Wojciech Bałtruszewicz, nr 145320, L15, wojciech.baltruszewicz@student.put.poznan.pl
Bartłomiej Kowalewski, nr 145204, L15, bartlomiej.p.kowalewski@student.put.poznan.pl
Środa 8:00, parzyste (pod kreską)

Przetwarzanie równoległe - laboratorium
Projekt 1 OMP

Pierwsza wersja sprawozdania

Wymagany termin sprawozdania: 4.05.2022r.

Rzeczywisty termin oddania sprawozdania: 4.05.2022r.

1 Opis realizowanego zadania

Realizowane zadanie polegało na analizie efektywności przetwarzania równoległego w komputerze z procesorem wielordzeniowym na przykładzie problemu znajdowania liczb pierwszych w określonym przedziale. Przetestowane zostały warianty algorytmu zarówno w wersji sekwencyjnej (metoda dzielenia oraz metoda Sita), jak i równoległej (podejście domenowe i funkcyjne w oparciu o metodę Sita).

2 Opis wykorzystywanego systemu obliczeniowego

2.1 Procesor

- Model: **Intel Core i7 4th Gen Haswell 4702MQ (2.20GHz - 3.20 GHz)**
- Liczba procesorów fizycznych: **4**
- Liczba procesorów logicznych: **8**
- Oznaczenie typu procesora: **MQ**
- Wielkość pamięci podręcznej: **6 MB**

- Organizacja pamięci podręcznej: **Intel® Smart Cache - architektura umożliwiająca wszystkim rdzeniom dynamiczne współdzielenie dostępu do pamięci podręcznej ostatniego poziomu**

2.2 Oprogramowanie

- System operacyjny: **Linux Mint 20.3 Una**
- Oprogramowanie wykorzystane do przygotowania kodu wynikowego: **Visual Studio Code**
- Oprogramowanie wykorzystane do przeprowadzania testów: **Intel VTune Profiler**

3 Wersje kodu

3.1 Sekwencyjne

3.1.1 Dzielenie liczb

Poniższy kod działa sekwencyjnie. Funkcja primeOrComplex sprawdza za pomocą dzielenia czy dana liczba jest liczbą pierwszą. Funkcja findPrimeNumbers odpowiada za stworzenie wektora który przechowuje wszytkie liczby pierwsze w danym zakresie.

```
#include <stdio.h>
#include <vector>
#include <math.h>
#include <algorithm>

#define COMPLEX 0;
#define PRIME 1;
using namespace std;

vector <int> primes = { 2 };

bool primeOrComplex(int number)
{
        if (find(primes.begin(), primes.end(), number) != primes.end())
                return PRIME;

        int upperLimit = floor( sqrt(number) );
        for (int i = 0; ; i++)
        {
                int divider = primes[i];
                if (upperLimit < divider)
                {
                        return PRIME;
                }
                else if (number % divider == 0)
                {
                        return COMPLEX;
                }
                else if (primes.size() - 1 == i)
                {
                        int nextPrimeNumber;
```

```
                        for (nextPrimeNumber = primes.back() + 1; !primeOrComplex(
                            nextPrimeNumber); nextPrimeNumber++)
                        {
                                continue;
                        }
                        primes.push_back(nextPrimeNumber);
                        divider = nextPrimeNumber;
                }
        }
}
vector <int> findPrimeNumbers(int lowerLimit, int upperLimit)
{
        vector <int> foundPrimes;
        for (int testedNumber = lowerLimit; testedNumber <= upperLimit; testedNumber++)
        {
                if (primeOrComplex(testedNumber))
                {
                        foundPrimes.push_back(testedNumber);
                }
        }
        return foundPrimes;
}
int main() {
        vector <int> tmp = findPrimeNumbers(100000000, 200000000);
        //for (int i = 0; i < tmp.size(); i++)
        //      printf("%d ", tmp[i]);
}
```

3.1.2 Metoda sita

Poniższy kod jest sekwencyjną realizacją sposobu wyznaczania liczb pierwszych za pomocą algorytmu Eratostenesa. Wcześniej wspomniany algorytm jest zaimplementowany w funkcji eratosthenesSieve(), która uzupełnia podany przez argument wektor o kolejno znalezione liczby pierwsze.

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#define PRIME 1
#define COMPLEX 0
using namespace std;
void printResultPrimes(vector <int> vec)
{
    for (int i = 0; i < vec.size(); i++)
    {
        printf("%d␣", vec[i]);
        if (i % 10 == 9)
            printf("\n");
    }
    printf("\nLiczba␣pierwszych:␣%ld\n", vec.size());
}
void eratosthenesSieve(int min, int max, vector<int> &primes) {
    int lastNum = (int)sqrt(max);
    vector<bool> isPrime;
    for (int i = 2; i <= max; i++)
        isPrime.push_back(PRIME);
    for (int divider = 2; divider <= lastNum; divider++)
    {
        if (isPrime[divider - 2] == COMPLEX)
            continue;
        for (int multiple = divider + divider; multiple <= max; multiple += divider)
            isPrime[multiple - 2] = COMPLEX;
    }
```

```
    for (int i = min - 2; i < isPrime.size(); i++)
    {
        if (isPrime[i] == PRIME)
            primes.push_back(i + 2);
    }
}
int main()
{
    vector<int> result;
    eratosthenesSieve(100000000, 200000000, result);
    //printResultPrimes(result);
}
```

3.2 Równoległe

3.2.1 Metoda sita - podejście domenowe

Poniższy kod przedstawia równoległą realizacje algorytmu sita Eratostenesa w wariancie domenowym. Funkcja initializeSubsets dzieli tablice wykreśleń na podzbiory które będą obsługiwać poszczególne wątki. Funkcja findStartingPrimes() wyznacza liczby pierwsze od dolnej granicy zakresu aż do pierwiastka kwadratowego z górnej granicy. W funkcji find-PrimesDomain wyznaczane są liczby pierwsze za pomocą podejścia domenowego, każdy wątek znajduje liczby pierwsze - "wykreśla" liczby w danym zakresie korzystając z tablicy liczb pierwszych. Na samym końcu wyniki z poszczególnych wątków są scalane. Dyrektywa pragma omp parallel num_threads(threadsNum) tworzy zespół wątków o określonej liczbie i rozpoczyna równolegle działający fragment kodu.

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <cmath>
#include <omp.h>
#define threadsNum 4
#define PRIME 1
#define COMPLEX 0
using namespace std;

void printResultPrimes(vector <int> primes)
{
    for (int i = 0; i < primes.size(); i++)
    {
        printf("%d ", primes[i]);
        if (i % 10 == 9)
            printf("\n");
    }
    printf("\nLiczba pierwszych: %ld\n", primes.size());
}
vector<vector<int>> initializeSubsets(int lowerLimit, int upperLimit, int subsetsNumber)
{
    int range = (upperLimit - lowerLimit) / subsetsNumber;
    vector <vector<int>> subsets;
    vector <int> subset;
    int nextNumber = lowerLimit;
    for (int i = 0; i < subsetsNumber - 1; i++)
    {
        subset = { nextNumber, nextNumber + range - 1};
```

```
        subsets.push_back(subset);
        nextNumber = nextNumber + range;
    }
    subset = { nextNumber, upperLimit };
    subsets.push_back(subset);

    return subsets;
}

void findStartingPrimes(int min, int max, vector<int> &startingPrimes) {

    int lastNum = (int)sqrt(max);
    vector<bool> isPrime;
    for (int i = 2; i <= max; i++)
        isPrime.push_back(PRIME);

    for (int divider = 2; divider <= lastNum; divider++)
    {
        if (isPrime[divider - 2] == COMPLEX)
            continue;

        for (int multiple = divider + divider; multiple <= max; multiple += divider)
            isPrime[multiple - 2] = COMPLEX;
    }

    for (int i = min - 2; i < isPrime.size(); i++)
    {
        if (isPrime[i] == PRIME)
            startingPrimes.push_back(i + 2);
    }
}
void findPrimesDomain(int minNum, int maxNum, vector<int> &primes)
{
    vector < vector <int> > subsets = initializeSubsets(minNum, maxNum, threadsNum);
    int lastNum = (int)sqrt(maxNum);
    vector <int> startingPrimes;
    findStartingPrimes(2, lastNum, startingPrimes);

    vector <bool> subset0;
    vector <bool> subset1;
    vector <bool> subset2;
    vector <bool> subset3;
    vector <bool> subset4;
    vector <bool> subset5;
    vector <bool> subset6;
    vector <bool> subset7;

    #pragma omp parallel num_threads(threadsNum)
    {
        int threadNumber = omp_get_thread_num();
        vector<int> privateSubset = subsets[threadNumber];
        int lowerSubsetLimit = privateSubset[0];
        int upperSubsetLimit = privateSubset[1];
        int subsetRange = upperSubsetLimit - lowerSubsetLimit + 1;

        vector<bool> subset(subsetRange, PRIME);

        for (int i = 0; i < startingPrimes.size(); i++)
        {
            int divider = startingPrimes[i];
            int multiple = lowerSubsetLimit;
            for (; multiple % divider != 0; multiple++)
                continue;
            if (multiple == divider)
                multiple = divider + divider;

            for (; multiple <= upperSubsetLimit; multiple += divider)
                subset[multiple - lowerSubsetLimit] = COMPLEX;
        }

        switch (threadNumber)
        {
        case 0:
            subset0 = subset;
            break;
        case 1:
            subset1 = subset;
            break;
        case 2:
            subset2 = subset;
            break;
        case 3:
            subset3 = subset;
            break;
        case 4:
            subset4 = subset;
```

```
            break;
        case 5:
            subset5 = subset;
            break;
        case 6:
            subset6 = subset;
            break;
        case 7:
            subset7 = subset;
            break;
        }
    }
    vector<bool> isPrime;
    isPrime.reserve(maxNum - minNum);

    if (threadsNum == 0)
        goto isPrimeCreated;
    isPrime.insert(isPrime.end(), subset0.begin(), subset0.end());

    if (threadsNum == 1)
        goto isPrimeCreated;
    isPrime.insert(isPrime.end(), subset1.begin(), subset1.end());

    if (threadsNum == 2)
        goto isPrimeCreated;
    isPrime.insert(isPrime.end(), subset2.begin(), subset2.end());

    if (threadsNum == 3)
        goto isPrimeCreated;
    isPrime.insert(isPrime.end(), subset3.begin(), subset3.end());

    if (threadsNum == 4)
        goto isPrimeCreated;
    isPrime.insert(isPrime.end(), subset4.begin(), subset4.end());

    if (threadsNum == 5)
        goto isPrimeCreated;
    isPrime.insert(isPrime.end(), subset5.begin(), subset5.end());

    if (threadsNum == 6)
        goto isPrimeCreated;
    isPrime.insert(isPrime.end(), subset6.begin(), subset6.end());

    if (threadsNum == 7)
        goto isPrimeCreated;
    isPrime.insert(isPrime.end(), subset7.begin(), subset7.end());

isPrimeCreated:
    for (int i = minNum - 2; i < isPrime.size(); i++)
    {
        if (isPrime[i] == PRIME)
            primes.push_back(i + 2);
    }
}
int main()
{
    vector<int> result;
    findPrimesDomain
    (2, 200000000, result);
    //printResultPrimes(result);
}
```

3.2.2 Metoda sita - podejście funkcyjne

Poniższy program równolegle realizuje znajdowanie liczb pierwszych za pomocą algorytmu sita Eratostenesa w wersji funkcyjnej. Funkcja findStartingPrimes() wyznacza liczby pierwsze od dolnej granicy zakresu aż do pierwiastka kwadratowego z górnej granicy. Funkcja findPrimesFunctional() realizuje podejście funkcyjne: każdy wątek korzysta z całej tablicy wykreśleń oraz operuje na podzbiorze wcześniej wyznaczonych liczb pierwszych. Po wykreśleniu liczb w ten sposób wyniki poszczególnych wątków są scalane oraz zwracany jest wynikowy wektor zawierający wszystkie liczby pierwsze w danym przedziale. Dyrektywa pragma omp parallel num_threads(threadsNum) tworzy zespół wątków o określonej liczbie i rozpoczyna równolegle działający fragment kodu. Dyrektywa pragma omp for schedule(dynamic) dzieli dynamicznie iteracje pętli for. W tym przypadku służy to do podzielenia początkowego zbioru liczb pierwszych na poszczególne wątki.

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

#define threadsNum 4

#define PRIME 1
#define COMPLEX 0

using namespace std;

void printPrimes(vector <int> primes)
{
    for (int i = 0; i < primes.size(); i++)
    {
        printf("%d ", primes[i]);
        if (i % 10 == 9)
            printf("\n");
    }
    printf("\nLiczba pierwszych: %ld\n", primes.size());
}

vector<int> findStartingPrimes(int min, int max) {

    int lastNum = (int)sqrt(max);
    vector<bool> isPrime;
    for (int i = 2; i <= max; i++)
        isPrime.push_back(PRIME);

    for (int divider = 2; divider <= lastNum; divider++)
    {
        if (isPrime[divider - 2] == COMPLEX)
            continue;

        for (int multiple = divider + divider; multiple <= max; multiple += divider)
            isPrime[multiple - 2] = COMPLEX;
    }

    vector<int> startingPrimes;
    for (int i = min- 2; i < isPrime.size(); i++)
    {
        if (isPrime[i] == PRIME)
            startingPrimes.push_back(i + 2);
    }

    return startingPrimes;

}
void findPrimesFunctional(int minNum, int maxNum, vector<int> &primes)
{
    int lastNum = (int)sqrt(maxNum);
    int range = (maxNum - minNum) + 1;
    vector <int> startingPrimes;
    startingPrimes = findStartingPrimes(2, lastNum);

    vector <bool> isPrime0;
    vector <bool> isPrime1;
    vector <bool> isPrime2;
    vector <bool> isPrime3;
    vector <bool> isPrime4;
    vector <bool> isPrime5;
    vector <bool> isPrime6;
    vector <bool> isPrime7;

    #pragma omp parallel num_threads(threadsNum)
    {
```

```
    int threadNumber = omp_get_thread_num();
    vector<bool> localIsPrime(range, PRIME);

    #pragma omp     for schedule(dynamic)
    for (int i = 0; i < startingPrimes.size(); i++)
    {
        int divider = startingPrimes[i];
        int multiple = minNum;
        for (; multiple % divider != 0; multiple++)
            continue;
        if (multiple == divider)
            multiple = divider + divider;

        for (; multiple <= maxNum; multiple += divider)
            localIsPrime[multiple - minNum] = COMPLEX;
    }

    switch (threadNumber)
    {
    case 0:
        isPrime0 = localIsPrime;
        break;
    case 1:
        isPrime1 = localIsPrime;
        break;
    case 2:
        isPrime2 = localIsPrime;
        break;
    case 3:
        isPrime3 = localIsPrime;
        break;
    case 4:
        isPrime4 = localIsPrime;
        break;
    case 5:
        isPrime5 = localIsPrime;
        break;
    case 6:
        isPrime6 = localIsPrime;
        break;
    case 7:
        isPrime7 = localIsPrime;
        break;
    }
}

vector <bool> isPrime;
switch (threadsNum)
{
case 1:
    for (int i = 0; i < range; i++)
        isPrime.push_back(isPrime0[i]);
    break;
case 2:
    for (int i = 0; i < range; i++)
        isPrime.push_back(
            isPrime0[i] * isPrime1[i]);
    break;
case 3:
    for (int i = 0; i < range; i++)
        isPrime.push_back(
            isPrime0[i] * isPrime1[i] * isPrime2[i]);
    break;
case 4:
    for (int i = 0; i < range; i++)
        isPrime.push_back(
            isPrime0[i] * isPrime1[i] * isPrime2[i] * isPrime3[i]);
    break;
case 5:
    for (int i = 0; i < range; i++)
        isPrime.push_back(
            isPrime0[i] * isPrime1[i] * isPrime2[i] * isPrime3[i] * isPrime4[i]);
    break;
case 6:
    for (int i = 0; i < range; i++)
        isPrime.push_back(
            isPrime0[i] * isPrime1[i] * isPrime2[i] * isPrime3[i] * isPrime4[i] *
                isPrime5[i]);
    break;
case 7:
    for (int i = 0; i < range; i++)
        isPrime.push_back(
            isPrime0[i] * isPrime1[i] * isPrime2[i] * isPrime3[i] * isPrime4[i] *
                isPrime5[i] * isPrime6[i]);
    break;
```

```
    case 8:
        for (int i = 0; i < range; i++)
            isPrime.push_back(
                isPrime0[i] * isPrime1[i] * isPrime2[i] * isPrime3[i] * isPrime4[i] *
                    isPrime5[i] * isPrime6[i] * isPrime7[i]);
        break;
    }
    for (int i = minNum - 2; i < isPrime.size(); i++)
    {
        if (isPrime[i] == PRIME)
            primes.push_back(i + 2);
    }
}

int main()
{
    vector<int> result;
    findPrimesFunctional(2, 200000000, result);
    //printPrimes(result);
}
```

3.3 Wcześniejsze wersje kodu

Początkowo staraliśmy się używać tylko statycznych struktur danych lecz niestety próby te zakończyły się niepowodzeniem. Nie byliśmy w stanie przeprowadzić odpowiednio długo trwających testów a przy zwiększaniu wielkości tablic zaczęły występować problemy z pamięcią, więc odstąpiliśmy od tego założenia. Poniżej przedstawiamy wersje powyższych rozwiązań problemów przy użyciu statycznych struktur.

3.3.1 Dzielenie sekwencyjne

```
#include <stdio.h>
#include <vector>
#include <math.h>
#include <algorithm>

#define NUMBER 10000000;
#define NOTPRIME 0;
#define PRIME 1;

int primeNumbers[NUMBER] = { 2 };
int lastAllocatedCellPrimeNumbers = 0;

bool isPrime(int number)
{
        for (int i = 0; i <= lastAllocatedCellPrimeNumbers; i++)
        {
                if (number == primeNumbers[i])
                return PRIME;
        }

        int upperLimit = floor( sqrt(number) );
        for (int i = 0; ; i++)
        {
                int divider = primeNumbers[i];
                if (upperLimit < divider)
                {
                        return PRIME;
                }
                else if (number % divider == 0)
                {
                        return NOTPRIME;
                }
                else if (lastAllocatedCellPrimeNumbers == i)
                {
```

```
                        int nextPrimeNumber;
                        for (nextPrimeNumber = primeNumbers[lastAllocatedCellPrimeNumbers] +
                             1; !isPrime(nextPrimeNumber); nextPrimeNumber++)
                        {
                                continue;
                        }
                        lastAllocatedCellPrimeNumbers++;
                        primeNumbers[lastAllocatedCellPrimeNumbers] = nextPrimeNumber;
                        divider = nextPrimeNumber;
                }
        }
}
void findPrimeNumbers(int lowerLimit, int upperLimit, int *resultArray, int &
    lastAllocatedIndexOfResult)
{
        for (int testedNumber = lowerLimit; testedNumber <= upperLimit; testedNumber++)
        {
                if (isPrime(testedNumber))
                        resultArray[lastAllocatedIndexOfResult++] = testedNumber;
        }
}
int main() {
        int resultPrime[NUMBER];
        int allocatedCellsArray = 0;
        findPrimeNumbers(2, NUMBER, resultPrime, allocatedCellsArray);
        for (int i = 0; i < allocatedCellsArray; i++)
                printf("%d␣", resultPrime[i]);
}
```

3.3.2 Metoda sita

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>

#define NUMBER 1000;
#define PRIME 1
#define COMPLEX 0

int lastAllocatedCellResult = 0;

void printPrimes(int *primes)
{
    for (int i = 0; i < lastAllocatedCellResult; i++)
    {
        printf("%d␣", primes[i]);
        if (i % 10 == 9)
            printf("\n");
    }
    printf("\nprime␣numbers␣count:␣%d\n", lastAllocatedCellResult+1);
}
void eratosthenesSieve(int minNum, int maxNum,int * primes) {

    int lastNum = (int)sqrt(maxNum);
    int primeOrComplex[NUMBER];
    int lastAllocatedPrimeOrComplex = 0;
    for (int i = 2; i <= maxNum; i++)
        primeOrComplex[lastAllocatedPrimeOrComplex++] = PRIME;

    for (int divider = 2; divider <= lastNum; divider++)
    {
        if (primeOrComplex[divider - 2] == COMPLEX)
            continue;

        for (int multiple = divider + divider; multiple <= maxNum; multiple += divider)
            primeOrComplex[multiple - 2] = COMPLEX;
    }

    for (int i = minNum - 2; i < lastAllocatedPrimeOrComplex; i++)
    {
        if (primeOrComplex[i] == PRIME)
            primes[lastAllocatedCellResult++] = i+2;
    }

}

int main()
```

```
{
    int result[NUMBER];
    eratosthenesSieve(2, 200, result);
    printPrimes(result);
}
```

3.3.3 Metoda sita podejście Domenowe

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <cmath>
#include <omp.h>
#include <algorithm>

#define threadsNum 7
#define NUMBER 1000;
#define PRIME 1
#define COMPLEX 0

int lastAllocatedCellResult = 0;

void printPrimes(int *primes)
{
    for (int i = 0; i < lastAllocatedCellResult; i++)
    {
        printf("%d␣", primes[i]);
        if (i % 10 == 9)
            printf("\n");
    }
    printf("\nprime␣numbers␣count:␣%d\n", lastAllocatedCellResult+1);
}

void createSubsets(int lowerLimit, int upperLimit, int subsetsNumber, int (* subsets)[2])
{
    int range = (upperLimit - lowerLimit) / subsetsNumber;
    int lastIndex =0;
    int nextNumber = lowerLimit;
    for (int i = 0; i < subsetsNumber - 1; i++)
    {
        subsets[i][0] = nextNumber;
        subsets[i][1] = nextNumber + range -1;
        lastIndex=i;
        nextNumber = nextNumber + range;
    }
    lastIndex++;
    subsets[lastIndex][0] = nextNumber;
    subsets[lastIndex][1] = upperLimit;
}

void createStartingPrimes(int minNum, int maxNum, int* startingPrimes, int &
    lastAllocatedIndex) {
    int lastNum = (int)sqrt(maxNum);
    bool primeOrComplex[NUMBER];
    for (int i = 2; i <= maxNum; i++)
        primeOrComplex[i-2] = PRIME;

    for (int divider = 2; divider <= lastNum; divider++)
    {
        if (primeOrComplex[divider - 2] == COMPLEX)
            continue;

        for (int multiple = divider + divider; multiple <= maxNum; multiple += divider)
            primeOrComplex[multiple - 2] = COMPLEX;
    }

    for (int i = minNum - 2; i < maxNum; i++)
    {
        if (primeOrComplex[i] == PRIME)
            startingPrimes[lastAllocatedIndex++] = i + 2;
    }
}

void parallelDomain(int minNum, int maxNum, int *primes)
{
    int subsets[10][2];
    createSubsets(minNum, maxNum, threadsNum, subsets);
    int lastNum = (int)sqrt(maxNum);
    int startingPrimes[NUMBER];
    int lastAllocatedStartingPrimes  = 0;
    createStartingPrimes(2, lastNum, startingPrimes, lastAllocatedStartingPrimes);
```

```
bool subset0[NUMBER];
bool subset1[NUMBER];
bool subset2[NUMBER];
bool subset3[NUMBER];
bool subset4[NUMBER];
bool subset5[NUMBER];
bool subset6[NUMBER];
bool subset7[NUMBER];
int subsetRange0;
int subsetRange1;
int subsetRange2;
int subsetRange3;
int subsetRange4;
int subsetRange5;
int subsetRange6;
int subsetRange7;

#pragma omp parallel num_threads(threadsNum)
{
    int threadNumber = omp_get_thread_num();
    int threadSubset[] = {subsets[threadNumber][0], subsets[threadNumber][1]};
    int lowerSubsetLimit = threadSubset[0];
    int upperSubsetLimit = threadSubset[1];
    int subsetRange = upperSubsetLimit - lowerSubsetLimit + 1;
    bool subset[NUMBER];
    std::fill_n(subset,subsetRange, PRIME);
    for (int i = 0; i < lastAllocatedStartingPrimes; i++)
    {
        int divider = startingPrimes[i];
        int multiple = lowerSubsetLimit;
        for (; multiple % divider != 0; multiple++)
            continue;
        if (multiple == divider)
            multiple = divider + divider;
        for (; multiple <= upperSubsetLimit; multiple += divider)
            subset[multiple - lowerSubsetLimit] = COMPLEX;
    }
    switch (threadNumber)
    {
    case 0:
        std::copy(std::begin(subset),std::end(subset), std::begin(subset0));
        subsetRange0 = subsetRange ;
        break;
    case 1:
        std::copy(std::begin(subset),std::end(subset), std::begin(subset1));
        subsetRange1 = subsetRange ;
        break;
    case 2:
        std::copy(std::begin(subset),std::end(subset), std::begin(subset2));
        subsetRange2 = subsetRange ;
        break;
    case 3:
        std::copy(std::begin(subset),std::end(subset), std::begin(subset3));
        subsetRange3 = subsetRange ;
        break;
    case 4:
        std::copy(std::begin(subset),std::end(subset), std::begin(subset4));
        subsetRange4 = subsetRange ;
        break;
    case 5:
        std::copy(std::begin(subset),std::end(subset), std::begin(subset5));
        subsetRange5 = subsetRange ;
        break;
    case 6:
        std::copy(std::begin(subset),std::end(subset), std::begin(subset6));
        subsetRange6 = subsetRange ;
        break;
    case 7:
        std::copy(std::begin(subset),std::end(subset), std::begin(subset7));
        subsetRange7 = subsetRange ;
        break;
    }
}
int lastAllocatedPrimeOrComplex = 0;
bool primeOrComplex[NUMBER];
for (int i = 0; i < subsetRange0 ; i++)
{
```

```
        primeOrComplex[lastAllocatedPrimeOrComplex++] = subset0[i];
    }
    if (threadsNum == 0)
        goto primeOrComplexCreated;

    for (int i = 0; i < subsetRange1 ; i++)
    {
        primeOrComplex[lastAllocatedPrimeOrComplex++] = subset1[i];
    }
    if (threadsNum == 1)
        goto primeOrComplexCreated;

    for (int i = 0; i < subsetRange2 ; i++)
    {
        primeOrComplex[lastAllocatedPrimeOrComplex++] = subset2[i];
    }
    if (threadsNum == 2)
        goto primeOrComplexCreated;

    for (int i = 0; i < subsetRange3 ; i++)
    {
        primeOrComplex[lastAllocatedPrimeOrComplex++] = subset3[i];
    }
    if (threadsNum == 3)
        goto primeOrComplexCreated;

    for (int i = 0; i < subsetRange4 ; i++)
    {
        primeOrComplex[lastAllocatedPrimeOrComplex++] = subset4[i];
    }
    if (threadsNum == 4)
        goto primeOrComplexCreated;

    for (int i = 0; i < subsetRange5 ; i++)
    {
        primeOrComplex[lastAllocatedPrimeOrComplex++] = subset5[i];
    }
    if (threadsNum == 5)
        goto primeOrComplexCreated;

    for (int i = 0; i < subsetRange6 ; i++)
    {
        primeOrComplex[lastAllocatedPrimeOrComplex++] = subset6[i];
    }
    if (threadsNum == 6)
        goto primeOrComplexCreated;

    for (int i = 0; i < subsetRange7 ; i++)
    {
        primeOrComplex[lastAllocatedPrimeOrComplex++] = subset7[i];
    }
    if (threadsNum == 7)
        goto primeOrComplexCreated;

primeOrComplexCreated:
    for (int i = minNum - 2; i < lastAllocatedPrimeOrComplex; i++)
    {
        if (primeOrComplex[i] == PRIME)
            primes[lastAllocatedCellResult++]= i + 2;
    }
}

int main()
{
    int result[NUMBER];
    parallelDomain(2,NUMBER,result);
    printPrimes(result);
}
```

3.3.4 Metoda sita podejście funkcyjne

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <cmath>
#include <omp.h>
#include <algorithm>

#define threadsNum 8
#define NUMBER 1000
```

```
#define PRIME 1
#define COMPLEX 0

int lastAllocatedCellResult = 0;

void printPrimes(int *primes)
{
    for (int i = 0; i < lastAllocatedCellResult; i++)
    {
        printf("%d␣", primes[i]);
        if (i % 10 == 9)
            printf("\n");
    }
    printf("\nprime␣numbers␣count:␣%d\n", lastAllocatedCellResult+1);
}

void createStartingPrimes(int minNum, int maxNum, int* startingPrimes, int &
    lastAllocatedIndex) {

    int lastNum = (int)sqrt(maxNum);
    bool primeOrComplex[NUMBER];
    for (int i = 2; i <= maxNum; i++)
        primeOrComplex[i-2] = PRIME;

    for (int divider = 2; divider <= lastNum; divider++)
    {
        if (primeOrComplex[divider - 2] == COMPLEX)
            continue;

        for (int multiple = divider + divider; multiple <= maxNum; multiple += divider)
            primeOrComplex[multiple - 2] = COMPLEX;
    }

    for (int i = minNum - 2; i < maxNum; i++)
    {
        if (primeOrComplex[i] == PRIME)
            startingPrimes[lastAllocatedIndex++] = i + 2;
    }
}

void parallelFunctional(int minNum, int maxNum, int *primes)
{
    int lastNum = (int)sqrt(maxNum);
    const int range = (maxNum - minNum) + 1;

    int startingPrimes[NUMBER];
    int lastAllocatedStartingPrimes  = 0;
    createStartingPrimes(2, lastNum, startingPrimes, lastAllocatedStartingPrimes);

    bool primeOrComplex0[NUMBER];
    bool primeOrComplex1[NUMBER];
    bool primeOrComplex2[NUMBER];
    bool primeOrComplex3[NUMBER];
    bool primeOrComplex4[NUMBER];
    bool primeOrComplex5[NUMBER];
    bool primeOrComplex6[NUMBER];
    bool primeOrComplex7[NUMBER];

    #pragma omp parallel num_threads(threadsNum)
    {
        int threadNumber = omp_get_thread_num();
        bool localPrimeOrComplex[NUMBER];
        std::fill_n(localPrimeOrComplex,range, PRIME);

        #pragma omp     for schedule(dynamic, 10)
        for (int i = 0; i < lastAllocatedStartingPrimes; i++)
        {
            int divider = startingPrimes[i];
            int multiple = minNum;
            for (; multiple % divider != 0; multiple++)
                continue;
            if (multiple == divider)
                multiple = divider + divider;

            for (; multiple <= maxNum; multiple += divider)
                localPrimeOrComplex[multiple - minNum] = COMPLEX;
        }

        switch (threadNumber)
        {
        case 0:
            std::copy(std::begin(localPrimeOrComplex),std::end(localPrimeOrComplex), std::
                begin(primeOrComplex0));
            break;
        case 1:
```

```
        std::copy(std::begin(localPrimeOrComplex),std::end(localPrimeOrComplex), std::
            begin(primeOrComplex1));
        break;
    case 2:
        std::copy(std::begin(localPrimeOrComplex),std::end(localPrimeOrComplex), std::
            begin(primeOrComplex2));
        break;
    case 3:
        std::copy(std::begin(localPrimeOrComplex),std::end(localPrimeOrComplex), std::
            begin(primeOrComplex3));
        break;
    case 4:
        std::copy(std::begin(localPrimeOrComplex),std::end(localPrimeOrComplex), std::
            begin(primeOrComplex4));
        break;
    case 5:
        std::copy(std::begin(localPrimeOrComplex),std::end(localPrimeOrComplex), std::
            begin(primeOrComplex5));
        break;
    case 6:
        std::copy(std::begin(localPrimeOrComplex),std::end(localPrimeOrComplex), std::
            begin(primeOrComplex6));
        break;
    case 7:
        std::copy(std::begin(localPrimeOrComplex),std::end(localPrimeOrComplex), std::
            begin(primeOrComplex7));
        break;
    }
}
bool primeOrComplex[NUMBER];
switch (threadsNum)
{
case 1:
    for (int i = 0; i < range; i++)
        primeOrComplex[i] = primeOrComplex0[i];
    break;
case 2:
    for (int i = 0; i < range; i++)
        primeOrComplex[i] =
            primeOrComplex0[i] *
            primeOrComplex1[i];
    break;
case 3:
    for (int i = 0; i < range; i++)
        primeOrComplex[i] =
            primeOrComplex0[i] *
            primeOrComplex1[i] *
            primeOrComplex2[i];
    break;
case 4:
    for (int i = 0; i < range; i++)
        primeOrComplex[i] =
            primeOrComplex0[i] *
            primeOrComplex1[i] *
            primeOrComplex2[i] *
            primeOrComplex3[i];
    break;
case 5:
    for (int i = 0; i < range; i++)
        primeOrComplex[i] =
            primeOrComplex0[i] *
            primeOrComplex1[i] *
            primeOrComplex2[i] *
            primeOrComplex3[i] *
            primeOrComplex4[i];
    break;
case 6:
    for (int i = 0; i < range; i++)
        primeOrComplex[i] =
            primeOrComplex0[i] *
            primeOrComplex1[i] *
            primeOrComplex2[i] *
            primeOrComplex3[i] *
            primeOrComplex4[i] *
            primeOrComplex5[i];
    break;
case 7:
    for (int i = 0; i < range; i++)
        primeOrComplex[i] =
            primeOrComplex0[i] *
```

```
                primeOrComplex1[i] *
                primeOrComplex2[i] *
                primeOrComplex3[i] *
                primeOrComplex4[i] *
                primeOrComplex5[i] *
                primeOrComplex6[i];
        break;
    case 8:
        for (int i = 0; i < range; i++)
            primeOrComplex[i] =
                primeOrComplex0[i] *
                primeOrComplex1[i] *
                primeOrComplex2[i] *
                primeOrComplex3[i] *
                primeOrComplex4[i] *
                primeOrComplex5[i] *
                primeOrComplex6[i] *
                primeOrComplex7[i];
        break;
    }

    for (int i = minNum - 2; i < range; i++)
    {
        if (primeOrComplex[i] == PRIME)
            primes[lastAllocatedCellResult++]= i + 2;
    }
}
int main()
{
    int result[NUMBER];
    parallelFunctional(2, NUMBER, result);
    printPrimes(result);
}
```

4 Prezentacja wyników

W celu wykonania eksperymentu obliczeniowo-pomiarowego skorzystaliśmy z dostępnego w programie Intel VTune trybu "Microarchitecture Exploration", który pozwala na analizę efektywności przetwarzania. W trybie tym dane zbierane są podczas pracy procesora za pomocą jednostek monitorujących wydajność, które zawierają liczniki wystąpienia różnych zdarzeń procesora.
"Microarchitecture Exploration" uruchomiliśmy zarówno dla metod sekwencyjnych, jak i wariantów zrównoleglenia. Programy sekwencyjne uruchamialiśmy dla jednego procesora. Natomiast programy równoległe uruchamiane zostały dla maksymalnej liczby dostępnych w systemie procesorów logicznych tj. 8 oraz dla maksymalnej liczby procesorów fizycznych tj. 4.
Wielkości zastosowanych przez nas w przetwarzaniu instacji to:

- 2 - 200000000 liczb (2...MAX)
- 2 - 100000000 liczb (2...MAX/2)

- 100000000 - 200000000 liczb (MAX/2 - MAX)

Wartości poszczególnych parametrów przetwarzania zapisaliśmy w tabelach (jedna dla metod sekwencyjnych, jedna dla koncepcji domenowej i i jedna dla koncepcji funkcyjnej). Postanowiliśmy stworzyć trzy tabelę, gdyż niemożliwe byłoby zmieszczenie na szerokości strony wszystkich wartości w jednej lub dwóch tabelach w zastosowanym przez nas układzie.

4.1 Przetwarzanie sekwencyjne

W poniższej tabeli znajdują się wyniki przetwarzania dla metod sekwencyjnych.

Tabela 1: Tabela wartości parametrów przetwarzania dla metod sekwencyjnych

Instancja testowa / Parametr	**Dzielenie sekwencyjne 2-200000000**	**Dzielenie sekwencyjne 2-100000000**	**Dzielenie sekwencyjne 100000000-200000000**	**Sito sekwencyjnie 2-200000000**	**Sito sekwencyjnie 2-100000000**	**Sito sekwencyjnie 100000000-200000000**
Elapsed time [s]	163,435	63,037	100,455	3,017	1,384	2,606
Instructions retired	927,942E+09	349,914E+09	578,041E+09	12,260E+09	6,237E+09	10,612E+09
Clockticks	354,222E+09	136,569E+09	217,641E+09	6,382E+09	2,989E+09	5,623E+09
Retiring [%]	52,2	52,0	52,4	63,9	80,6	63,8
Front-end bound [%]	36,2	38,2	38,9	6,1	4,4	5,1
Back-end bound [%]	10,2	8,1	7,4	26,3	11,5	29,2
Memory bound [%]	2,1	1,8	1,6	14,6	5,2	19,7
Core bound [%]	8,0	6,3	5,8	11,7	6,3	9,5
Effective physical core [%]	24,6	24,6	24,5	24,0	24,1	24,5
Prędkość przetwarzania	1,22373E+06	1,58637E+06	0,995470E+06	66,291E+06	72,254E+06	38,372E+06

Z tabeli powyżej wynika, że dużo bardziej efektywną metodą jest metoda Sita, gdyż osiąga ona prędkość przetwarzania nawet 54 razy większą w przypadku instancji o zakresie 2...MAX, niż metoda oparta o dzielenie. Analizując tabelę możemy również zauważyć, że każdy program prawie w pełni wykorzystuje jeden dostępny wątek. Najkrótszy czas przetwarzania osiągnęła metoda sita w instancji o zakresie 2...MAX/2.

4.2 Przetwarzanie równoległe

Poniżej znajdują się dwie tabele z wartościami parametrów przetwarzania zarówno dla podejścia domenowego, jak i funkcyjnego.

4.2.1 Podejście domenowe

Tabela 2: Tabela wartości parametrów przetwarzania dla wariantu domenowego

Instancja testowa / Parametr	**Sito (8 wątków) 2-200000000**	**Sito (8 wątków) 2-100000000**	**Sito (8 wątków) 100000000-200000000**	**Sito (4 wątki) 2-200000000**	**Sito (4 wątki) 2-100000000**	**Sito (4 wątki) 100000000-200000000**
Elapsed time [s]	2,375	1,032	1,951	2,476	1,050	1,984
Instructions retired	12,342E+09	5,931E+09	9,186E+09	12,117E+09	6,124E+09	9,354E+09
Clockticks	14,214E+09	5,003E+09	8,642E+09	9,933E+09	3,731E+09	7,727E+09
Retiring [%]	59,4	32,4	43,6	47,3	88,2	72,8
Front-end bound [%]	11,8	11,9	12,2	2,1	3,2	2,6
Back-end bound [%]	24,9	51,8	44,2	50,1	2,8	18,7
Memory bound [%]	23,3	23,8	23,5	42,9	1,6	13,7
Core bound [%]	1,6	28,0	20,7	7,2	1,2	5,0
Effective physical core [%]	42,6	41.5	42,1	39,6	36,9	37,4
Przyspieszenie przetwarzania	1,270	1.341	1,336	1,218	1,318	1,313
Prędkość przetwarzania [liczb/s]	84,211E+06	96,899E+06	57,110E+06	80,775E+06	95,238E+06	50,403E+06
Efektywność przetwarzania	0,745	0.808	0,793	0,769	0,892	0,878

Analizując tabelę wartości parametrów przetwarzania dla wariantu domenowego możemy zauważyć, że zdecydowanie najkrótszy czas osiągnięto dla przydzielonych 8 wątków dla instancji 2...MAX/2. W tym przypadku osiągnięto najwyższą prędkość przetwarzania wynoszącą około 96,899E+06 liczb/s.

4.2.2 Podejście funkcyjne

Tabela 3: Tabela wartości parametrów przetwarzania dla wariantu funkcyjnego

Instancja testowa / Parametr	**Sito (8 wątków) 2-200000000**	**Sito (8 wątków) 2-100000000**	**Sito (8 wątków) 100000000-200000000**	**Sito (4 wątki) 2-200000000**	**Sito (4 wątki) 2-100000000**	**Sito (4 wątki) 100000000-200000000**
Elapsed time [s]	3,893	1,847	3,289	3,142	1,554	2,752
Instructions retired	22,048E+09	10,905E+09	17,928E+09	17,855E+09	8,830E+09	14,150E+09
Clockticks	20,009E+09	8,674E+09	15,386E+09	11,818E+09	5,359E+09	10,215E+09
Retiring [%]	55,9	78,0	67,9	54,3	63,4	59,1
Front-end bound [%]	4,4	2,7	3,6	2,6	3,9	3,2
Back-end bound [%]	38,3	20,7	24,0	41,7	32,1	34,0
Memory bound [%]	33,1	17,6	19,7	34,8	29,0	30,1
Core bound [%]	5,2	3,1	4,3	6,9	3,2	3,9
Effective physical core [%]	38,4	36,5	37,4	37,7	31,8	35,7
Przyspieszenie przetwarzania	0,775	0,749	0,792	0,960	0,891	0,947
Prędkość przetwarzania [liczb/s]	51,374E+06	54,142E+06	30,404E+06	63,653E+06	64,350E+06	36,337E+06
Efektywność przetwarzania	0,505	0,512	0,529	0,636	0,701	0,663

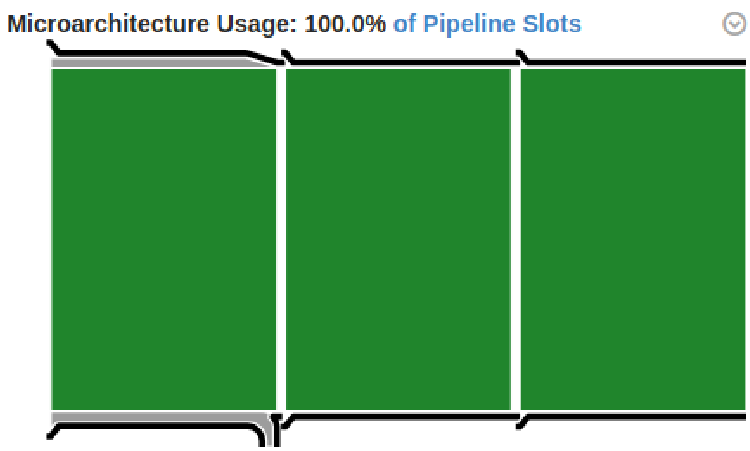
Z powyższej tabeli wynika, że przetwarzanie w oparciu o podejście funkcyjne jest mniej efektywne niż przetwarzanie w wariancie domenowym. Najlepszy czas uzyskany w wariancie domenowym jest o 50% większy niż najlepszy czas w wariancie funkcyjnym. Interesującym aspektem jest osiąganie czasu przetwarzania zbliżonego do przetwarzania w oparciu o sekwencyjną metodę Sita. Niska efektywność podejścia funkcyjnego może być spowodowana dużą liczbą komunikacji i synchronizacji.

5 Wnioski

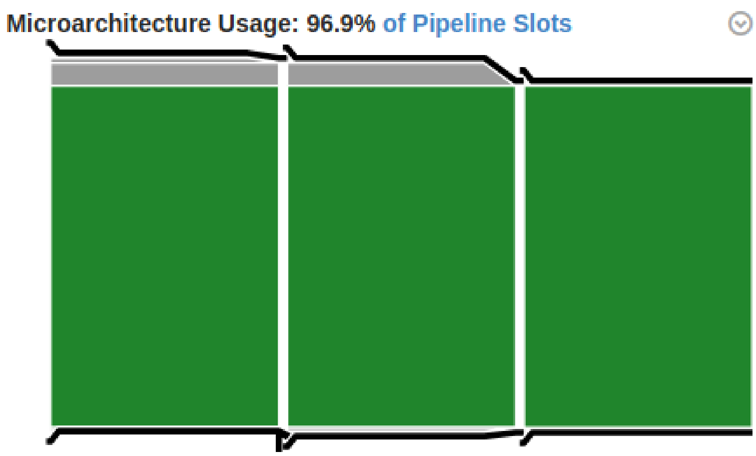
Największą prędkością przetwarzania wykazało się podejście Równoległe domenowe osiągając czas przetwarzania 2.375s dla liczb od 2-200000000. Dla odpowiadającego problemu wersja funkcyjna osiągnęła czas 3.893s, sito sekwencyjne - 3.017s, 163.435s dla dzielenia sekwencyjnego.

W przeprowadzonym eksperymencie najlepsze wykorzystanie struktury procesora uzyskaliśmy w następujących przypadkach:

- wariant domenowy (4 wątki, instancja 2...MAX/2):

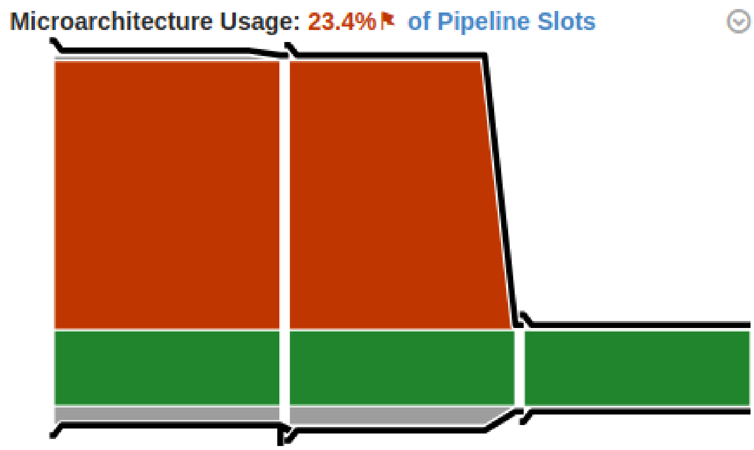


- wariant funkcyjny (8 wątków, instancja 2...MAX/2):

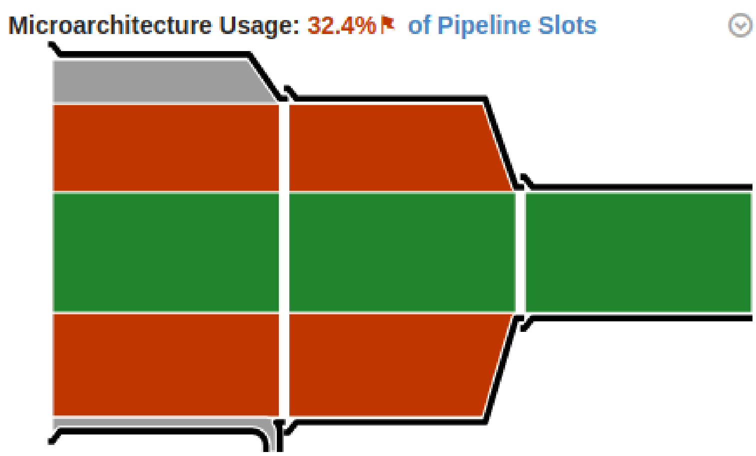


Natomiast najgorsze wykorzystanie mikroarchitektury wystąpiło dla:

- wariant funkcyjny (4 wątki, instancja 2...MAX/2):



- wariant domenowy (8 wątków, instancja 2...MAX/2):



W powyższych dwóch przypadkach powstają wąskie gardła.
Podejście domenowe charakteryzuje się przyspieszeniem przetwarzania 1.270. Podejście funkcyjne osiągnęło przyspieszenie o wartości 0.775 (Przyspieszenia zostały podane dla instancji problemu - 8 wątków 2-200000000). Efektywne wykorzystanie procesora dla tej samej instancji problemu to kolejno: 42,6% oraz 38,4% dla przetwarzania domenowego i funkcyjnego. Jakość zrównoleglenia przetwarzania jest znacząco lepsze w podejściu domenowym i jako jedyna prowadzi do wymiernych korzyści w czasie przetwarzania.

Najbardziej efektywnym podejściem równoległym w przeprowadzanym przez nas eksperymencie okazało się być podejście domenowe. Osiągnęliśmy tutaj efektywność na poziomie średnio 0,814. W przypadku podejścia funkcyjnego efektywność jest niższa i wynosi średnio 0,591. W wariancie domenowym efektywność utrzymuje się na podobnym poziomie zarówno w przypadku przetwarzania dla 4, jak i 8 wątków. Natomiast przy podejściu funkcyjnym efektywność wzrasta przy przetwarzaniu z przydzielonymi 4 wątkami w porównaniu do 8 wątków.
Ograniczenia, które mogą wpływać na efektywność przetwarzania to duża liczba komunikacji i synchronizacji.

5.1 Tabela podsumowująca

Tabela 4: Tabela podsumowująca

Metoda / Parametry	**Dzielenie**	**Usuwanie wielokrotności funkcyjne**	**Usuwanie wielokrotności domenowe**
Wielkość instancji [zakres liczb]	2-100000000	2-100000000	2-100000000
Liczba procesorów	1	2	4
Liczba wątków	1	4	8
Prędkość przetwarzania [liczb/s]	1,586E+06	64,350E+06	96,899E+06

Na podstawie powyższej tabeli można stwierdzić, że zdecydowanie największą prędkością przetwarzania charakteryzuje się przetwarzanie w oparciu o wariant domenowy zrównoleglenia.