sekwencyjnego z dzieleniem.

Zaprezentowany algorytm sekwencyjny z wykorzystaniem metody dzielenia (Rysunek 1.) w funkcji main() inicjalizuje zmienne: sum - ilość znalezionych liczb pierwszych, primeNumbers – tablica o rozmiarze równej zakresowi liczb badanego przedziału, start- początkowa wartość. Pierwsza instrukcja if-else zapewnia, iż najmniejsza liczba z badanego przedziału to liczba 2. Następnie wykonywana pętla for dla każdej liczby z badanego przedziału wywołuje funkcję isprime(int liczba) z rysunku 2. sprawdzając za pomocą metody dzielenia czy liczba jest liczbą pierwszą bądź też nie. Instrukcja if wewnątrz pętli for funkcji isprime sprawdza wartości reszty z dzielenia testowanej liczby, zwracając prawda (1) bądź fałsz (0) w zależności czy liczba jest liczbą pierwszą (reszta z dzielenia =/= 0).

2.1.2 Sekwencyjny – metoda "Sita" *(usuwanie wielokrotności): SEQ_S*

```
26  int main()
27  {
28      long start;
29      int bound = int(sqrt(MAX));
30      bool isNumberPrime[MAX + 1] { false };
31      vector<long> checkingPrimes;
32      checkingPrimes.reserve(bound);
33      for (int i = 2; i <= bound; i++)
34      {
35          if (isprime(i))
36          {
37              checkingPrimes.push_back(i);
38          }
39      }
40      int startingPrimesCount = checkingPrimes.size();
41      for (int i = 0; i < startingPrimesCount; i++)
42      {
43          int multiple = checkingPrimes[i] * 2;
44
45          while (multiple <= MAX)
46          {
47              isNumberPrime[multiple] = true;
48              multiple += checkingPrimes[i];
49          }
50      }
51      int count = 0;
52      vector<long> primeNumbers;
53      primeNumbers.reserve(bound);
54      for (long i = MIN; i <= MAX; i++)
55      {
56          if (!isNumberPrime[i])
57          {
58              primeNumbers.push_back(i);
59              count ++;
60          }
61      }
62      return 0;
63  }
```

Rysunek 3. Kod Funkcji int main() algorytmu sekwencyjnego z wykorzystaniem Sita.

Zaprezentowany algorytm sekwencyjny korzysta z metody Sita Eratostenesa – usuwanie ze zbioru liczb wielokrotności liczb od najmniejsze. Z badanego zbioru testowego, algorytm przesiewa w taki sposób, że pozostają tylko liczby, które nie są wielokrotnością liczb wcześniejszych, tym samym są to liczby pierwsze. Zaprezentowany algorytm na Rysunku 3. przyjmuje górny limit jako pierwiastek kwadratowy ostatniej liczby z badanego przedziału (zmienna bound). Następnie za pomocą przedstawionej na Rysunku 2. funkcji isprime zostaje utworzona tablica liczb pierwszych z badanego zakresu. Po wypełnieniu tablicy sprawdzonych liczb tworzony jest wektor liczb pierwszych w celu porównania obecności kolejnych liczb całkowitych ze wskazanego zakresu w tablicy sprawdzonych.

2.2 Algorytmy Równoległe

2.2.1 Omówienie zastosowanych dyrektyw OMP

Do zrównoleglenia pracy programów zastosowano poznane na laboratoriach dyrektywy OMP, miejsca ich występowania wskazują jawnie bloki kodu które powinny zostać wykonane równolegle.

```
#pragma omp parallel num_threads(num)
```

Dyrektywa ustawia ilość wątków, które wezmą udział w wykonaniu regionów równoległych.

```
#pragma omp for schedule(dynamic)
```

Dyrektywa dla każdego wątku przypisywana jest określonym parametrem liczba iteracji (domyślnie =1). Po wykonaniu obliczeń wątki otrzymują kolejną część iteracji do wykonania.

```
omp_get_thread_num()
```

Funkcja pobiera i zwraca numer id bieżącego wątku.

2.2.2 Omówienie zagadnienia podziału pracy

"schedule" w dyrektywie *#pragma omp for schedule()* służy do definiowania sposobu podziału pracy na wątki w pętli for, następnie drugi parametr dyrektywy służy do określenia liczności każdego podzbioru iteracji. Możliwe rodzaje podziału pracy:

- static - iteracje rozdzielane są na zbiory o rozmiarze podanego parametru. podział dokonany przed uruchomieniem pętli, zatem najmniejszy narzut czasu wykonania. Bez określenia drugiego parametru zbiór iteracji wynosi w przybliżeniu liczba_iteracji / liczba_wątków.

- dynamic - każdemu wątkowi przypisywana jest liczba iteracji określona przez parametr. Po wykonaniu obliczeń wątki otrzymują kolejną porcję iteracji do wykonania. Ze względu na konieczność ubiegania się o przydział pracy przez wątki występują opóźnienia. Ostatni przydzielony podzbiór iteracji może mieć wielkość mniejszą niż ta, która jest określona przez podany parametr.

- guided - iteracje rozdzielane są pomiędzy wątki podobnie jak przy opcji dynamic. Różnica polega na tym, że w tym przypadku rozmiar przypisywanych podzbiorów iteracji zmniejsza się w czasie.

- runtime - podział zależy od wartości zmiennej środowiskowej OMP SCHEDULE.

2.2.3 Omówienie zagadnienia potencjalnych problemów

a) Problemy poprawnościowe

- Wyścig – Termin, którego używa się do oznaczenia sytuacji, kiedy co najmniej dwa wątki wykonują operację na zasobach współdzielonych. Konflikt ten prowadzi do nadpisywania danych w przypadkowej kolejności i w rezultacie uzyskania niedeterministycznych, prawdopodobnie niepoprawnych wyników, ostateczny wynik operacji jest zależny od momentu jej realizacji. Występowanie wyścigu może być konsekwencją braku implementacji mechanizmów zapewniających bezpieczne przydzielenie sekcji krytycznej w dostępie do współdzielonych zasobów. Aby zapobiec warunkom wyścigu należy wykorzystać mechanizm niedopuszczający do sytuacji, gdy więcej niż jeden proces zyskuje dostęp do zasobów dzielonych w tym samym czasie - wzajemne wykluczanie pozwala na zachowanie własności bezpieczeństwa. Dostępne są dyrektywy umożliwiające synchronizację dostępu wątków do sekcji krytycznej oraz możliwy jest odpowiedni podział pracy wątków.

- Żywotność – Własność polegająca na zapewnieniu możliwości spełnienia warunku koniecznego do wykonania każdego procesu w programie, uniknięcie sytuacji zagłodzenia procesu oczekującego na uzyskanie dostępu do zasobów bądź sekcji krytycznej. Problem ten nie występuje w przedstawionych poniżej programach z powodu braku sytuacji, gdzie wątek mógłby nie doczekać się na dostęp do zasobu.

b) Problemy efektywnościowe

- False sharing – Fałszywe współdzielenie danych, polega na wielokrotnym unieważnieniu linii pamięci w pamięci podręcznej procesorów, które stają się nieaktualne w wyniku zapisu przez różne procesy, nawet jeśli są niezależne od siebie. Różne procesy uzyskujące dostęp do różnych elementów w tej samej linii pamięci widzą wiersz oznaczony jako unieważniony. Są zmuszone do pobrania aktualnej kopii danych z pamięci lub z innego miejsca, nawet jeśli element, do którego uzyskano dostęp, nie został zmodyfikowany. Wynika to z faktu, że spójność pamięci podręcznej jest utrzymywana na podstawie linii pamięci podręcznej, a nie dla poszczególnych elementów.

2.2.4 Równoległy - wersja domenowa: *PAR_DOM*

W celu zrównoleglenia algorytmu dzielenia dołączono dodatkową tablicę do przechowywania prywatnych obliczonych liczb pierwszych procesu. W zrównoleglonym regionie każdy proces oblicza własną część liczb pierwszych w badanym zbiorze, a następnie zapisuje swoje wyniki na odpowiednie miejsce we właściwej wynikowej tablicy przechowującej liczby pierwsze. Poniższy kod wykorzystuje wcześniej zaprezentowaną funkcję na Rysunku 2, w celu sprawdzenia czy podana liczba jest liczbą pierwszą.

Podejście domenowe do algorytmu Sita Eratostenesa charakteryzuje się przydziałem do procesu całej tablicy wykreśleń przy przekazaniu mu fragmentu tablicy liczb pierwszych we wskazanym zakresie.

```
28  long* primesToCheck(long *tab, long start, long bound){
29      long indice = 0;
30
31      for (long i = start; i <= bound; i++)
32      {
33          if (isprime(i))
34          {
35              tab[indice] = i;
36              indice++;
37          }
38      }
39      return tab;
40  }
```

Rysunek 4. Kod funkcji primesToCheck algorytmu równoległego z metodą domenową.

```
42  int main()
43  {
44
45      clock_t cstart, cstop;
46      int maxThreadsCount = 4;
47      int threadsCount = omp_get_num_threads();
48      int bound = int(sqrt(MAX));
49      long start;
50      vector<long> checkingPrimes;
51      checkingPrimes.reserve(bound);
52
53
54      for (int i = 2; i <= bound; i++)
55      {
56          if (isprime(i))
57          {
58              checkingPrimes.push_back(i);
59          }
60
61      }
62      int startingPrimesCount = checkingPrimes.size();
63
64     bool** isNumberPrime = new bool* [maxThreadsCount];
65     for (int i = 0; i < maxThreadsCount; i++)
66     {
67         isNumberPrime[i] = new bool[MAX + 1]{ false };
68     }
```

Rysunek 5. Część I kodu funkcji main() algorytmu równoległego z metodą domenową.

```
70      #pragma omp parallel num_threads(maxThreadsCount)
71      {
72      #pragma omp for schedule(dynamic)
73          for (int i = 0; i < startingPrimesCount; i++)
74          {
75              int multiple = checkingPrimes[i] * 2;
76
77              while (multiple <= MAX)
78              {
79                  isNumberPrime[omp_get_thread_num()][multiple] = true;
80                  multiple += checkingPrimes[i];
81              }
82
83          }
84
85      }
86
87      long primeNumbers[MAX+1] { 0 };
88
89
90      for (int i = MIN; i < MAX; i++)
91      {
92          int sum = false;
93          for (int j = 0; j < maxThreadsCount; j++)
94          {
95              sum |= isNumberPrime[j][i];
96          }
97          if (!sum)
98          {
99              primeNumbers[i] = i;
100         }
101
102     }
103
104     delete[] isNumberPrime;
105
106     return 0;
107 }
```

Rysunek 6. Część II kodu funkcji main() algorytmu równoległego z metodą domenową.

Zaprezentowane rozwiązanie wykorzystuje dyrektywę #pragma omp for schedule(dynamic) (Rysunek 6, linia 72), dzięki czemu do każdego wątku przypisywany jest zakres iteracji, domyślnie o wielkości przydziału jeden. Rozwiązanie te usprawnia szybkość działania programu w stosunku do zastosowania dyrektywy #pragma omp for. Następnie, w bloku równoległym za pomocą zastosowania odrębnych tablic zminimalizowano ryzyko wystąpienia false sharing'u, oraz po wykonaniu części równoległej następuje scalenie tablic wątków w jedną (o odpowiedniej kolejności), tym samym uniknięto błędów prowadzących wystąpienia wyścigu (Rysunek6).

2.2.5 Równoległy - wersja funkcyjna: *PAR_FUN*

Wersja funkcyjna algorytmu Sita Eratostenesa polega na przydziale do procesu fragmentu tablicy wykreśleń przy przekazaniu mu całego zbioru liczb pierwszych, których wielokrotności są usuwane. Poniższy kod wykorzystuje wcześniej zaprezentowaną funkcję na Rysunku 2, w celu sprawdzenia czy podana liczba jest liczbą pierwszą.

```
28  int startingPoint(int threadNumber, int maxThreadCount){
29      int startPoint = MIN + (MAX - MIN) / maxThreadCount * threadNumber;
30
31      return startPoint;
32  }
33
34  int endingPoint(int threadNumber, int maxThreadCount){
35      if (threadNumber < maxThreadCount - 1)
36      {
37          int endPoint = startingPoint(threadNumber, maxThreadCount) + (MAX - MIN) / maxThreadCount - 1;
38          return endPoint;
39      }
40      else
41      {
42          return MAX;
43      }
44
45  }
```

Rysunek 7. Kod funkcji startingPoint() oraz endingPoint() algorytmu równoległego z metodą funkcyjną.

```
48  int main()
49  {
50      int maxThreadsCount = 4;
51      int bound = int(sqrt(MAX));
52      long start;
53      vector<long> checkingPrimes;
54      checkingPrimes.reserve(bound);
55
56      for (int i = 2; i <= bound; i++)
57      {
58          if (isprime(i))
59          {
60              checkingPrimes.push_back(i);
61              cout << i<<endl;
62          }
63      }
64      int startingPrimesCount = checkingPrimes.size();
65
66     bool** isNumberPrime = new bool* [maxThreadsCount];
67     for (int i = 0; i < maxThreadsCount; i++)
68     {
69         isNumberPrime[i] = new bool[MAX + 1]{ false };
70     }
```

Rysunek 8. Część I kodu funkcji main() algorytmu równoległego z metodą funkcyjną.

```
72      #pragma omp parallel num_threads(maxThreadsCount)
73      {
74
75          int startPoint = startingPoint(omp_get_thread_num(), maxThreadsCount);
76          int endPoint = endingPoint(omp_get_thread_num(), maxThreadsCount);
77
78          for (int i = 0; i < startingPrimesCount; i++)
79          {
80              int multiple = checkingPrimes[i] * 2;
81              int power = ((startPoint - checkingPrimes[i] * 2) / checkingPrimes[i]) * checkingPrimes[i];
82
83              if (power > 0)
84              {
85                  multiple += power;
86              }
87
88              while (multiple <= endPoint)
89              {
90                  isNumberPrime[omp_get_thread_num()][multiple] = true;
91                  multiple += checkingPrimes[i];
92              }
93          }
94      }
95      long primeNumbers[MAX+1] { 0 };
96
97      for (int i = MIN; i <= MAX; i++)
98      {
99          int sum = false;
100         for (int j = 0; j < maxThreadsCount; j++)
101         {
102             sum |= isNumberPrime[j][i];
103         }
104         if (!sum)
105         {
106             primeNumbers[i] = i;
107         }
108     }
109     delete[] isNumberPrime;
110     return 0;
111 }
```

Rysunek 9. Część II kodu funkcji main() algorytmu równoległego z metodą funkcyjną.

Algorytm tak jak algorytm w wersji domenowej wykorzystuje zaimplementowaną wersje dzielenia równoległego do pobrania początkowej tablicy liczb pierwszych z badanego zbioru ograniczonego do pierwiastka kwadratowego górnej granicy (zmienna MAX). Wątki osobno otrzymują zakres tablicy liczb do sprawdzania, których granice zostają przydzielone za pomocą funkcji getStart() getEnd(), W celu usprawnienia wydajności oraz jakości otrzymywanych rozwiązań ograniczono zakres przeszukiwania dla każdego wątku, tym samym nie sprawdzając od początku całości wyznaczonego fragmentu. Za pomocą zastosowania dodatkowych prywatnych tablic dla każdego wątku w bloku równoległym, następnie scaleniem na jednej tablicy w odpowiedniej kolejności wyników zapisanych na prywatnych tablicach wątków, wyeliminowano problemy z wyścigiem oraz false sharing'iem.

2.3 Tabela podsumowująca

Program	Poprawność	Przyczyna	Wyścig	False Sharing	Przyczyna
SEQ_D	TAK	Praca na jednym wątku	NIE	NIE	Praca na jednym wątku
SEQ_S	TAK	Praca na jednym wątku	NIE	NIE	Praca na jednym wątku
PAR_DOM	TAK	Scalenie wektora wyników po zakończeniu bloku równoległego	NIE	NIE	Każdy wątek operuje na osobnej tablicy, które dopiero po zakończeniu równoległości są scalane: brak operowania na wspólnej linii adresowej
PAR_FUN	TAK	Scalenie wektora wyników po zakończeniu bloku równoległego	NIE	NIE	Wątki operują na wyznaczonym zakresie tablicy wykreśleń, zapewniono nieprzeplatanie się zakresów

Tabela 2. Podsumowanie przedstawionych programów, zestawienie problemów poprawnościowych i efektywnościowych w poszczególnych programach.

3. Prezentacje wyników i omówienie przebiegu eksperymentu obliczeniowo-pomiarowego

3.1. Przedstawienie eksperymentu

Eksperyment został przeprowadzany na urządzeniu za pomocą zaimplementowanych programów. Dla każdego testowego programu test przeprowadzany był dla trzech rodzajów instancji testowych wyznaczających różne zakresy poszukiwania liczb pierwszych. Wymagane instancje testowe:

a) 2...MAX
b) MAX/2...MAX
c) 2....MAX/2

Parametr MAX w programie został przyjęty jako 10^8.

Liczba kolejno użytych procesorów dla każdej z instancji testowej:

a) Przetwarzanie sekwencyjne- Jeden procesor.
b) Przetwarzanie równoległe - Liczba procesorów fizycznych: dwa procesory.
c) Przetwarzanie równoległe- Liczba procesorów logicznych: cztery procesory.

3.2 Omówienie narzędzia Intel® VTune™ Profiler

W celach analizy zaimplementowanych programów skorzystano z narzędzia Intel® VTune™ Profiler, pozwalającego na ocenę jakości kodu pod kątem pracy procesora. Narzędzie wspomaga dokładne przejrzenie czasu pracy konkretnych wątków, VTune pomaga w różnego rodzaju profilowaniu kodu, w tym próbkowaniu stosu, profilowaniu wątków i próbkowaniu zdarzeń sprzętowych. Wynik profiler'a składa się z szczegółowej prezentacji danych, które można prześledzić do poziomu instrukcji. Czas potrzebny na wykonanie instrukcji wskazuje na możliwość istnienia wąskiego gardła w danym miejscu kodu. u. Narzędzie może być również wykorzystane do analizy wydajności wątków. W projekcie korzystano przede wszystkim z danych zebranych przy użyciu Microarchitecture Exploration.

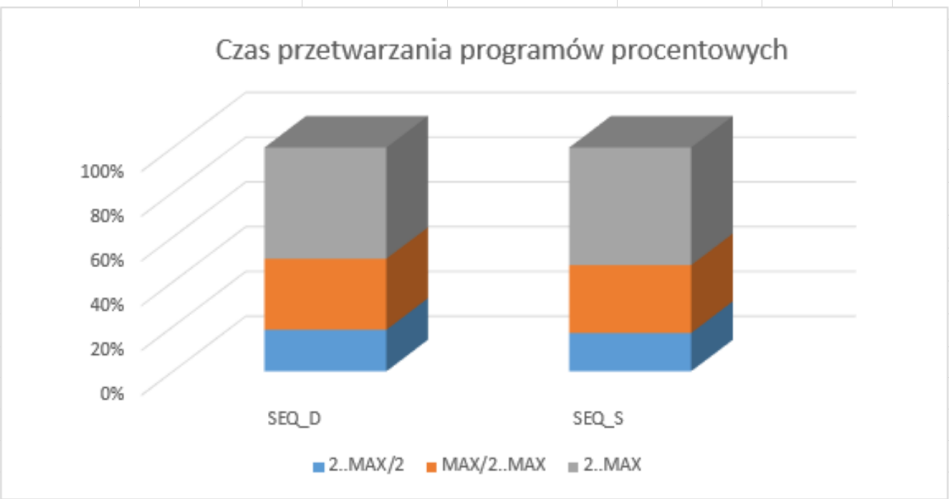
3.3 Wyniki uzyskane dla sekwencyjnych wersji programów

Poniżej przedstawione zostały wyniki uzyskane w Vtune dla wersji sekwencyjnych. Kod przetestowano dla wszystkich doborów zakresów przedziałów poszukiwań liczb pierwszych w celu zapoznania się ze szczegółami faktycznej pracy procesora dla programów nie działających w trybie równoległym o zróżnicowanej złożoności jako baza pod dalszą analizę programów równoległych.

	SEQ_D			SEQ_S		
	1 wątek					
	2..MAX	HALF..MAX	2..HALF	2..MAX	HALF..MAX	2..HALF
Elapsed time[s]	506,389	323,305	189,058	6,142	3,544	2,003
Instructions retired	2,27302E+12	1,43131E+12	8,40984E+11	6,06E+09	4,87E+09	2,97E+09
Clockticks	1896619790	1,19449E+12	7,05438E+12	1,51E+10	1,33E+10	7,46E+09
Retiring	43,80%	44,40%	44,30%	11%	9%	10%
Front-end bound	41,70%	42,10%	42,10%	4,80%	6%	6%
Back-end bound	14%	13%	13%	82,20%	82%	83%
Memory bound	0%	0%	0%	34%	37%	32%
Core bound	14%	13%	13%	48,10%	45%	51%
Effective physical core utilization	23,50%	37,80%	23,20%	22,50%	23,50%	20,50%
Speed [n/s]	592429,9264	463958,1819	793407,3036	48844024,42	42325056,43	74887667,5
instructions [retired / clockticks]	1,20E+03	1,20E+00	1,19E-01	4,02E-01	3,66E-01	3,97E-01

Tabela 3. Wyniki oceny programów sekwencyjnych, SEQ_D, SEQ_S za pomocą narzędzia VTune.

Na wykresie poniżej przedstawiono zestawienie procentowego czasu obliczeń algorytmów sekwencyjnych dla każdej instancji. Wykres pokazuje, że algorytm SEQ_S, choć wielokrotnie szybszy, procentowo dłużej przetwarzał największą instancję 2...MAX niż algorytm SEQ_D. Może to być spowodowane wykorzystaniem przez SEQ_S wyszukiwaniem liczb pierwszych dla fragmentu zakresu przy użyciu SEQ_D, aby uzyskać wstępną tablicę liczb pierwszych: dla większych zakresów liczb wydajność algorytmu dzielenia mocno się obniża.



Wykres 1. Procentowe zestawienie przetwarzania konkretnych instancji przy użyciu algorytmów sekwencyjnych

3.4 Wyniki uzyskane dla równoległych wersji programów

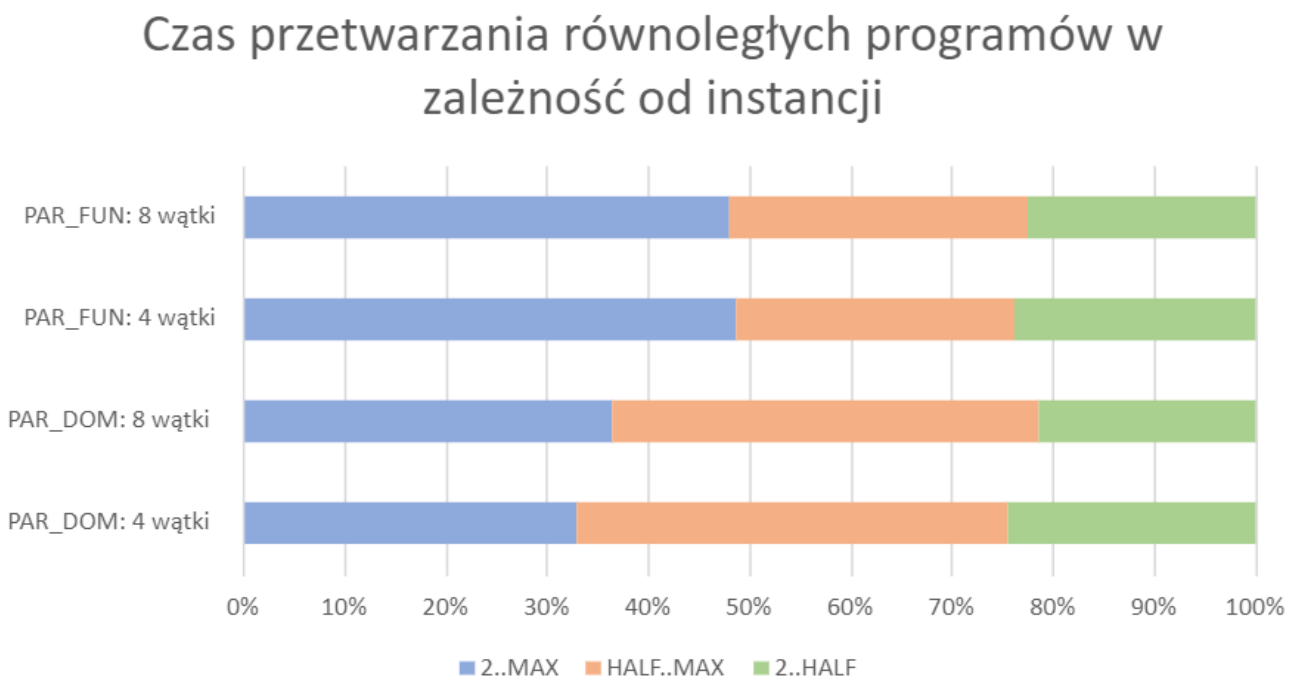
Poniżej przedstawione zostały wyniki uzyskane w Vtune dla wersji równoległej. Kod przetestowano dla wszystkich instancji testowych poszukiwań liczb pierwszych w celu zapoznania się ze szczegółami faktycznej pracy procesora dla programów równoległych,

	PAR_DOM				
	4 Wątki			8 Wątki	
	2..MAX	HALF..MAX	2..HALF	HALF..MAX	2..HALF
Elapsed time[s]	2,638	3,406	1,947	3,821	1,918
Instructions retired	5,21E+09	5,32E+09	2,61E+09	6,74E+09	3,29E+09
Clockticks	3,18E+10	4,18E+10	2,32E+10	7,67E+10	3,53E+10
Retiring	10,00%	5,60%	4,90%	5,60%	5,10%
Front-end bound	3,70%	3,40%	5,10%	8,80%	8,30%
Back-end bound	86,50%	90,80%	89,30%	84,80%	86,30%
Memory bound	34,50%	36,60%	37,90%	37,70%	35,90%
Core bound	51,80%	54,20%	51,50%	47,00%	50,40%
Effective physical core utilization	68,90%	72,10%	50,70%	70,80%	65,40%
Speed [n/s]	113722516,3	44039929,54	77041601,44	39256739,07	78206464,03
instructions [retired / clockticks]	1,64E-01	1,27E-01	1,12E-01	8,79E-02	9,31E-02

Tabela 4. Wyniki oceny programu równoległego PAR_DOM działającego na 4 procesorach fizycznych oraz 8 procesorach logicznych za pomocą narzędzia VTune.

	PAR_FUN				
	4 Wątki			8 Wątki	
	2..MAX	HALF..MAX	2..HALF	HALF..MAX	2..HALF
Elapsed time[s]	3,292	1,864	1,597	2,312	1,76
Instructions retired	5,22E+09	3,17E+09	2,59E+09	4,30E+09	3,25E+09
Clockticks	4,02E+10	2,08E+10	1,94E+10	3,61E+10	3,00E+10
Retiring	6,50%	7,10%	4,20%	8,60%	14,00%
Front-end bound	4,00%	4,60%	4,50%	10%	9,80%
Back-end bound	88%	87,20%	90,70%	78,90%	74,20%
Memory bound	36,50%	36,30%	39,20%	35,50%	27,60%
Core bound	51,80%	50,90%	51,50%	43,40%	46,60%
Effective physical core utilization	62,90%	61,60%	70%	55,00%	65,90%
Speed [n/s]	91130011,54	80472103	93926110,21	64878892,73	85227271,59
instructions [retired / clockticks]	1,30E-01	1,53E-01	1,34E-01	1,19E-01	1,08E-01

Tabela 4. Wyniki oceny programu równoległego PAR_FUN działającego na 4 procesorach fizycznych oraz 8 procesorach logicznych za pomocą narzędzia VTune.



Wykres 2. Procentowe zestawienie przetwarzania konkretnych instancji przy użyciu algorytmów równoległych działających na 4 bądź 8 procesorach

Kolejny wykres przedstawia zestawienie procentowego czasu obliczeń algorytmów równoległych dla kolejnych wielkości instancji. Niemal wszystkie algorytmy najdłużej, bo około 50% czasu przetwarzały największą instancję testową 2...MAX. Wyraźnie wyróżnia się tu algorytm PAR_DOM (równoległe wykreślanie w wersji domenowej) przy użyciu 4 wątków: wynik około 30%.

3.5 Analiza i omówienie zebranych wyników

Miary prędkości oraz efektywność przetwarzania równoległego w ramach czytelnej reprezentacji danych zostały umieszczone w osobnej tabeli z zastosowaniem formatowania warunkowego: Miary przyspieszenia i efektywności algorytmów równoległych z podziałem na wielkość instancji.

- *Acceleration*- przyspieszenie przetwarzania równoległego dla badanego wariantu kodu równoległego, parametr będący ilorazem czasu przetwarzania najlepszego dostępnego przetwarzania (dowolnym algorytmem w tym samym systemie) oraz czasu przetwarzania równoległego danego programu, dla którego przyspieszenie jest wyznaczane.

- *Efficiency - Efektywność przetwarzania równoległego*, iloraz przyspieszenia przetwarzania równoległego i liczby użytych w przetwarzaniu procesorów fizycznych.

		2..MAX			HALF..MAX			2..HALF	
Program	2..MAX	Acceleration	Efficiency	HALF..MAX	Acceleration	Efficiency	2..HALF	Acceleration	Efficiency
PAR_DOM wątki: 4	2,638	1	0,25	3,406	0,547269524	0,136817381	1,947	0,820236261	0,205059065
PAR_DOM wątki: 8	3,298	0,799878714	0,099984839	3,821	0,487830411	0,060978801	1,918	0,832638165	0,104079771
PAR_FUN wątki: 4	3,292	0,801336574	0,200334143	1,864	1	0,25	1,597	1	0,25
PAR_FUN wątki: 8	3,779	0,698068272	0,087258534	2,312	0,806228374	0,100778547	1,76	0,907386364	0,113423295

Tabela 5. Miary przyspieszenia i efektywności algorytmów równoległych z podziałem na wielkość instancji.

Wartość przyspieszenia i efektywności oznaczają jednobarwne paski - im większe pole powierzchni paska, tym lepszy wynik uzyskany przez algorytm dla danej miary. Z zestawienia odczytać można, że zdecydowanie najlepsze wyniki czasowe osiągnęły dwa algorytmy: PAR_FUN (równoległe wykreślane funkcyjne) dla 4 wątków oraz algorytm PAR_DOM (równoległe wykreślanie domenowe) dla ilości wątków 4.

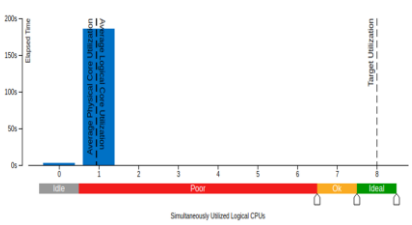
Istotnym elementem analizy jest przyjrzenie się udziałowi wykorzystanych zasobów procesora do przetwarzania kodu. Z wykresu poniżej, zaobserwować można wykorzystanie zasobów zaledwie 5%-10% dla większości badanych algorytmów równoległych. Małym wykorzystaniem zasobów procesora charakteryzują się te algorytmy, które w poprzednim zestawieniu (przyspieszenie i efektywność) uzyskały dobre rezultaty: PAR_ FUN z wykorzystniaem 4 wątków dla instancji testowej MAX/2... MAX oraz PAR_DOM z wykorzystniem 4 wątków dla instancji testowych 2...MAX oraz 2... MAX/2. Zwraca to uwagę na związek między niewielkim wykorzystaniem zasobów procesora a zwiększeniem prędkości przetwarzania. Największym wykorzystaniem zasobów procesora charakteryzuje się program sekwencyjny z dzieleniem, ze względu na sekwencyjny charakter przechodzenia i dzielenia przez każdą pojedynczą liczbę w liczności od 100 milionów do 300 milionów.

Wykres 3. Wykres przedstawiający udział procentowy wykorzystanych zasobów procesora do przetwarzania kodu (Retiring).

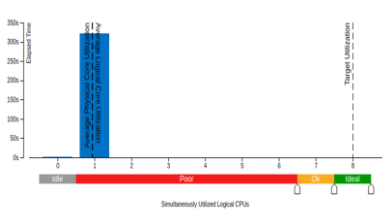
Z powyższych wyników zdecydowano wybrać wyróżnione algorytmy dla analizy udziału procesorów:

- SEQ_D – sekwencyjny algorytm z metodą dzielenia
- PAR_FUN 4 - równoległy algorytm - równoległy algorytm wykreślania funkcyjnie, 4 wątki.
- PAR_DOM 4 - równoległy algorytm wykreślania domenowo, 4 wątki

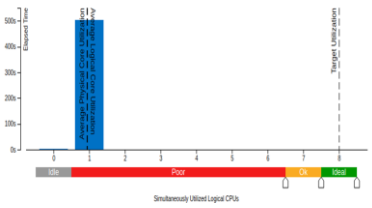
SEQ_D



2... HALF

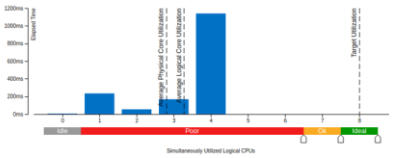


HALF...MAX

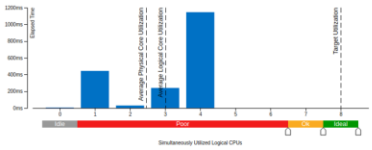


2... MAX

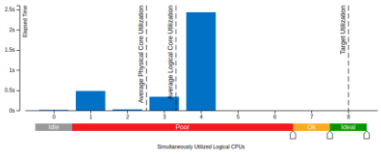
PAR_FUN 4



2... HALF

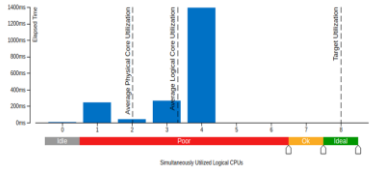


HALF...MAX

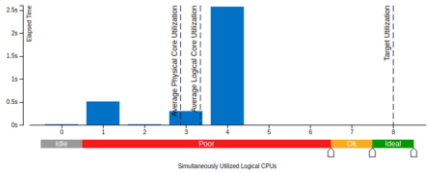


2... MAX

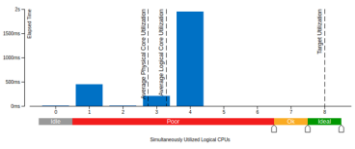
PAR_DOM 4



2... HALF



HALF...MAX



2... MAX

4. Wnioski

Nie wszystkie użyte podejścia okazały się efektywnie wykorzystywać struktury wewnętrzne procesora. Najlepszym algorytmem pod tym względem okazała się wersja równoległego algorytmu wykreślania funkcyjnego, w której praca rozłożona była na 4 procesory fizyczne, a osiągnięte wyniki czasowe okazały się najlepsze ze wszystkich wersji równoległych. Wpływ na to może mieć operowanie na odrębnych fragmentach tablicy wykreśleń przez wątki, gdyż zarządzanie równoległością od strony dyrektyw OMP zarówno tu, jak i w wersji domenowej przetestowano podobnie i w wersji funkcyjnej osiągnięto najlepsze efekty, natomiast w wersji domenowej próby ulepszenia kodu innymi dyrektywami przynosiły nieznaczne tylko w skali wszystkich wyników rezultaty.

Wśród ograniczeń efektywnościowych w przypadku algorytmów sekwencyjnych dominują front-end bound - ograniczenie wejścia. Sytuacja przedstawia się odwrotnie w przypadku pozostałych algorytmów równoległych, gdzie zdecydowaną większość ograniczeń stanowią ograniczenia back-end - ograniczenie wyjścia bound oraz memory bound - ograniczenie systemu pamięci (nie korzystanie z pamięci podręcznej) i core bound - ograniczenie jednostek wykonawczych (oznacza to, że duża liczba jednostek wykonawczych procesora zostaje niewykorzystana).