Jakub Ciulęba 148103 jakub.ciuleba@student.put.poznan.pl Rzeczywisty: 01.05.2023

Kamil Malinowski 148214 kamil.malinowski@student.put.poznan.pl Wymagany: 02.05.2023

Wtorek parzyste 9:45 grupa L12 Pierwsza wersja sprawozdania

Projekt 1 OMP

Wstęp:

Zadanie polegało na wyznaczeniu liczb pierwszych poprzez wykorzystanie różnych algorytmów (metoda dzielenia, sito Eratostenesa), a następnie zrównolegleniu algorytmu sita poprzez wykorzystanie funkcji biblioteki OpenMP. Zostały zastosowane dwa podejścia: domenowe oraz funkcyjne w próbach zrównoleglenia kodu.

Punkt 1: Sprzęt i oprogramowanie

Procesor	System i oprogramowanie
Intel® Core™ i5-8600K Liczba procesorów fizycznych: 6 Liczba procesorów logicznych: 6 **Cache L1**: 64 KB / rdzeń **Cache L2**: 256 KB / rdzeń **Cache L3**: 9 MB	Windows 11 Home 64 bit Visual Studio 2022 Intel oneAPI Base Toolkit 2023.1.0 zintegrowane w środowisku Visual Studio

Punkt 2: Prezentacja kodów

Wersja 1 kodu: Metoda dzielenia:

Kod 2.1

```
#include <iostream>
#include <math.h>
#include <time.h>

#define START 2
#define LIMIT 1000
#define PRIME true;
#define COMPLEX false;
using namespace std;

bool CheckIfPrime(int number)
{
    for (int i = 2; i <= sqrt(number); i++)
        if (number % i == 0)
            return COMPLEX;
    return PRIME;
}

int main()
{
    clock_t start, stop;
    int count = 0;

    start = clock();

    for (int i = START; i < LIMIT; i++)
        if (CheckIfPrime(i))
            count++;

    stop = clock();

    cout << endl << "Found " << count << " primes" << endl;
    cout << "Processing " << LIMIT << " took: " << (double)(stop - start) /
CLOCKS_PER_SEC << " sec" << endl;
}
```

START, LIMIT – dyrektywa do jakiej liczby ma działać główna pętla programu

PRIME, COMPLEX – dyrektywy określające czy liczba jest czy nie jest pierwsza

Funkcja zwracająca czy liczba jest złożona czy nie

Algorytm przetwarza liczby nieparzyste w pętli od START do LIMIT, gdy znajdzie liczbę pierwszą, zmienna count jest inkrementowana

Wersja 2 kodu: Sekwencyjne sito Eratostenesa

Kod 2.2

```
#include <iostream>
#include <ctime>
#include <cmath>
#include <vector>

#define START 1000000000
#define LIMIT 2000000000
#define PRIME true
#define COMPLEX false

using namespace std;

bool* sieve(int start, int end)
{
    clock_t start_t, stop_t;
    bool* primes = new bool[end + 1];
    for (int i = 0; i <= end; i++)
        *(primes + i) = PRIME;
    primes[0] = primes[1] = false;
    start_t = clock();
    int limit = sqrt(end);
    for (int i = 2; i <= limit; i++)
        if (*(primes + i))
            for (int j = i * i; j <= end; j += i)
                *(primes + j) = COMPLEX;
    stop_t = clock();
    cout << "Processing " << START << " to " << LIMIT << " took: " <<
(double)(stop_t - start_t) / CLOCKS_PER_SEC << "s" << endl;
    return primes;
}

int main()
{
    int count = 0;
    bool* primes = sieve(START, LIMIT);

    for (int i = START; i < LIMIT; i++)
        if (*(primes + i))
            count++;
    cout << "Found " << count << " primes" << endl;
    return 0;
}
```

START, LIMIT – dyrektywa do jakiej liczby ma działać główna pętla programu

PRIME, COMPLEX – dyrektywy określające czy liczba jest czy nie jest pierwsza

Inicjalizacja tablicy primes

Iterowanie po tablicy primes do pierwiastka odgórnej granicy, wykreślanie wielokrotności liczby pierwszej

Wersja 3 kodu: Równoległe sito Eratostenesa, podejście funkcyjne

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <omp.h>
#include <math.h>

#define PRIME true
#define COMPLEX false

#define B1 1000000000
#define B2 2000000000

using namespace std;

int threads = omp_get_max_threads();
double start_t, stop_t;

void basicSieve(int upper, vector<int>& input)
{
    bool* primes = new bool[upper - 1];
    for (int i = 0; i <= upper - 2; i++)
        primes[i] = PRIME;

    for (int p = 2; p * p <= upper; p++) {
        if (primes[p - 2] == true) {
            for (int i = p * p; i <= upper; i += p)
                primes[i - 2] = false;
        }
    }

    for (int i = 0; i <= upper - 2; i++)
        if (primes[i] == PRIME) {
            input.push_back(i + 2);
        }
    delete primes;
}
```

B1, B2 – dyrektywa do jakiej liczby ma działać główna pętla programu

PRIME, COMPLEX – dyrektywy określające czy liczba jest czy nie jest pierwsza

Kod sita sekwencyjnego do pierwiastka z górnego zakresu

Kod 2.3

Wersja 3 kodu: Równoległe sito Eratostenesa, podejście funkcyjne c.d.

Kod 2.3
c.d.

```
vector<int> function(int lower, int upper)
{
    vector<int> primes;
    vector <int> primesToSqrt;
    vector <vector<bool>> primeInRange(threads);
    start_t = omp_get_wtime();
    basicSieve(sqrt(upper), primesToSqrt);


#pragma omp  parallel
    {
        vector<bool> localIsPrime(upper - lower + 1, PRIME);

#pragma omp  for schedule(dynamic)
        for (int i = 0; i < primesToSqrt.size(); i++)
        {
            int sieved = primesToSqrt[i];
            int number = lower;
            for (; number % sieved != 0; number++)
                continue;
            if (number == sieved)
                number *= 2;

            for (; number <= upper; number += sieved)
                localIsPrime[number - lower] = COMPLEX;
        }
        primeInRange[omp_get_thread_num()] = localIsPrime;
    }

    stop_t = omp_get_wtime();

    /*
    vector <bool> primesMerge(2, COMPLEX);
    bool flag;
    for (int i = 0; i < upper - lower + 1; i++)
    {
        flag = true;
        for (int j = 0; j < threads; j++)
            flag = flag && primeInRange[j][i];
        primesMerge.push_back(flag);
    }

    for (int i = 0; i < primesMerge.size(); i++)
        if (primesMerge[i] == PRIME)
            primes.push_back(i);*/
    cout << "Processing " << upper - lower << " took: " << stop_t - start_t << "
sec" << endl;
    return primes;
}


int main()
{
    vector<int> result = function(2, B1);
}
```

Inicjalizacja tablic przechowujących dane, znalezienie liczb pierwszych do pierwiastka z B2

Zmienna sieved oznacza kolejne liczby pierwsze znalezione przez sito sekwencyjne

W pętli kolejno wykreślamy wielokrotności liczb pierwszych

Na końcu przypisujemy wynik do indywidualnego wektora wątku

W kodzie zastosowano równoległe przetwarzanie z użyciem OpenMP, które pozwala na efektywniejsze poszukiwanie liczb pierwszych w zakresie od lower do upper. Zbiór zadań jest podzielony na sekcje równoległe, gdzie każdy wątek ma przydzielony swój lokalny wektor localIsPrime. Wielkość zbioru zadań zależy od ilości liczb pierwszych w przedziale sqrt(upper). Koszt wykonania zadania jest zmienny i zależy od ilości liczb pierwszych do sprawdzenia.

W kodzie zastosowano dyrektywę #pragma omp parallel oraz #pragma omp for schedule(dynamic) do przydzielania zadań do wątków. Dynamiczne przydzielanie zadań oznacza, że OpenMP przydziela zadania do wątków w miarę ich dostępności.

Zastosowanie OpenMP pozwala na równoległe przetwarzanie, co skutkuje lepszym wykorzystaniem zasobów systemowych. Wybór dynamicznego przydzielania zadań (schedule(dynamic)) pozwala na lepsze zrównoważenie obciążenia procesorów, gdyż zadania są przydzielane do wątków w miarę ich dostępności.

#pragma omp parallel: Dyrektywa ta tworzy równoległe regiony kodu, które są wykonywane przez wiele wątków jednocześnie.

#pragma omp for schedule(dynamic): Dyrektywa ta pozwala na równoległe wykonanie pętli for z dynamicznym przydzielaniem zadań do wątków, co wpływa na efektywność przetwarzania. Pozwala to na lepsze zrównoważenie obciążenia procesorów.

Wyścig to sytuacja, w której wynik operacji zależy od tego, który wątek zakończy wykonywanie swojego zadania jako pierwszy. W analizowanym kodzie nie występuje wyścig, ponieważ każdy wątek pracuje na swoim własnym lokalnym wektorze localIsPrime, co eliminuje możliwość kolizji między wątkami podczas zapisu lub odczytu danych. Ponadto, gdy wątki kończą swoje zadania, ich wyniki są łączone w sposób sekwencyjny, co zapewnia poprawność wyników. Brak występowania wyścigów w tym przypadku przyczynia się do poprawy prędkości przetwarzania, ponieważ wątki mogą działać niezależnie i nie muszą oczekiwać na dostęp do wspólnych zasobów.

Fałszywe współdzielenie (ang. false sharing) to zjawisko występujące w programach równoległych, które korzystają z pamięci podręcznej (cache) wielowątkowych procesorów. Fałszywe współdzielenie może prowadzić do nieoptymalnej wydajności programu z powodu niepotrzebnych operacji na pamięci podręcznej. Zjawisko to występuje, gdy różne wątki niezależnie modyfikują zmienne lub dane, które są umieszczone blisko siebie w pamięci i mieszczą się w jednej linii pamięci podręcznej.

Kiedy wątek modyfikuje zmienną znajdującą się w linii pamięci podręcznej, ta linia musi być uaktualniona. Jeśli jednak inny wątek próbuje jednocześnie modyfikować inną zmienną w tej samej linii pamięci podręcznej, linia ta musi być wcześniej unieważniona, a następnie ponownie wczytana do pamięci podręcznej drugiego wątku. W efekcie, operacje na pamięci podręcznej są wykonywane wielokrotnie, co prowadzi do spadku wydajności programu.

W celu zminimalizowania fałszywego współdzielenia, można stosować różne techniki, takie jak:

Pady: Dodanie dodatkowego miejsca pomiędzy zmiennymi, które są modyfikowane przez różne wątki, tak aby zmienne te nie były umieszczone w tej samej linii pamięci podręcznej.

Używanie lokalnych zmiennych: Przechowywanie zmiennych w stosie wątku (przy użyciu zmiennych lokalnych) zamiast w pamięci współdzielonej. Wątki nie będą się wzajemnie wpływać na siebie, ponieważ będą korzystać z własnych kopii zmiennych.

Wyrównywanie danych: Umieszczanie zmiennych, które mają być modyfikowane przez różne wątki, w oddzielnych liniach pamięci podręcznej, na przykład poprzez wykorzystanie atrybutów wyrównywania kompilatora.

W analizowanym kodzie false sharing raczej nie występuje, ponieważ każdy wątek pracuje na swoim własnym, lokalnym wektorze localIsPrime.

W kodzie nie ma jawnie zastosowanej synchronizacji między wątkami. Istnieje jednak niejawna synchronizacja wynikająca z końca równoległego regionu kodu przed dyrektywą #pragma omp parallel. Na końcu tego regionu wszystkie wątki muszą zakończyć swoje zadania, zanim program przejdzie do kolejnej sekwencyjnej części kodu. Ta synchronizacja wpływa na czas obliczeń, ponieważ program musi czekać na zakończenie wszystkich wątków przed kontynuowaniem, ale ogólnie wpływ ten będzie pozytywny, ponieważ zapewnia poprawność wyników.

W kodzie zastosowano dynamiczne przydzielanie zadań (schedule(dynamic)), co prowadzi do lepszego zrównoważenia obciążenia procesorów. Dzięki temu zadania są przydzielane do wątków w miarę ich dostępności, co oznacza, że wątki, które kończą swoje zadania wcześniej, mogą otrzymać kolejne zadania do wykonania. To podejście prowadzi do lepszego zrównoważenia obciążenia procesorów i ogólnie przyczynia się do poprawy wydajności obliczeń.

Jednakże, mogą wystąpić sytuacje, w których niektóre wątki mogą być bardziej obciążone niż inne ze względu na różne ilości pracy do wykonania. Ponieważ zadania są przydzielane dynamicznie, może być trudno przewidzieć, jak obciążenie zostanie rozłożone na wątki. W przypadku znaczącego nierównomiernego obciążenia, niektóre wątki mogą kończyć swoje zadania wcześniej niż inne, prowadząc do sytuacji, w której niektóre wątki oczekują na zakończenie pracy przez inne. W takim przypadku, zastosowanie bardziej zaawansowanych technik równoważenia obciążenia, takich jak dynamiczne przydzielanie zadań opartych na wielkości obciążenia, może być pomocne w dalszym poprawieniu wydajności obliczeń.

Wersja 4 kodu: Równoległe sito Eratostenesa, podejście domenowe

Kod 2.4

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <omp.h>
#include <math.h>

#define PRIME true
#define COMPLEX false
#define B1 1000000000
#define B2 2000000000

using namespace std;

int threads = omp_get_max_threads();
double start_t, stop_t;

void displayResults(vector <int> primes)
{
    cout << "Primes: " << endl;

    // for(int prime : primes)
    //     cout<<prime<<" ";

    cout << endl << "Found " << primes.size() << " primes" << endl;
}

int** divide(int lowerLimit, int upperLimit)
{
    int** output = new int* [threads];
    for (int i = 0; i < threads; i++)
        output[i] = new int[2];
    int intsPerSet = (upperLimit - lowerLimit) / threads;
    for (int i = 1; i <= threads; i++)
        if (i == threads)
        {
            output[i - 1][0] = lowerLimit + intsPerSet * (i - 1);
            output[i - 1][1] = upperLimit;
        }
        else
        {
            output[i - 1][0] = lowerLimit + intsPerSet * (i - 1);
            output[i - 1][1] = lowerLimit + intsPerSet * i - 1;
        }
    return output;
}
```

Podział zbioru na podzakresy realizowany funkcją divide

intsPerSet oznacza ilość liczb w przedziale dla wątku

Przypisujemy górną i dolną granicę przetwarzania do zwracanej tablicy dwuwymiarowej

Wersja 4 kodu: Równoległe sito Eratostenesa, podejście domenowe c.d.

Kod 2.4 c.d.

```
void basicSieve(int upper, vector<int>& input)
{
    bool* primes = new bool[upper - 1];
    for (int i = 0; i <= upper - 2; i++)
        primes[i] = PRIME;

    for (int p = 2; p * p <= upper; p++) {
        if (primes[p - 2] == true) {
            for (int i = p * p; i <= upper; i += p)
                primes[i - 2] = false;
        }
    }

    for (int i = 0; i <= upper - 2; i++)
        if (primes[i] == PRIME) {
            input.push_back(i + 2);
        }
    delete primes;
}
```

Sito sekwencyjne do pierwiastka z górnego zakresu

```
vector<int> domain(int lower, int upper)
{
    vector<int> primes;
    int** subsets = divide(lower, upper);
    vector <int> primesToSqrt;
    vector <vector<bool>> subsetOutput(threads);

    start_t = omp_get_wtime();
    basicSieve(sqrt(upper), primesToSqrt);

#pragma omp  parallel
    {
        int threadNum = omp_get_thread_num();

        vector<bool> subset(subsets[threadNum][1] - subsets[threadNum][0] + 1,
PRIME);

        for (int i = 0; i < primesToSqrt.size(); i++)
        {
            int sieved = primesToSqrt[i];
            int number = subsets[threadNum][0];
            for (; number % sieved != 0; number++)
                continue;
            if (number == sieved)
                number *= 2;

            for (; number <= subsets[threadNum][1]; number += sieved)
                subset[number - subsets[threadNum][0]] = COMPLEX;
            }
        }
        subsetOutput[threadNum] = subset;
    }

    stop_t = omp_get_wtime();
```

Inicjalizacja wektora z przedziałami dla wątków

Główna pętla programu, w której dla zadanego zakresu na podstawie wektora liczb pierwszych do pierwiastka wykreślamy ich wielokrotności

Wersja 4 kodu: Równoległe sito Eratostenesa, podejście domenowe c.d.

Kod 2.4
c.d.

```
    /*for (int i = 0; i < threads; i++)
    {
        for (int j = 0; j < subsetOutput[i].size(); j++)
        {
            if (subsetOutput[i][j] == PRIME)
            {
                primes.push_back(subsets[i][0] + j);
            }
        }
    }*/

    cout << "Processing " << upper - lower << " took: " << stop_t - start_t << "
sec" << endl;

    return primes;
}

int main()
{
    vector<int> result = domain(2, B1);
}
```

Zbiór zadań jest podzielony na podzbiory, a każdy z podzbiorów zawiera liczby, które będą badane przez wątek. Wielkość zbioru zadań zależy od liczby dostępnych wątków i zakresu liczb, które są analizowane. Koszt wykonania zadania (czas) zależy od liczby liczb pierwszych w danym zakresie.

Zadania są przydzielane do procesów za pomocą funkcji "divide", która dzieli zakres liczb na podzbiory, z których każdy zostaje przypisany do jednego wątku. Liczba zadań przydzielana do procesów zależy od liczby dostępnych wątków.

Podział przetwarzania na zadania pozwala na równoległe wykonywanie obliczeń, dzięki czemu oszczędza czas i zwiększa wydajność. Wybrany sposób przydzielania zadań pozwala na równomierny podział pracy między wątki, co prowadzi do lepszego zrównoważenia procesorów przetwarzaniem.

#pragma omp parallel: Dyrektywa ta uruchamia równoległe wykonanie bloku kodu przez dostępne wątki.

omp_get_max_threads(): Funkcja ta zwraca maksymalną liczbę wątków, które mogą być używane w regionie równoległym.

omp_get_thread_num(): Funkcja ta zwraca numer wątku, który aktualnie wykonuje się w regionie równoległym.

Omówienie występujących w kodzie potencjalnych problemów poprawnościowych:

W powyższym kodzie nie ma problemów z wyścigami, ponieważ każdy wątek działa na swoim własnym podzbiorze liczb, a wektor wynikowy nie jest modyfikowany w trakcie przetwarzania równoległego.

Omówienie występujących w kodzie potencjalnych problemów efektywnościowych:

W analizowanym kodzie false sharing nie występuje, ponieważ każdy wątek działa na swoim własnym podzbiorze liczb, a wektor wynikowy nie jest modyfikowany w trakcie przetwarzania równoległego.

Synchronizacja:

Synchronizacja odnosi się do zarządzania dostępem do wspólnych zasobów przez różne wątki. W powyższym kodzie nie ma potrzeby synchronizacji, ponieważ każdy wątek działa na swoim własnym podzbiorze liczb i nie ma żadnych wspólnych zasobów, które muszą być chronione przed równoczesnym dostępem.

Brak zrównoważenia procesorów obliczeniami:

Brak zrównoważenia procesorów obliczeniami występuje, gdy różne wątki wykonują różne ilości pracy, co prowadzi do marnowania zasobów. W powyższym kodzie nie ma dużego braku zrównoważenia procesorów obliczeniami, ponieważ zadania są przydzielane równomiernie do wątków. Niemniej jednak, w niektórych przypadkach, gdy liczba liczb pierwszych w danym podzbiorze jest znacząco różna, może wystąpić pewien brak zrównoważenia. Jednym ze sposobów rozwiązania tego problemu jest dynamiczne przydzielanie zadań, które pozwala na bardziej równomierne rozłożenie pracy między wątki.

Poprzednie wersje kodu:

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <omp.h>
#include <math.h>

#define PRIME true
#define COMPLEX false
#define K1 1000
#define K10 10000
#define K100 100000
#define M1 1000000
#define M10 10000000
#define M100 100000000
#define B1 1000000000
#define B2 2000000000

using namespace std;

int threads = omp_get_max_threads();
double start_t, stop_t;

void displayResults(vector <int> primes)
{
    /*   cout << "Primes: " << endl;
```

```
        for(int prime : primes)
            cout<<prime<<" ";*/

    cout << endl << "Found " << primes.size() << " primes" << endl;
}


void basicSieve(int upper, vector<int>& input)
{
    bool* primes = new bool[upper - 1];
    for (int i = 0; i <= upper - 2; i++)
        *(primes + i) = PRIME;


    for (int p = 2; p * p <= upper; p++) {
        if (*(primes + p - 2)) {
            for (int i = p * p; i <= upper; i += p)
                *(primes + i - 2) = false;
        }
    }

    for (int i = 0; i <= upper - 2; i++)
        if (*(primes + i)) {
            input.push_back(i + 2);
        }
    delete primes;
}

vector<int> function(int lower, int upper)
{
    vector<int> primes;
    vector <int> primesToSqrt;
    bool* subOutput1, * subOutput2, * subOutput3, * subOutput4, * subOutput5, *
subOutput6;
    start_t = omp_get_wtime();
    basicSieve(sqrt(upper), primesToSqrt);

#pragma omp  parallel
    {
        int subsetRange = upper - lower + 1;
        int threadNum = omp_get_thread_num();
        bool* localPrime = new bool[subsetRange];
        for (int i = 0; i < subsetRange; i++)
            *(localPrime + i) = PRIME;

#pragma omp  for schedule(dynamic)
        for (int i = 0; i < primesToSqrt.size(); i++)
        {
            int sieved = primesToSqrt[i];
            int number = lower;
            for (; number % sieved != 0; number++)
                continue;
            if (number == sieved)
                number *= 2;

            for (; number <= upper; number += sieved)
                *(localPrime + number - lower) = COMPLEX;
        }
        if (threadNum == 0)
            subOutput1 = localPrime;
        else if (threadNum == 1)
            subOutput2 = localPrime;
```

Kod 2.5

```
            else if (threadNum == 2)
                subOutput3 = localPrime;
            else if (threadNum == 3)
                subOutput4 = localPrime;
            else if (threadNum == 4)
                subOutput5 = localPrime;
            else if (threadNum == 5)
                subOutput6 = localPrime;

            delete localPrime;
        }

        stop_t = omp_get_wtime();

        for (int i = 0; i < upper - lower + 1; i++)
            if (*(subOutput1 + i) && *(subOutput2 + i) && *(subOutput3 + i) &&
*(subOutput4 + i) && *(subOutput5 + i) &&
                *(subOutput6 + i))
                primes.push_back(i);
        cout << "Processing " << upper - lower << " took: " << stop_t - start_t << "
sec" << endl;
        return primes;
}

int main()
{
        vector<int> result = function(2, B1);
        displayResults(result);

}
```

Powyższe podejście było próbą podejścia funkcyjnego, jednakże wyniki nie były satysfakcjonujące z powodu zbyt dużego ograniczenia Core Bound, a także szybkością działania programu. Było to spowodowane złym zarządzaniem pamięcią, a co za tym idzie także false sharingiem. Dodatkowym problemem w tym wypadku było zastosowanie dyrektywy schedule(dynamic), co powodowało dodatkową synchronizację pomiędzy wątkami, co jeszcze bardziej spowodowało spowolnienie programu.

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <omp.h>
#include <math.h>

#define PRIME true
#define COMPLEX false
#define K1 1000
#define K10 10000
#define K100 100000
#define M1 1000000
#define M10 10000000
#define M100 100000000
#define B1 1000000000
#define B2 2000000000
#define LIMIT 2000000000

using namespace std;

int threads = omp_get_max_threads();
double start_t, stop_t;

void displayResults(vector <int> primes)
{
    cout << "Primes: " << endl;

    // for(int prime : primes)
    //     cout<<prime<<" ";

    cout << endl << "Found " << primes.size() << " primes" << endl;
}

int** divide(int lowerLimit, int upperLimit)
{
    int** output = new int* [threads];
    for (int i = 0; i < threads; i++)
        output[i] = new int[2];
    int intsPerSet = (upperLimit - lowerLimit) / threads;
    for (int i = 1; i <= threads; i++)
        if (i == threads)
        {
            output[i - 1][0] = lowerLimit + intsPerSet * (i - 1);
            output[i - 1][1] = upperLimit;
        }
        else
        {
            output[i - 1][0] = lowerLimit + intsPerSet * (i - 1);
            output[i - 1][1] = lowerLimit + intsPerSet * i - 1;
        }
    return output;
}

void basicSieve(int upper, vector<int>& input)
{
    bool* primes = new bool[upper - 1];
    for (int i = 0; i <= upper - 2; i++)
        *(primes + i) = PRIME;

    for (int p = 2; p * p <= upper; p++) {
        if (*(primes + p - 2)) {
            for (int i = p * p; i <= upper; i += p)
                *(primes + i - 2) = false;
```

Kod 2.6

```
        }
    }

    for (int i = 0; i <= upper - 2; i++)
        if (*(primes + i)) {
            //cout << i + 2 << endl;
            input.push_back(i + 2);
        }
    delete primes;
}

void appendResults(vector<int>& toAppend, vector<int>& result)
{
    for (int i : toAppend)
        result.push_back(i);
}

vector<int> domain(int lower, int upper)
{
    vector<int> primes;
    int** subsets = subsets = divide(lower, upper);
    vector <int> primesToSqrt;
    vector<int> subOutput1, subOutput2, subOutput3, subOutput4, subOutput5,
subOutput6;

    start_t = omp_get_wtime();
    basicSieve(sqrt(upper), primesToSqrt);

#pragma omp  parallel
    {
        int threadNum = omp_get_thread_num();
        int lowerBound = subsets[threadNum][0], upperBound =
subsets[threadNum][1];
        int subsetRange = upperBound - lowerBound + 1;

        bool* subset = new bool[subsetRange];
        for (int i = 0; i < subsetRange; i++)
            *(subset + i) = PRIME;

        for (int i = 0; i < primesToSqrt.size(); i++)
        {
            int sieved = primesToSqrt[i];
            int number = lowerBound;
            for (; number % sieved != 0; number++)
                continue;
            if (number == sieved)
                number *= 2;

            for (; number <= upperBound; number += sieved) {
                subset[number - lowerBound] = COMPLEX;
            }
        }
        if (threadNum == 0)
        {
            for (int i = 0; i < subsetRange; i++)
                if (*(subset + i))
                    subOutput1.push_back(lowerBound + i);
        }
        else if (threadNum == 1)
        {
            for (int i = 0; i < subsetRange; i++)
                if (*(subset + i))
                    subOutput2.push_back(lowerBound + i);
```

```
        }
        else if (threadNum == 2)
        {
            for (int i = 0; i < subsetRange; i++)
                if (*(subset + i))
                    subOutput3.push_back(lowerBound + i);
        }
        else if (threadNum == 3)
        {
            for (int i = 0; i < subsetRange; i++)
                if (*(subset + i))
                    subOutput4.push_back(lowerBound + i);
        }
        else if (threadNum == 4)
        {
            for (int i = 0; i < subsetRange; i++)
                if (*(subset + i))
                    subOutput5.push_back(lowerBound + i);
        }
        else if (threadNum == 5)
        {
            for (int i = 0; i < subsetRange; i++)
                if (*(subset + i))
                    subOutput6.push_back(lowerBound + i);
        }

        delete subset;
    }

    stop_t = omp_get_wtime();

    appendResults(subOutput1, primes);
    appendResults(subOutput2, primes);
    appendResults(subOutput3, primes);
    appendResults(subOutput4, primes);
    appendResults(subOutput5, primes);
    appendResults(subOutput6, primes);


    cout << "Processing " << upper - lower << " took: " << stop_t - start_t << "
sec" << endl;

    return primes;
}

int main()
{
    vector<int> result = domain(2, B1);

}
```

Podejście w kodzie 2.6 (próba podejścia domenowego) również było nieoptymalne ze względu na przydział pamięci dla każdego z przedziałów. Problemem mógł się okazać również nierówny przydział pracy do wątków, przez co niektóre z nich mogły zakończyć działanie wcześniej, tym samym nie wnosząc nic do działania programu. Brak synchronizacji prowadził również do błędnych wyników działania programu.

Punkt 3: Prezentacja i omówienie wyników

Tabela 3.1

Wyniki algorytmów sekwencyjnych

instancja / parametr	**Dzielenie 2 .. 2E+9[1] [kod 2.1]**	**Dzielenie 1E+9 .. 2E+9[2] [kod 2.1]**	**Dzielenie 2 .. 1E+9[3] [kod 2.1]**	**Sito 2 .. 2E+9 [kod 2.2]**	**Sito 1E+9 .. 2E+9 [kod 2.2]**	**Sito 2 .. 1E+9 [kod 2.2]**
Elapsed time [s]	64,460	67,228	63,397	20,640	16,799	8,333
Instructions retired	9,5580**E**+10	1,293336**E**+11	1,708956**E**+11	2,79144**E**+10	2.57328**E**+10	1,52496**E**+10
Clockticks	8,84232**E**+10	1,180656**E**+11	1,553868**E**+11	7,71732**E**+10	6.83568**E**+10	3,31308**E**+10
Retiring [%]	61,4	57,5	53,8	8,0	8,1	9,9
Front-end Bound [%]	51,3	50,5	47,2	3,4	0,9	0,7
Back-end Bound [%]	0	0	0	87,8	90,5	89,0
Memory Bound [%]	0	0	0	43,5	44,7	44,1
Core Bound [%]	0	0	0	44,3	45,7	44,9
Wykorzystanie rdzeni fizycznych procesora [%]	5,5	16,4	9,8	15,1	16,2	44,9
Prędkość przetwarzania [1/s]	3,103**E**+7	1,487**E**+7	1,577**E**+7	9,690**E**+7	5,953**E**+7	1,200**E**+8

[1] Obliczenia przerwane po minucie
[2] Obliczenia przerwane po minucie
[3] Obliczenia przerwane po minucie

Wszystkie pomiary były wykonywane na 6 wątkach, ponieważ liczba rdzeni fizycznych była identyczna z liczbą wątków

Tabela 3.2

Wyniki algorytmów równoległych

instancja / parametr	**Sito funkcyjne 2 .. 2E+9 [kod 2.3]**	**Sito funkcyjne 1E+9 .. 2E+9 [kod 2.3]**	**Sito funkcyjne 2 .. 1E+9 [kod 2.3]**	**Sito domenowe 2 .. 2E+9 [kod 2.4]**	**Sito domenowe 1E+9 .. 2E+9 [kod 2.4]**	**Sito domenowe 2 .. 1E+9 [kod 2.4]**
Elapsed time [s]	8,592	4,328	4,115	8,400	4,276	3,863
Instructions retired	9,32976**E**+10	4,63248**E**+10	4,59648**E**+10	8,83764**E**+10	4.52340**E**+10	4,38084**E**+10
Clockticks	1,764972**E**+11	8,86608**E**+10	8,45388**E**+10	1,79658**E**+11	9.06768**E**+10	8,7174**E**+10
Retiring [%]	15.5	15,0	16,1	14,2	14,5	15,2
Front-end Bound [%]	1.2	1,4	1,2	1,6	5,6	1,5
Back-end Bound [%]	83.0	83,3	82,3	84,1	79,7	83,3
Memory Bound [%]	76.0	76,3	76,3	77,4	71,4	75,9
Core Bound [%]	7.0	5,8	6,0	6,7	8,3	7,3
Wykorzystanie rdzeni fizycznych procesora [%]	82.8	81,9	84,2	87,6	85,6	91,4
Prędkość przetwarzania [1/s]	1,0858**E**+10	1,07033**E**+10	1,11699**E**+10	1,0519**E**+10	1,0578**E**+10	1,1340**E**+10
Efektywność przetwarzania równoległego	0,40	0,64	0,34	0,41	0,65	0,36

Intel Vtune:

Oprogramowanie Intel Vtune Profiler wykorzsytuje próbkowanie sprzętowe, zbierając dane z liczników zdarzeń wbudowanych w procesor. Liczniki zbierają informację na temat ilości wykonanych instrukcji, taktowania lub pamięci podręcznej. Wydobycie danych odbywa się przez ustawienie progu na wartościach, a następnie wygenerowanie przerwania gdy próg zostanie osiągnięty. Vtune jest również w stanie dodawać specjalne instrukcje w kodzie, by zebrać bardziej szczegółowe dane odnośnie wykonania programu. Vtune ponadto analizuje jak wykonanie programu wpływa na system (zużycie pamięci, dysku) umożliwiając dogłębną analizę.

Oto wartości które były przez nas badane:

Elapsed time – czas, który upłynął od początku do końca działania programu, obejmuje wszystkie operacje wykonywane przez program.

Clockticks – jednostki czasu procesora reprezentujące liczbę cykli zegara procesora.

Instructions retired – liczba instrukcji procesora zakończonych (wykonanych) podczas analizowanego okresu. Innymi słowy jest to miara jak szybko program wykonuje poszczególne zadania i jak wykorzystuje zasoby.

Retiring – procent przedziałów alokacji, które zostały użyte i wykonane, tzn. nie zostały ograniczone przez back-end ani front-end, ani nie doszło do błędnej spekulacji

Front-end bound – wskaźnik jak często procesor czekał na dostarczenie instrukcji do wykonania

Back-end bound – wskaźnik jak często procesor czekał na zakończenie wykonywania instrukcji

Memory bound – wskaźnik jak często procesor czekał na dostęp do pamięci

Core bound – wskaźnik, który określa jak często procesor czekał na dostęp do zasobów rdzenia

Wykorzystanie rdzeni fizycznych procesora – w jakim stopniu rdzenie fizyczne były wykorzystywane podczas działania programu

Prędkość przetwarzania – ile liczb na sekundę zostało przetworzonych w trakcie działania

Efektywność przetwarzania równoległego – stosunek przyspieszenia działania programów równoległych, podzielonych na liczbę rdzeni fizycznych

Przetestowane kody równoległe charakteryzowały się wysokim ograniczeniem pamięciowym DRAM. Jest to spowodowane brakiem paddingu, który pomógłby efektywniej rozdzielać dane w pamięci podręcznej, przez co wątki nie unieważniałyby sobie pracy nawzajem (gdy wątek ubiega się o dostęp do pamięci, linie cache należy unieważnić). Pomimo wielu testów nie znaleźliśmy poprawnej metody/sposobu na efektywniejsze zarządzanie pamięcią. W sytuacji gdy pamięć była lepiej zarządzana, wkradały się błędy w wyznaczaniu lub szybkość działania była wyraźnie gorsza.

Punkt 4: Wnioski

Kod równoległy wypadł niewiele lepiej niż podejścia sekwencyjne. Czasowo mówimy o dwukrotnym przyspieszeniu, ale jeśli spojrzymy na efektywność przetwarzania równoległego, największe przyspieszenie na rdzeń wyniosło 0,65.

W wykonanym zadaniu procesor jest wykorzystywany w względnie wysokim stopniu ~85%, jednak brak doświadczenia w pisaniu efektywnego kodu do przetwarzania równoległego powoduje problemy z zarządzaniem pamięcią procesora, niskim współczynnikiem trafień do pamięci procesora, oraz wysokim wskaźnikiem odwołań do pamięci RAM.

Jednym z możliwych rozwiązań, które rozważaliśmy było zliczanie ilości liczb pierwszych w wersji kodu do przetestowania urządzeniem VTune, gdy użytkownik chciałby wyświetlić liczby byłyby one wyświetlane losowo, co związałoby się z mniejszą złożonością pamięciową, a to pozytywnie wpłynęłoby na szybkość działania kodu.

Mniejsza złożoność pamięciowa oznaczałaby również wyższy wskaźnik trafień do pamięci procesora. Głównym czynnikiem takiego efektu jest fakt, że nasze kody nieefektywnie zarządzają pamięcią, tak jak było to wspomniane w poprzednim punkcie. Lepsza implementacja rozwiązań takich jak padding pomogłaby osiągnąć lepsze wyniki przetwarzania, a także ograniczyć wykorzystanie zasobów. Najszybszym podejściem w naszym wypadku okazało się podejście domenowe (kod 2.4), osiągając prawie 2.5-krotne przyspieszenie w porównaniu do podejścia sekwencyjnego dla instancji 2 .. MAX (2..2E+9). Podejście domenowe osiągnęło również najwyższe zużycie procesora na poziomie 91,4%. W naszym przypadku większe zużycie procesora charakteryzowało się szybszym działaniem programu, jednakże mogłoby być ono mniejsze ze względu na ograniczenie pamięciowe, które było plagą naszej pracy. Taki efekt był spowodowany cache missami, ponieważ dane znajdowały się zbyt blisko siebie i regularnie były unieważniane przez inne wątki, przez co procesory musiały cały czas czekać na załadowanie danych do pamięci cache i dopiero wtedy wykonywać swoje obliczenia.

W podejściu funkcyjnym - może prowadzić do niewykorzystania pełnego potencjału procesora w sytuacjach, gdy niektóre wątki zakończą swoje zadania szybciej niż inne. W takim przypadku, niektóre rdzenie procesora mogą pozostać bezczynne, czekając na zakończenie pracy przez pozostałe wątki. Ograniczenia tego podejścia wynikają głównie z potencjalnego nierównomiernego obciążenia poszczególnych wątków. Możliwym rozwiązaniem jest dyrektywa schedule(dynamic), ale wprowadzamy dodatkową synchronizację z powodu dynamicznego przydziału pracy. Natomiast ograniczenia pamięciowe mogą wystąpić, gdy każdy z wątków potrzebuje dużych ilości danych do przechowywania informacji związanych z jego funkcją. Może to prowadzić do większego zużycia pamięci operacyjnej, zwłaszcza gdy liczba wątków jest duża.

W podejściu domenowe - W tym przypadku, poszczególne wątki mają równolegle przypisane różne zakresy wartości do przetworzenia. Jeśli długość tych zakresów jest znacznie różna, wątki z mniejszymi zakresami zakończą swoje zadania szybciej, co również może prowadzić do nierównomiernego obciążenia rdzeni procesora i czasu przestoju. Pamięciowo, w przypadku podejścia domenowego, każdy wątek pracuje na określonym zakresie danych, co może prowadzić do mniejszego zużycia pamięci w porównaniu z podejściem funkcyjnym. Niemniej jednak, w pewnych sytuacjach (np. gdy dane są złożone lub wymagają dużego buforowania), również może wystąpić duże zużycie pamięci. Ponadto, jeśli komunikacja między wątkami jest konieczna, może to również wpłynąć na zużycie pamięci.