LABORATORIUM Z PRZETWARZANIA RÓWNOLEGŁEGO			
Autorzy:	Alan Kowańdy Julia Banach	Adres email:	alan.kowandy@student.put.poznan.pl julia.banach@student.put.poznan.pl
Termin oddania sprawozdania:	10.12.2024	Wersja sprawozdania:	Wersja pierwsza
Termin wymagany:	30.12.2023	Indeksy:	Alan Kowańdy – 147259 Julia Banach - 154015

1. System obliczeniowy

Procesor:	AMD Ryzen 5 3600
Liczba procesorów fizycznych:	6
Liczba procesorów logicznych:	12
Liczba uruchamianych w systemie wątków:	12
Oznaczenie typu procesora:	100-000000031
Wielkość i organizacja pamięci podręcznych procesora:	L1: 384KB L2: 3MB L3: 32MB
Wersja systemu operacyjnego:	Ubuntu x86 64-bit 23.10
Oprogramowanie:	Visual Studio Code

2. Prezentacja przygotowanych wariantów kodów z wyjaśnieniem przewidywanego przebiegu przetwarzania

Zacznijmy od przedstawienia definicji wyścigu oraz false sharingu:

Wyścig danych występuje, gdy dwa wątki mają dostęp do tej samej pamięci bez odpowiedniej synchronizacji. Może to skutkować tym, że program generuje nieterministyczne wyniki w trybie równoległym.

Symultaniczne aktualizacje pojedynczych elementów w tej samej linii pamięci podręcznej, pochodzące z różnych procesorów, powodują unieważnienie całych linii podręcznej, nawet jeśli te aktualizacje są logicznie niezależne od siebie. Każda aktualizacja pojedynczego elementu linii pamięci podręcznej oznacza całą linię jako nieaktualną. Inne procesory, które mają dostęp do innego elementu w tej samej linii, widzą ją oznaczoną jako nieaktualną. Są zmuszone do pobrania bardziej aktualnej kopii linii z pamięci lub innego źródła, chociaż dostępny element nie został zmodyfikowany. Dzieje się tak, ponieważ spójność pamięci podręcznej jest utrzymywana na poziomie linii pamięci podręcznej, a nie dla poszczególnych elementów. W rezultacie występuje wzrost ruchu międzyukładowego i nadmiarów. Ponadto, podczas trwania aktualizacji linii pamięci podręcznej, dostęp do elementów w linii jest zahamowany.

Ta sytuacja nazywana jest fałszywym współdzieleniem. Jeśli zdarza się to często, wydajność i skalowalność aplikacji OpenMP ulegną znacznemu pogorszeniu.

Mając na uwadzę poprzednie definicje przyjrzyjmy się metodami obliczania liczb pierwszych i omówmy je.

- Dzielenie sekwencyjne

```
int isPrimeNumber(int number)
{
    if (number < 2)
    {
        return 1;
    }
    int sqrtNumber = sqrt(number) + 1;
    for (int i = 2; i < sqrtNumber; ++i)
    {
        if (number % i == 0)
        {
            return 1;
        }
    }
    return 0;
}

void sequentiallyFindPrimeNumbers(int start, int MAX, int *array)
{
    for (int i = start; i <= MAX; i++)
    {
        array[i] = 1;
    }

    for (int i = start; i <= MAX; ++i)
    {
        if (isPrimeNumber(i) == 1)
        {
            array[i] = 0;
        }
    }
}
```

Poprawne dzielenie sekwencyjne 1

Kod przedstawia implementację funkcji, która odpowiada za znalezienie wszystkich liczb pierwszych w podanym zakresie z wykorzystaniem metody dzielenia. Początkowo pętla for inicjalizuje tablicę, ustawiając wszystkie elementy na 1 (oznaczając, że są potencjalnie liczbami pierwszymi). W dalszej części algorytm zajmuje się „usuwaniem" liczb, które nie są liczbami pierwszymi – oznaczenie ich jako 0 - w tym celu korzysta z funkcji isPrimeNumber, której działanie opisaliśmy w poprzednim kroku.

Funkcja ta jest wykonywana sekwencyjnie przez jeden wątek. Oznacza to, że wszystkie operacje są wykonywane sekwencyjnie tzn. krok po kroku. Tym sposobem nie wykorzystujemy pełnego potencjału sprzętu, jednakże nie musimy się obawiać o poprawność wykonania kodu pod względem synchronizacji działania.

- Dzielenie równoległe

```
void parallelFindPrimeNumbers(int start, int MAX, int *array)
{
    #pragma omp parallel num_threads(12)
    {
        #pragma omp for
        for (int i = start; i <= MAX; i++)
        {
            array[i] = 1;
        }

        #pragma omp for schedule(dynamic)
        for (int i = start; i <= MAX; ++i)
        {
            if (isPrimeNumber(i) == 1)
            {
                array[i] = 0;
            }
        }
    }
}
```

Poprawne dzielenie równolegle 1

Kod przedstawia implementację funkcji, która odpowiada za znalezienie wszystkich liczb pierwszych w podanym zakresie z wykorzystaniem metody dzielenia w sposób równoległy. Na początku używamy dyrektywy **#pragma omp for**, która uruchamia blok kodu równolegle (rozdziela iteracje pętli pomiędzy dostępne wątki) - w przypadku tej funkcji, równolegle iteruje przez zakres liczb od poczatek do MAX i ustawia wartość 1 dla każdej liczby w tablicy. Takie działanie powinno usprawnić wykonanie inicjalizacji tablicy, ponieważ wątki pracują równolegle.

Następnie korzystamy z dyrektywy **#pragma omp for schedule(dynamic)**, która korzysta z dynamicznego przydzielania pracy dostępnym wątkom. W tym przypadku, równolegle iteruje przez zakres liczb od poczatek do MAX, sprawdza, czy liczba jest pierwsza, i aktualizuje tablicę. Praca jest dynamicznie przydzielana wątkom, co oznacza, że wątki mogą przejąć kolejne zadania w miarę ich dostępności.

W kodzie istnieje ryzyko wystąpienia wyścigu danych. Tablica jest współdzieloną strukturą danych, do której równocześnie dostęp mają różne wątki. Oba bloki kodu, które są oznaczone dyrektywą **#pragma omp for**, manipulują tą tablicą. Ponieważ te bloki są równoległe, istnieje możliwość, że dwa lub więcej wątków jednocześnie próbują aktualizować elementy tej tablicy. W szczególności, gdy jeden wątek ustawia 1, a inny ustawia 0, może to prowadzić do niestabilnego stanu, gdzie wynik zależy od kolejności wykonywania wątków.

W kodzie istnieje ryzyko wystąpienia false sharingu - w przypadku tablicy „array”, szczególnie w pętli, gdzie różne wątki jednocześnie modyfikują różne elementy tej tablicy. Ponieważ elementy tablicy są umieszczone w kolejnych komórkach pamięci, możliwe jest, że dwa wątki, pracujące na różnych elementach tablicy, będą wymagały dostępu do tej samej linii pamięci podręcznej, co potencjalnie prowadzi do false sharingu.

- Usuwanie przez dodawanie sekwencyjnie

```
void sequentialSieve(int* array, int max, int root) {
    for (int i = 2; i <= max; i++) {
        array[i] = 1;
    }

    for (int i = 2; i <= root; i++) {
        if (array[i] == 1) {
            for (int j = i * i; j <= max; j += i) {
                if (array[j] == 1)
                array[j] = 0;
            }
        }
    }
}
```

Poprawne podejście sekwencyjne sita 1

Kod przedstawia implementację funkcji sita Eratostenesa w sposób sekwencyjny (ang. sequential). Sito Eratostenesa to algorytm służący do znajdowania wszystkich liczb pierwszych w danym przedziale. Początkowa pętla **for** inicjalizuje tablicę **array**, ustawiając wszystkie elementy na 1 (oznaczając, że są potencjalnie liczbami pierwszymi). W następnej części algorytmu zajmujemy się usuwaniem liczb, które nie są liczbami pierwszymi, poprzez oznaczenie ich jako 0. Implementacja usuwania jest wykonywana w drugiej pętli **for** w funkcji. Ta pętla iteruje od 2 do pierwiastka kwadratowego z **max**. Ten zakres wystarcza do odfiltrowania liczb złożonych. Wewnątrz pętli sprawdzamy, czy liczba **i** jest oznaczona jako liczba pierwsza. W ten sposób unikamy zbędnych iteracji wewnętrznej pętli **for**. Wewnętrzna pętla **for** iteruje przez wielokrotności liczby **i**, np. wielokrotności liczby 2 (2, 4, 6, 8...), wielokrotności liczby 3 począwszy od 3*3, 3*5, 3*7, 3*9, 3*11 itd. Poprzez sprawdzenie warunku, czy wielokrotność nie jest liczbą pierwszą, unikamy zbędnych zapisów w tablicy. Oznaczamy wielokrotności liczby **i** jako liczby złożone, ustawiając odpowiednie elementy tablicy **array** na 0.

Powyższy algorytm posłuży nam jako podstawa do zrównoleglenia poprzez metodę funkcyjną oraz domenową.

Funkcja ta jest wykonywana sekwencyjnie przez jeden wątek. Oznacza to, że wszystkie operacje w obrębie algorytmu sita Eratostenesa są wykonywane sekwencyjnie, krok po kroku. Tym sposobem nie wykorzustejmy pełnego potencjału używanego sprzętu, jednakże nie musimy się obawiać o poprawność wykonania kodu pod względem synchronizacji działania.

- Usuwanie przez dodawanie równolegle metodą funkcyjną

```
void functionSieve(int* array, int max, int root) {
    #pragma omp parallel num_threads(12)
    {
        #pragma omp for
        for (int i = 2; i <= max; i++) {
            array[i] = 1;
        }

        for (int i = 2; i <= root; i++) {
            if (array[i] == 1) {
                #pragma omp for
                for (int j = i * i; j <= max; j += i) {
                    if (array[j] == 1)
                        array[j] = 0;
                }
            }
        }
    }
}
```

Poprawne podejście funkcyjne 1

Przyjrzyjmy się podejściu do zrównoleglenia przedstawionego wcześcniej algorytmu sita Eratostenesa. Na początku deklarujemy sekcję, która będzie wykonywana przez wiele wątków (tutaj 12). Deklaracja sekcji odbywa się za pomocą polecenia:

#pragma omp parallel num_threads(12)

Wewnątrz sekcji mamy do czynienia z tymi samymi pętlami co przy usuwaniu sekwencyjnie. Używamy polecenia: **#pragma omp for** aby dystrybuuować iteracje pętli pomiędzy dostępne wątki. Dzięki temu w teorii powinniśmy uzyskać lepszy czas wykonania inicjalizacji tablicy, ponieważ każda iteracja jest potencjalnie wykonana równolegle z innymi iteracjami. Nie musimy się martwić o zjawisko wyścigu, gdyż każdy wątek pracuje na innej części iteracji.

Następne użycie polecenia **#pragma omp for** wykonujemy dla pętli zajmującej się usuwaniem liczb złożonych. Obrane podejście wynika z definicji podejścia funkcjonalnego, które mówi, że procesy otrzymują całą tablicę wykreśleń (dlatego nie używamy polecenia **#pragma omp for** dla zewnętrznej pętli) i fragment zbioru liczb pierwszych, których wielekrotności są usuwane. Z tego powodu inicjujemy wykonanie równoległe dla wewnętrznej pętli i dzielimy zbiór liczb pierwszych, których wielokrotności są usuwane na fragmenty.

Kod ten nie wydaje się zawierać problemów z wyścigiem, ponieważ każdy wątek ma swoje własne iteracje do przetworzenia i zajmuje się inną wielokrotnością liczby pierwszej, co eliminuje konflikty.

W kodzie nie występuje wspólny dostęp do danych, co mogłoby prowadzić do false sharing. Każdy wątek pracuje na swojej części tablicy, eliminując ten problem. Omówmy jednak inne próbę optymalizacji tego podejścia.

```
void functionSieve(int* array, int max, int root) {
    #pragma omp parallel num_threads(12)
    {
        #pragma omp for schedule(static)
        for (int i = 2; i <= max; i++) {
            array[i] = 1;
        }

        for (int i = 2; i <= root; i++) {
            if (array[i] == 1) {
                #pragma omp for schedule(dynamic)
                for (int j = i * i; j <= max; j += i) {
                    if (array[j] == 1)
                        array[j] = 0;
                }
            }
        }
    }
}
```

Próba optymalizacji metody funkcyjnej 1

W tej próbie staramy się zastosować różne metody szeregowania poprzez dodanie dyrektywy **schedule**. Typ szeregowania określamy w nawasie. Zastosowaliśmy szeregowanie static dla inicjalizacji tabeli. Użycie statycznego harmonogramowania dla pętli inicjalizującej tablicę w tym przypadku może być uzasadnione, gdyż operacje w tej pętli są relatywnie proste i mają stały, przewidywalny czas wykonania dla każdej iteracji. W przypadku statycznego harmonogramowania, iteracje są przydzielane w sposób stały, co oznacza, że każdy wątek dostaje równą liczbę iteracji do wykonania, a podział pracy jest dokładnie określony przed wykonaniem pętli.

W przypadku pętli inicjalizującej tablicę, wszystkie iteracje wykonują podobne, prostsze operacje (ustawianie wartości elementu tablicy na 1), więc równomierne rozdzielenie pracy między wątki może przyspieszyć ten etap. Ponieważ operacje są prostsze i mają stały czas wykonania, statyczne harmonogramowanie może skutkować równomiernym obciążeniem wątków i potencjalnie efektywniejszym wykorzystaniem zasobów.

Użycie dynamicznego harmonogramowania dla wewnętrznej pętli, która wykreśla wielokrotności, może być uzasadnione ze względu na nieregularność i zróżnicowanie czasów wykonania poszczególnych iteracji tej pętli. W takim przypadku dynamiczne harmonogramowanie może pomóc w lepszym rozdzieleniu obciążenia między wątki, dostosowując się do zmienności czasów wykonania poszczególnych iteracji.

- Usuwanie przez dodawanie równolegle metodą domenową

```
void domainSieve(int* array, int max, int root) {
    #pragma omp parallel num_threads(12)
    {
        #pragma omp for
        for (int i = 2; i <= max; i++) {
            array[i] = 1;
        }

        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();

        int start = 2 + thread_id;

        for (int i = start; i <= root; i += num_threads) {
            if (array[i] == 1) {
                for (int j = i * i; j <= max; j += i) {
                    array[j] = 0;
                }
            }
        }
    }
}
```

Poprawne podejście domenowe 1

Przeanalizujmy metodę domenową usuwania przez dodawanie. Podobnie jak w przypadku podejścia funkcyjnego, deklarujemy sekcję wykonywaną równolegle. W obecnej metodzie podział na sekcję jest bardzo ważny, ponieważ pozwala nam na uzyskanie potrzebnych danych do podziału tablicy na fragmenty. Tak jak w poprzednich przypadkach używamy polecenia **#pragma omp for** do zrównoleglenia inicjalizacji tablicy. Dzięki temu w teorii otrzymujemy szybszy czas populacji tablicy ze względu na potencjalne równoległe wykonanie pracy przez wątki.

Pozyskujemy ID każdego z wątków oraz liczbę używanych wątków i zapisujemy je w zmiennych. Następnie obliczamy zmienną start, która jest zależna od ID danego wątku. W ten sposób początek obliczeń jest inny dla każdego wątku. Przeanalizujmy jak to wygląda w praktyce: wątek o ID równym 0 otrzymuje start równy 2, wątek o ID równym 1 otrzymuje start równy 3 itd.

Wykorzystujemy to samo podejście obliczania liczb pierwszych co w poprzednich metodach, jednakże tym razem ustawiamy inny start i zmieniamy krok obliczeń poprzez dodawanie do i liczby używanych wątków. Tym sposobem praca jest podzielona równo pomiędzy wątki, gdzie każdy wątek otrzymuje swoją część tablicy do obliczenia. Weźmy pod lupę przykład: wątkek o ID równym 0 obliczy iterację dla i równego 2, a następnie, zakładając że używamy 12 wątków, dodajemy do i 12, więc następna iteracja dla wątku to zmienna i równa 14. Dla wątku o ID równym 1 będzie to kolejno 3, 15, 27 itd. Tym sposobem dzielimy tablicę na następujące fragmenty: i=start, i+12, i+24, i+36, i+48 itd. Choć ogólnie taki podział pracy może poprawić lokalność dostępu, w tym konkretnym przypadku może to być utrudnione przez nieregularny dostęp do fragmentów tablicy. W teorii kolejne iteracje wykonywane przez każdy z wątków są w pobliżu siebie, co wzmacniałoby lokalność dostępu, jednakże wątki wykonują pracę w różnym tempie, przez co iteracje oddalają się od siebie.

Nie mamy do czynienia ze zjawiskiem wyścigu, ponieważ każdy wątek operuje na innym fragmencie tablicy. False sharing występuje, gdy różne wątki modyfikują różne elementy tablicy, ale te elementy znajdują się w tym samym cachu procesora i nie są oddalone daleko od siebie, przynajmniej przy pierwszych iteracjach.

Należy również zwrócić uwagę że przy iteracjach wewnętrznej pętli wykreślania wielokrotności brakuje warunku sprawdzającego czy dana iteracja jest liczbą pierwszą. Robimy tak, ponieważ przy podziale tabeli liczb pierwszych na fragmenty nie musimy się martwić o przechodzenie przez te same wielokrotności, dlatego też ten warunek jest zbędny, a usunięcie go zwiększa prędkość algorytmu ze względu na jedną operacje porównania mniej.

Dane podejście jest wykonane w taki sposób aby zgadzało się z definicją metody domenowej, gdzie procesy otrzymują fragment tablicy wykreśleń i cały zbiór liczb pierwszych, których wielokrotności są usuwane.

Nie stosujemy dyrektywy **#pragma omp for** dla zewnętrznej pętli obliczające liczby pierwszej, ponieważ powodowałoby to dodatkowy podział pracy wątków. Na tę chwilę każdy wątek ma swój obszar nad którym pracuje. Dodając tę dyrektywą spowodowalibyśmy, że dany fragment dodatkowo jest liczony przez wiele wątków. Zwiększa to obciążenie wątków, gdyż muszą one się dodatkowo zająć swoją wcześniej przydzieloną częścią tablicy.

```
#pragma omp parallel
    {
        #pragma omp for
        for (int i = 2; i <= max; i++) {
            array[i] = 1;
        }

        int num_threads = omp_get_num_threads();
        int thread_id = omp_get_thread_num();

        long long chunk_size = (sqrt(MAX) - 1) / num_threads + 1;
        long long start = MIN + thread_id * chunk_size;
        long long end = (thread_id == num_threads - 1) ? sqrt(MAX) : start + chunk_size - 1;

        // Upewnij się, że koniec nie przekracza MAX
        end = (end > sqrt(MAX)) ? sqrt(MAX) : end;

        for (long long i = start; i <= end; ++i) {
            if (array[i] == 1) {
                for (long long j = i * i; j <= MAX; j += i) {
                    array[j] = 0;
                }
            }
        }
    }
```

Próba optymalizacji podejścia domenowego 1

Zobaczmy jedną z prób optymalizacji podejścia domenowego. Standardowo używamy dyrektywy **#pragma omp for** aby zrównoleglić inicjalizację tablicy. Następnie pozyskujemy ilość używanych wątków, oraz ich ID. Podział pracy następuje w oparciu o dynamiczne przydzielanie iteracji pętli do

wątków. Każdy wątek otrzymuje pewien zakres wartości **i** do przetworzenia, co jest realizowane poprzez obliczenie **start** i **end** dla każdego wątku. **start** jest obliczane na podstawie numeru identyfikacyjnego bieżącego wątku i rozmiaru chunka. Określa on początkowy indeks zakresu wartości **i**, który będzie przetwarzany przez bieżący wątek. **end** jest obliczane w zależności od tego, czy bieżący wątek jest ostatnim wątkiem czy nie. Jeśli tak, to **end** ustawiane jest na sqrt(MAX). W przeciwnym razie, **end** to **start + chunk_size - 1**, co określa końcowy indeks zakresu wartości **i**, który będzie przetwarzany przez bieżący wątek.

Działanie opiera się na równoległym przetwarzaniu iteracji pętli, co teoretycznie zapewnia równomierne obciążenie wątków. Jednakże, równomierny podział pracy może być zakłócony przez nierównomierny rozkład liczb pierwszych w badanym zakresie – ilość pracy nie zależy tylko od szerokości badanej tablicy. Dodatkowo uniemożliwia to lokalność dostępu do danych w przypadku dużych instancji.

Nie mamy do czynienia ze zjawiskiem wyścigu, ponieważ każdy wątek operuje na innym fragmencie tablicy. Zjawisko false sharingu potencjalnie może wystąpić, gdy różne wątki jednocześnie operują na sąsiednich elementach tablicy. Jednak, w tym przypadku, prawdopodobieństwo false sharing jest niskie, ponieważ operacje są na ogół na oddalonych od siebie indeksach tablicy.

3. Prezentaja wyników i omówienie przebiegu eksperymentu obliczeniowo-pomiarowego

Wyniki przedstawiamy w tabeli pokazującej czas obliczeń, przyspieszenie, ilość obliczanych liczb na sekundę oraz efektywność zrównoleglenia. Wyniki odnoszą się do następujących kodów:

- Dzielenie sekwencyjnie – poprawne dzielenie sekwencyjnie 1

- Dzielenie równoległe – poprawne dzielenie równoległe 1

- Usuwanie przez dodawanie sekwencyjnie - Poprawne podejście sekwencyjne sita 2

- Usuwanie przez dodawanie równoległe metodą funkcyjną – Poprawne podejście funkcyjne 1

- Usuwanie przez dodawanie równoległe metodą domenową – Poprawne podejście domenowe 1

Testowane instancje są różne dla metody dzielenia i usuwania przez dodawanie. Dla metody dzielenia stosujemy MAX = 10 mln, natomiast dla usuwania przez dodawanie MAX = 100 mln. Zdecydowaliśmy się na taki podział, gdyż dla metody dzielenia tak duża instancja jak 100 mln powoduje bardzo długi czas obliczeń, przekraczający minutę. W ostateczności instancje prezentują się następująco:

Dzielenie:

- 2..MAX – [2, 10 000 000]

- MAX/2..MAX – [5 000 000, 10 000 000]

- 2..MAX/2 – [2, 5 000 000]

Usuwanie przez dodawanie:

- 2..MAX – [2, 100 000 000]

- MAX/2..MAX – [50 000 000, 100 000 000]

- 2..MAX/2 – [2, 50 000 000]

Dla obliczeń wykorzystaliśmy różne liczby wątków:

- Przetwarzanie równoległe dla maksymalnej liczby dostępnych w systemie procesorów logicznych, czyli w naszym przypaku 12

- Przetwarzanie równoległe dla maksymalnej liczby procesorów fizycznych, czyli w naszym przypadku 6

- Przetwarzanie równoległe dla liczby procesorów równej ½ liczby procesorów fizycznych, czyli w naszym przypadku 3.

Na następnej stronie prezentujemy tabelę z wynikami.

METODA DODAWANIA

INSTANCJE	SEKWENCYJNIE	RÓWNOLEGLE								
		LICZBA UŻYTYCH PROCESORÓW	PODEJŚCIE FUNKCYJNE				PODEJŚCIE DOMENOWE			
	CZAS PRZETWARZANIA		CZAS PRZETWARZANIA [s]	PRZYŚPIESZENIE	PRĘDKOŚĆ PRZETWARZANIA [l/s]	EFEKTYWNOŚĆ PRZETWARZANIA [%]	CZAS PRZETWARZANIA [s]	PRZYŚPIESZENIE	PRĘDKOŚĆ PRZETWARZANIA [l/s]	EFEKTYWNOŚĆ PRZETWARZANIA [%]
2 ... MAX	1.799747	3	0.689514	2.692	1.45e+08	89.731	0.894087	2.076	1.12e+08	69.200
		6	0.480689	3.865	2.08e+08	64.414	0.776277	2.393	1.29e+08	39.886
		12	0.457442	4.076	2.19e+08	33.967	0.577272	3.230	1.73e+08	23.994
MAX/2 ... MAX	1.857463	3	0.692887	2.674	7.22e+07	89.120	0.890381	2.081	5.62e+07	69.352
		6	0.481299	3.933	1.04e+08	65.557	0.768942	2.488	6.50e+07	41.470
		12	0.459333	4.104	1.09e+08	34.201	0.596875	3.134	8.38e+07	26.116
2 ... MAX/2	0.912515	3	0.339200	2.673	1.47e+08	89.099	0.417089	2.174	1.20e+08	72.460
		6	0.235443	3.901	2.12e+08	65.023	0.356920	2.574	1.40e+08	42.893
		12	0.222047	4.080	2.25e+08	33.996	0.277690	3.298	1.80e+08	27.482

METODA DZIELENIA

INSTANCJE	SEKWENCYJNIE	RÓWNOLEGLE				
	CZAS PRZETWARZANIA	LICZBA UŻYTYCH PROCESORÓW	CZAS PRZETWARZANIA [s]	PRZYŚPIESZENIE	PRĘDKOŚĆ PRZETWARZANIA [l/s]	EFEKTYWNOŚĆ PRZETWARZANIA [%]
2 ... MAX	8.516115	3	2.977039	2.905	3.36e+06	96.844
		6	1.551821	5.483	6.44e+06	91.392
		12	1.480268	5.753	6.76e+06	47.942
MAX/2 ... MAX	5.437753	3	1.859831	2.924	2.69e+06	97.460
		6	0.958767	5.684	5.22e+06	94.729
		12	0.927734	5.776	5.39e+06	48.134
2 ... MAX/2	3.213382	3	1.123417	2.860	4.45e+06	95.345
		6	0.595653	5.416	8.39e+06	90.271
		12	0.552413	5.831	9.05e+06	48.593

Dzielenie:

- 2..MAX – [2, 10 000 000]
- MAX/2..MAX – [5 000 000, 10 000 000]
- 2..MAX/2 – [2, 5 000 000]

Usuwanie przez dodawanie:

- 2..MAX – [2, 100 000 000]
- MAX/2..MAX – [50 000 000, 100 000 000]
- 2..MAX/2 – [2, 50 000 000]

Warto z góry zwrócić uwagę na wyniki metody dodawania dla instancji MAX/2..MAX. Widzimy, że w porównaniu z instancją 2..MAX/2 wyniki są gorsze, czasami nawet osiągają prędkości niższego rzędu. Jest to spowodowane brakiem poprawnej implementacji obliczania zakresu od MAX/2. W trakcie implementacji napotkaliśmy problem, gdzie algorytm działa świetnie dla instancji zaczynającej się od 2, jednakże zwraca niepoprawny wynik dla instancji zaczynającej się od innej liczby niż 2. Po wielu próbach udało się nam znaleźć sposób, który by zaradził temu problemowi, jednakże spowodowało to, że prędkość obliczania znacznie spadła. Oto rozwiązanie, które zostało zastosowane:

```
void sequentialSieve(int* array, int max, int root, int min) {
    for (int i = 2; i <= max; i++) {
        array[i] = 1;
    }

    for (int i = 2; i <= root; i++) {
        if (array[i] == 1) {
            for (int j = (i * i > min) ? i * i : min; j <= max; j += i) {
                if (array[j] == 1) {
                    array[j] = 0;
                }
            }
        }
    }
}
```

Próba naprawy zakresu 1

Takie podejście znacznie zwiększa ilość operacji do wykonania, co powoduje mniejszą prędkość algorytmu. W naszych testach podejście obliczania metodą pokazaną w *Poprawne podejście sekwencyjne sita 3* nadal było szybsze, mimo iż obliczało zakres od 2 do MAX i tylko wypisywało sam wynik od MAX/2 do MAX. Dlatego też zrezygnowaliśmy z podejścia pokazanego w *Próba naprawy zakresu 1.*

4. Wnioski

Porównując metodę dzielenia i dodawania sekwencyjnego możemy zauważyć ogromną różnicę w czasie wykonania i prędkości przetwarzania. Sam czas wykonania dzielenia dla 10 krotnie mniejszego MAX wynosi 8.5 sekundy, natomiast dla dodawania 1.79 sekundy, ale dla instancji znacznie większej. W przeprowadzonych testach instancja 100 mln używając metody dzielenia zajmowała się obliczaniem blisko 4 minuty, czyli 130 razy dłużej niż sekwencyjna metoda dodawania.

W naszym przypadku metoda dodawania funkcyjnego okazała się być najszybszą metodą. Algorytm osiągnął prędkość przetwarzania rzędu 10^8 z przewagą nad metodą domenową. Biorąc pod uwagę instancję 2..MAX osiągnęliśmy najlepsze czasy przy korzystaniu z 12 wątków. Podejście funkcyjne osiągnęło prędkość 2.19e+08, natomiast podejście domenowe 1.73e+08.

Pod względem czystych liczb króluje metoda dzielenia równoległego, która okazała się efektywna w okolicach 90% dla 3 i 6 wątków oraz w okolicach 48% dla 12 wątków. Przyspieszenie jest 5-krotne dla 6 i 12 wątków w porównaniu z wykonaniem sekwencyjnym. Musimy jednak mieć na uwadze, że te liczby są dla 10-krotnie mniejszej instancji w porównaniu z metodą dodawania.

Analizując instancję 2..MAX, metoda dodawania prezentuje efektywność nawet rzędu 89% dla podejścia funkcyjnego z 3 wątkami. Natomiast dla 6 i 12 wątków jest to kolejno 64% i 33%. Podejście

domenowe odpowiada efektywnością dla 3, 6 i 12 wątków kolejno: 23%, 39%, 69%. Przyspieszenie również wypada na korzyść podejścia funkcjonalnego. Dla 12 wątków osiągamy nawet 4-krotne przyspieszenie, a dla podejścia domenowego 3-krotne przyspieszenie. Jest to najprawdopodobniej spowodowane nieoptymalizacją kodu zaprezentowanego przez nas jako Poprawne podejście domenowe 1. Mamy tam do czynienia z potencjalnym false sharingiem, który wpływa na jakość wyników. Przyjrzyjmy się próbom zoptymalizowania podejścia domenowego. Biorąc pod lupę kod Próba optymalizacji podejścia domenowego 1, widzimy, że podział jest dzielony na podstawie liczby wątków. Takie podejście przyniosło nam prędkość rzędu 10^7 w porównaniu do Poprawne podejście domenowe 1, gdzie uzyskaliśmy prędkość rzędu 10^8. Możemy zatem wyciągnąć wnioski, że potencjalne zjawisko false sharingu jest w tym przypadku mniej wpływowe niż nierównomierny podział pracy na wątkach. Co jeszcze bardziej potwierdza ten argument, to fakt, że w Próba optymalizacji podejścia domenowego 1 zjawisko false sharingu nie występuje.

Spójrzmy również na kod Próba optymalizacji metody funkcyjnej 1. Staramy się tam odpowiednio zastosować szeregowanie dla dyrektywy #pragma omp for. Zastosowane szeregowanie w kodzie Próba optymalizacji metody funkcyjnej 1 są tylko przykładowymi, które zastosowaliśmy. W naszej opinii wypadają najlepiej pod względem teoretycznym. Niestety, nie osiągnęliśmy lepszych wyników dla żadnego z szeregowania. Uzyskiwaliśmy przyspieszenie 1-krotnie mniejsze niż w algorytmie Poprawne podejście funkcyjne 1. Ostatecznie nie udało się nam znaleźć bardziej optymalnej wartości szeregowania.

Zwróćmy uwagę na zależność prędkości przetwarzania od ilości używanych wątków i wielkości instancji. Niestety, ze względu na problem zakresu dla metody dodawania opisanego wcześniej, nie posiadamy jednoznacznej odpowiedzi co do różnic zależnych od wykonywanej instancji. Jednakże, analizując metodę dzielenia, jesteśmy w stanie wyciągnąć wnioski, że wykonanie instancji 2..MAX/2 jest szybsze niż wykonanie instancji MAX/2..MAX. Dzieje się tak, ponieważ w instancji do połowy MAX mamy do czynienia z liczbami o mniejszej złożoności. Widzimy natomiast, że większa ilość wątków pozytywnie wpływa na prędkość, choć różnica pomiędzy użyciem 3 i 6 wątków, a użyciem 6 i 12 wątków, jest większa w przypadku porównania 3 i 6 wątków, za wyjątkiem podejścia domenowego. Możemy zaobserwować, że dla przykładowo instancji 2..MAX/2 dla metody dzielenia prędkość wzrasta z 4.45e+06 do 8.39e+06 dla 3 i 6 wątków, natomiast dla 6 i 12 jest to kolejny wzrost z 8.39e+06 do 9.05e+06. Zatem w przypadku metody dzielenia wzrost prędkości pomiędzy użyciem 3 a 6 wątków jest niemalże 2-krotny. Dla metody dodawania i podejścia funkcyjnego są to kolejno wzrosty z 1.47e+08 do 2.12e+08, a następnie do 2.25e+08. Wzrost nie jest tak zatrważający jak dla metody dzielenia, jednakże możemy zauważyć podobne zjawisko. W podejściu domenowym widzimy wzrosty od 1.20e+08 dla 3 wątków do 1.40e+08 dla 6 wątków, poprzez 1.80e+08 dla 12 wątków. Widzimy więc, że uzyskujemy większy wzrost przy zmianie z 6 na 12 wątków. Jest to spowodowane zjawiskiem false sharingu, które w naszym przypadku nasila się wraz ze spadkiem liczby używanych wątków. W pozostałych dwóch przypadkach coraz mniejszy wzrost prędkości obliczeń może być spowodowany zwiększonym zapotrzebowaniem na komunikację między wątkami.

Efektywność skaluje się wraz ze wzrostem liczbą wątków. Zauważamy, że im mniejsza liczba wątków, tym większa efektywność wykonywanego algorytmu. Dzieje się tak w obydwu metodach i w obydwu podejściach w metodzie dodawania. Efektywność obliczamy poprzez iloraz czasu wykonania sekwencyjnego przez iloczyn liczby użytych wątków i czasu wykonania równoległego. Mniejsza efektywność może być spowodowana koniecznością komunikacji między wątkami oraz przepełnienia pamięci podręcznej (tzw. cache trashing).

Największym ograniczeniem przy podejściu domenowym w metodzie dodawania jest odpowiedni dobór podziału tablicy na fragmenty. W naszym przypadku taki a nie inny dobór doprowadza do

zjawiska false sharingu. Największym ograniczeniem podejścia funkcyjnego jest odpowiednia synchroniczność dyrektywy #pragma omp for. Jest tak, ponieważ możemy napotkać zbyt małe zadania dla wątków, a co za tym idzie, spadek efektywności. Natomiast największym ograniczeniem metody równoległego dzielenia jest fakt, że każda iteracja pętli drugiej (**#pragma omp for schedule(dynamic)**) jest zależna od wyniku poprzednich iteracji. W praktyce oznacza to, że wynik sprawdzenia liczby pierwszej dla jednej liczby wpływa na wyniki dla kolejnych liczb, co utrudnia efektywną równoległość.