Wydział Informatyki i Telekomunikacji

Przetwarzanie równoległe

Laboratorium
Termin:*X*

Znajdowanie liczb pierwszych

Projekt 1 - *wersja 1*

Wymagany termin oddania:X

Rzeczywisty termin oddania:X

Autorzy:
X

Prowadzący zajęcia:
dr inż. Rafał Walkowiak

23 czerwca 2023

Spis treści

1 Wstęp

Projekt polegał na przeprowadzenie eksperymentów, sprawdzających przetwarzanie równoległe różnych wersji kodu. Kody polegają na przeszukiwaniu liczb pierwszych. Przygotowane zostały dwa podejścia koncepcyjne, polegające na dzieleniu bananej liczby przez liczby pierwsze oraz usuwaniu ze zbioru liczb bedących liczbami złożonymi. Dla drugiego podejścia zostały również przygotowane dwa warianty zrównoleglenia, funkcyjne i domenowe. Wszystkie kody były uruchamiane sekwencyjnie i równolegle. Dla uruchomień równoległych ustawiane były liczby procesorów równe liczbie procesorów logicznych w systemie oraz liczbie procesorów fizycznych. Wszystkie kody były uruchamiane na trzech instancjach testowych, 2...MAX, 2...$\frac{MAX}{2}$ oraz $\frac{MAX}{2}$...MAX, gdzie MAX przyjęliśmy jako 250 000 000.

2 Opis wykorzystanego systemu obliczeniowego

Procesor:

- Intel Core i7-6500U 2.5GHz
- **Liczba procesorów fizycznych:** 2
- **Liczba procesorów logicznych:** 4
- **Pamięć podręczna procesora:** 4MiB

Oprogramowanie:

- **System operacyjny:** Manjaro Linux x86_64 5.13.19-2-MANJARO
- **Oprogramowanie do przeprowadzenia testów:** Intel VTUNE 2022.1.0-98-1
- **Oprogramowanie do przygotowania kodów:** Visual Studio Code 1.65.2-1

3 Prezentacja przygotowanych wariantów kodów

Tablice wykreśleń przechowywaliśmy w tablicy typu boolean **tab**, natomiast liczby pierwsze potrzebne do podejścia z sitem w tablicy typu integer **PRIMES**. Stałe **MAX** i **MIN**, były zgodne z założeniami instancji testowych, dlatego też wykonaliśmy trzy kompilacje dla każdej z instancji.

```
bool tab[MAX - MIN + 1];
int PRIMES[(int)std::sqrt(MAX)];
```

Listing 1: Przechowywanie danych

Aby, ułatwić proces testowania kodów, odczytywane są dwa parametry do programu, liczba wątków oraz numer funkcji. Tym sposobem można było uruchamiać konkretną wersję kodu bez konieczności kompilacji dla każdego wariantu.

```
...
string arg = argv[1];
try {
    size_t pos;
    threads = stoi(arg, &pos);
    if (pos < arg.size()) {
        cerr << "Trailing characters after number: " << arg << '\n';
    }
} catch (invalid_argument const &ex) {
    cerr << "Invalid number: " << arg << '\n';
} catch (out_of_range const &ex) {
    cerr << "Number out of range: " << arg << '\n';
}
cout << "Threads number: " << threads << "\n";
...
```

Listing 2: Przykład odczytu parametru określającego liczbę wątków

Również w celu uproszczenia testowania poprawności kodów wykonaliśmy dwie funkcje, **printPrimes** oraz **countPrimes**. Funkcja **printPrimes** wypisywała wszystkie liczby pierwsze w badanym zbiorze w rzędach po 10 liczb, natomiast funkcja **countPrimes** zliczała ile liczb pierwszych zostało znalezionych.

```
void printPrimes(int m, int n) {
    int count = 0;
    for (int i = 0; i < n - m + 1; i++)
    {
        if (tab[i] == true)
        {
            std::cout << m + i << ", ";
            count++;
        }

        if (count == 10){
            std::cout << std::endl;
            count = 0;
        }
    }
```

```
16      std::cout << std::endl;
17  }
18
19  void countPrimes(int m, int n) {
20      int primes_num = 0;
21      for (int i = 0; i < n - m + 1; i++)
22      {
23          if (tab[i] == true)
24          {
25              primes_num++;
26          }
27      }
28      std::cout << std::endl;
29      std::cout << "From " << m << " to " << n << std::endl;
30      std::cout << "Found " << primes_num << " primes" << std::endl;
31  }
```

Listing 3: Funkcje pomocnicze

Do przygotowania tablicy wykreśleń oraz sita z liczbami pierwszymi użyliśmy funkcji **prepareTab** oraz **prepareSieve**.

```
1  void prepareTab(int m, int n, bool tab[]) {
2      for (int i = 0; i < n - m + 1; i++)
3      {
4          tab[i] = true;
5      }
6  }
```

Listing 4: Przygotowanie tablicy wykreśleń

```
1  void prepareSieve(int n, int PRIMES[], int *primesCount){
2      int sqr_variable = (int)pow(n, 1.0 / 2);
3
4      int i = 0;
5      for (int num = 2; num <= sqr_variable; num++)
6    {
7      bool is_prime = true;
8      for (int i = 2; i <= pow(num, 1.0 / 2); i++)
9      {
10       if (num%i == 0)
11       {
12         is_prime = false;
13         break;
14       }
15     }
16     if (is_prime)
17     {
18       PRIMES[i++] = num;
19             (*primesCount)++;
20     }
21   }
22 }
```

Listing 5: Przygotowanie tablicy z sitem

Pierwszym zestawem funkcji, które testowaliśmy były funkcje sprawdzające podzielność każdej liczby. Podstawową wersją była funkcja sekwencyjna **DivSeq**, która sprawdzała po kolei każdą liczbę z tablicy wykreśleń, czy jest podzielna przez kolejne wielokrotności liczb pierwszych. Jak można się domyślić nie jest to optymalne rozwiązanie dla tego problemu, ponieważ będziemy musieli wykonywać operację modulo dla bardzo dużych liczb.

```
1  void DivSeq(int m, int n, bool tab[]) {
2      for (int i = m; i <= n; i++){
3          for (int j = 2; j * j <= i; j++){
4              if (i % j == 0){
5                  tab[i - m] = false;
6                  break;
7              }
8          }
9      }
10 }
```

Listing 6: Funkcja sekwencyjna metody z dzieleniem

Do zrównoleglenia tego podejścia dodaliśmy przed pierwszą pętlą **for** dyrektywy **omp parallel** oraz **omp for schedule** (funkcja DivStatic). Do przydziały zadań procesów zdecydowaliśmy się na podział statyczny cykliczny, ponieważ zagwarantuje on zrównoważone rozłożenie liczb do sprawdzania. Ponadto w testach sprawdził się on szybciej od podziału dynamicznego, z uwagi na równomierne rozłożenie pracy. Wykorzystaliśmy również klauzulę **nowait**, aby usunąć nieotrzebne synchornizacje.

```
1 #pragma omp parallel
2 #pragma omp for schedule(static, 1) nowait
```

Listing 7: Zrównoleglenie funkji z dzieleniem

Drugim zestawem funkcji, które należało przetestować były funkcje wykreślające kolejne wielokrotności liczb pierwszych z zakresu $2...\sqrt{MAX}$. W tym rozwiązaniu na samym początku podzieliliśmy zbiór na mniejsze podzbiory o wielkości 1000000 (**chunk_size**). W każdym podzbiorze, wykreślane są kolejne wielokrotności liczb pierwszych. Funkcja sekwencyjna **SieveSeq**.

```
1  void SieveSeq(int m, int n, bool tab[], int PRIMES[], int primesCount, int
       chunk_size){
2    for (int chunk_num = 0; chunk_num <= (n-m)/chunk_size ; chunk_num++){
3      int my_start = m + chunk_num * chunk_size;
4      int my_end = my_start+chunk_size;
5      if (my_end > n)my_end = n;
6      for (int i = 0; i < primesCount; i++){
7              int z_lewe=std::max((double)2,ceil((double)my_start/(double)PRIMES[i]));
8              int z_prawe=floor((double)my_end/(double)PRIMES[i]);
9              for (int j = z_lewe; j <= z_prawe; j++)
10             {tab[j*PRIMES[i] - m] = false;}
11     }
12   }
13 }
```

Listing 8: Funkcja sekwencyjna metody z sitem

W celu zrównoleglenia podejścia domenowego (**SieveDom**), ponownie przed pierwszą pętlą dodaliśmy klauzule **omp parallel** oraz **omp for schedule**. Ponownie zastosowailśmy podział statyczny cykliczny, aby zagwarantować równe rozłożenie podziału pracy procesorów. Podczas testów jednak nie byliśmy w stanie uzyskać pełnego rozłożenia pracy dla wątków, dopiero w momencie kiedy usunęliśmy klauzulę **nowait** wynik ten zaczął się poprawiać, jednak przez synchronizację czas znacznie się pogorszył, dlatego pozostawiliśmy kod z klauzulą **nowait**.

```
#pragma omp parallel
#pragma omp for schedule(static, 1) nowait
```

Listing 9: Zrównoleglenie funkcji z podejściem domenowym

Drugą funkcją wymagającą zrównoleglenia była funkcja z podejściem funkcyjnym (**SieveFun**). W tym przypadku zmian było więcej, ponieważ należało podzielić przydział liczb pierwszych, a nie podzbiorów. Dlatego też, każdy z procesorów pracował na całym zbiorze, ale dla swojego lokalnego zbioru liczb pierwszych. Do podziału ponownie wykorzystaliśmy podział statyczny cykliczny, aby zrównoważyć podział pracy.

```
#pragma omp parallel
{
int LocalPRIMES[primesCount];
int countLocalPrimes = 0;
#pragma omp for schedule(static, 1) nowait
for (int i = 0; i < primesCount; i++)
{
    LocalPRIMES[countLocalPrimes++] = PRIMES[i];
}
...
}
```

Listing 10: Zrównoleglenie funkcji z podejściem funkcyjnym

4 Prezentacja wyników i omówienie przebiegu eksperymentu obliczeniowo-pomiarowego

4.1 Przygotowanie testów

Do przetestowania zostały użyte funkcje **DivSeq**, **DivStatic**, **SieveSeq**, **SieveFun** oraz **SieveDom**. Wszystkie funkcje były testowane na trzech zbiorach. Funkcje równoległe zostały uruchomione na dwóch i czterech wątkach. Do zebrania danych, które umieściliśmy w tabelce posłużyliśmy się trybem **Microarchitecture Exploration**, który pozwala na wykrycie ograniczeń sprzętu w aplikacji lub systemie przetwarzającym. Łączna ilość testów potrzebna do zebrania danych wyniosła 24 testy.

4.2 Sposób zbierania przez oprogramowanie Intel Vtune informacji o efektywności przetwarzania

Dane są zbierane przez program za pomocą PMU (performance monitoring units). PMU zawierają liczniki sprzętowe, które można zaprogramować do zliczania wystąpień różnych zdarzeń procesora.
Program zbiera informacje o zdarzeniach za pomocą podejścia Event Base Sampling (EBS) – testowania przebiegu przetwarzania w oparciu o zdarzenia procesora.

MIN=2, MAX=250000000	sekwencyjnie					
	DivSeq 1 wątek			SieveSeq 1 wątek		
	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
Czas przetwarzania [s]	473,847	176,39	298,209	1,009	0,532	0,536
Instructions retired	1,73E+12	6,42E+11	1,09E+12	4,71E+09	2,34E+09	2,37E+09
Clockticks	1,40E+12	5,22E+11	8,84E+11	2,88E+09	1,50E+09	1,51E+09
Retiring	55,5%	55,8%	55,7%	50,5%	39,0%	26,7%
Front-end bound	43,5%	43,6%	43,6%	4,3%	13,4%	12,2%
Back-end bound	0,8%	0,3%	0,4%	44,7%	45,5%	57,7%
Memory bound	0,0%	0,0%	0,0%	21,5%	20,9%	28,2%
Core bound	0,8%	0,3%	0,4%	23,1%	24,6%	29,5%
Effective physical core utilization	47,7%	47,4%	47,6%	47,5%	59,1%	59,4%
Przyspieszenie	x	x	x	x	x	x
Prędkość przetwarzania	5,28E+05	7,09E+05	4,19E+05	2,48E+08	2,35E+08	2,33E+08
Efektywność	x	x	x	x	x	x

Rysunek 1: Wynik przetwarzania sekwencyjnego

MIN=2, MAX=250000000	równolegle DivStatic					
	2 wątki			4 wątki		
	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
Czas przetwarzania [s]	238,203	88,307	149,798	181,564	67,17	113,477
Instructions retired	1,73E+12	6,42E+11	1,09E+12	1,74E+12	6,42E+11	1,09E+12
Clockticks	1,41E+12	5,23E+11	8,87E+11	2,10E+12	7,77E+11	1,32E+12
Retiring	56,0%	55,9%	56,1%	73,9%	74,1%	74,0%
Front-end bound	43,7%	43,7%	43,9%	26,0%	26,4%	26,1%
Back-end bound	0,1%	0,2%	0,0%	0,0%	0,0%	0,0%
Memory bound	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%
Core bound	0,1%	0,2%	0,0%	0,0%	0,0%	0,0%
Effective physical core utilization	97,1%	97,0%	97,1%	96,8%	97,0%	97,6%
Przyspieszenie	1,99	2,00	1,99	2,61	2,63	2,63
Prędkość przetwarzania	1,05E+06	1,42E+06	8,34E+05	1,38E+06	1,86E+06	1,10E+06
Efektywność	0,99	1,00	1,00	0,65	0,66	0,66

Rysunek 2: Wynik przetwarzania równoległego dla dzielenia

MIN=2, MAX=250000000	równolegle SieveFun					
	2 wątki			4 wątki		
	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
Czas przetwarzania [s]	0,783	0,413	0,427	0,76	0,566	0,504
Instructions retired	4,73E+09	2,34E+09	2,38E+09	4,73E+09	2,36E+09	2,40E+09
Clockticks	3,63E+09	1,90E+09	1,95E+09	5,84E+09	4,29E+09	3,40E+09
Retiring	27,2%	29,8%	27,4%	36,3%	20,5%	34,7%
Front-end bound	5,5%	7,3%	9,1%	15,5%	16,4%	24,4%
Back-end bound	64,2%	60,3%	62,3%	51,5%	57,9%	36,9%
Memory bound	32,8%	27,6%	28,4%	22,7%	27,1%	17,7%
Core bound	31,4%	32,7%	33,9%	28,8%	30,8%	19,2%
Effective physical core utilization	77,3%	89,5%	95,5%	85,1%	61,5%	53,4%
Przyspieszenie	1,29	1,29	1,26	1,33	0,94	1,06
Prędkość przetwarzania	3,19E+08	3,03E+08	2,93E+08	3,29E+08	2,21E+08	2,48E+08
Efektywność	0,64	0,64	0,63	0,33	0,23	0,27

Rysunek 3: Wynik przetwarzania równoległego dla sita funkcyjnego

MIN=2, MAX=250000000	równolegle SieveDom					
	2 wątki			4 wątki		
	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
Czas przetwarzania [s]	0,687	0,367	0,388	0,796	0,412	0,455
Instructions retired	4,73E+09	2,34E+09	2,38E+09	4,74E+09	2,35E+09	2,39E+09
Clockticks	3,09E+09	1,61E+09	1,71E+09	6,09E+09	3,05E+09	3,38E+09
Retiring	46,4%	50,1%	49,0%	32,6%	45,5%	34,4%
Front-end bound	6,5%	3,1%	12,2%	19,4%	28,7%	17,2%
Back-end bound	43,4%	47,6%	36,9%	44,1%	21,3%	47,2%
Memory bound	20,4%	9,0%	16,3%	19,0%	8,5%	21,9%
Core bound	23,0%	38,6%	20,6%	25,0%	12,7%	25,3%
Effective physical core utilization	75,1%	71,9%	79,6%	62,4%	57,0%	58,3%
Przyspieszenie	1,47	1,45	1,38	1,27	1,29	1,18
Prędkość przetwarzania	3,64E+08	3,41E+08	3,22E+08	3,14E+08	3,03E+08	2,75E+08
Efektywność	0,73	0,72	0,69	0,32	0,32	0,29

Rysunek 4: Wynik przetwarzania równoległego dla sita domenowego

5 Wnioski

Jak można zauważyć na wynikach, w przypadku przetwarzania sekwencyjnego zdecydowanie przeważa metoda z sitem Erastotenesa, która jest średnio ponad 400 razy szybsza od metody z dzieleniem modulo. Warto również zauważyć, że dla zbiorów testowych 2...$\frac{MAX}{2}$ oraz $\frac{MAX}{2}$...MAX dla metody z dzieleniem czasy znacznie się różnią, co wynika z tego, że w drugim zbiorze mamy większe liczby, a co za tym idzie dużo więcej operacji dzielenia. Natomiast dla metody z sitem czasy są przybliżone ponieważ złożoność obliczeniowa w obu zbiorach była porównywalna.

W przypadku przetwarzania równoległego dla dzielenia, otrzymujemy dwukrotne przyspieszenie dla dwóch wątków oraz przyspieszenie 2.61 przy użyciu 4 wątków. Przy użyciu dwóch wątków efektywność wynosi 1, natomiast dla 4 wątków 0.66. W każdym z przypadków efektywne wykorzystanie rdzeni fizycznych procesora wynosi około 97%. Możemy więc wywnioskować, że użyty podział statyczny cykliczny zrównoważył dobrze podział pracy procesorów. Wąskim gardłem okazał się tutaj fron-end bound, czyli w ograniczenie efektywności przetwarzania części wejściowej procesora.

Dla metod wykorzystujących sito zarówno funkcyjnie jak i domenowo przetwarzanie równoległe nie okazało się tak efektywne jak przy dzieleniu. Dla rozwiązania funkcyjnego przyspieszenie wyniosło 1.29 dla 2 wątków oraz 1.3 dla zbioru 2...MAX i 1 dla zbiorów 2...$\frac{MAX}{2}$ i $\frac{MAX}{2}$ przy użyciu 4 wątków. Zdecydowanie sam podział statyczny cykliczny liczb pierwszych dla poszczególnych procesorów nie wystarczył, mimo wysokiej efektywności wykorzystania rdzeni fizycznych procesora. Wąskim gardłem był tu zdecydowanie back-end bound, memory bound oraz core bound. Back-end bound odpowiedzialny jest za ogranczenia części wyjściowej procesora, czyli przypadku gdy część czołowa dostarcza mikrooperację, lecz nie może jej przekazać części wyjściowej, która nie jest gotowa do obsługi. Kategoria Memory Bound dotyczy podsystemu pamięci. Braki trafień do pamięci podręcznej i dostępy do pamięć DRAM mogą powodować utknięcia związane z pamięcią i ograniczenia tego typu. Utknięcia z kategorii Core Bound są spowodowane mniej niż optymalnym wykorzystaniem dostępnych jednostek wykonawczych w CPU podczas każdego cyklu. Rozwiązanie domenowego osiągnęło zdecydowanie lepsze czasy, jednak nadal nie uzyskało lepszej efektywności. Podczas użycia 4 wątków również zaobserwowaliśmy spadek przyspieszenia. W tym przypadku jednak efektywność dla 4 wątków była bardziej wyrównana niż w rozwiązaniu funkcyjnym. Tutaj również wąskim gardłem okazały się również back-end bound, memory bound oraz core bound, jednak memory bound był zdecydowanie niższy, w szczególności dla zbioru testowego 2...$\frac{MAX}{2}$.

Podsumowując najlepszym rozwiązaniem okazało się przetwarzanie równoległe domenowe dla dwóch wątków, które osiągnęło prędkość przetwarzania równą 3,64E+08 dla zbioru 2...$\frac{MAX}{2}$, natomiast najgorszym rozwiązaniem okazało się przetwarzanie statyczne cykliczne z dzieleniem dla instancji $\frac{MAX}{2}$...MAX. Jest to w dużym stopniu zależne od rozmia©ów instancji, z uwagi na złożoność metody z dzieleniem. Najefektywniejsze okazało się podejście zrównoleglania w metodzie z dzieleniem,

natomiast dla metody z sitem zdecydowanie lepsze jest podejście domenowe. Warto jednak dodać, że zarówno podejście domenowe jak i funkcyjne ma dużo wąskich gardeł i dla obu podejść wraz ze wzrostem wątków przyspieszenie maleje. Może to po części wynikać z dużej ilości obliczeń zmienno przecinkowych, wykonywanych podczas liczenia przedziałów, dla danej iteracji. W metodzie funkcyjnej wynik ten może być gorszy z uwagi na występowanie *false-sharingu* na tablicy wykreśleń.
Zjawisko to polega na wielokrotnym, cyklicznym (wywołanym przez wielokrotne zapisy w różnych procesorach wartości do tej samej linii pamięci podręcznej procesora):

- unieważnianiu powielonych linii pamięci w pp procesorów, które przestają być aktualne w wyniku zapisu danych do jednej z wielu kopii tej linii
- konieczności sprowadzenia do pamięci podręcznej procesora realizującego kod wątku (korzystającego z danych sąsiednich) aktualnej kopii danych z unieważnionej linii.

MIN=2, MAX=250000000	DivSeq	DivStatic	SieveSeq	SieveFun	SieveDom
Prędkość przetwarzania	7,09E+05	1,86E+06	2,48E+08	3,29E+08	3,64E+08
Wielkość instancji	2...MAX/2	2...MAX/2	2...MAX	2...MAX	2...MAX
Liczba wątków	1	4	1	4	2

Rysunek 5: Podsumowanie wyników prędkości przetwarzania

Spis rysunków

Listings