Projekt 1 OMP

Laboratorium z przetwarzania równoległego

Mateusz Kreczmer 151736

mateusz.kreczmer@student.put.poznan.pl

Piotr Krzyszowski 151909

piotr.krzyszowski@student.put.poznan.pl

Grupa L4, czwartek 16:50, tygodnie nieparzyste

Politechnika Poznańska

Informatyka, WIiT, semestr VI

Wymagany termin oddania: 10.05.2024

Rzeczywisty termin oddania: 10.05.2024

Wersja sprawozdania: 1.0

Opis zadania:

Badanie efektywności przetwarzania równoległego w problemie znajdowania liczb pierwszych za pomocą Sita Eratostenesa w różnych wariantach.

Wykorzystany system obliczeniowy

Procesor CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz

Szybkość podstawowa: 2,42 GHz

Liczba gniazd: 1

Liczba rdzeni: 4

Liczba procesorów logicznych: 8

Pamięć podręczna poziomu 1: 320 KB

Pamięć podręczna poziomu 2: 5,0 MB

Pamięć podręczna poziomu 3: 8,0 MB

System operacyjny: Windows 11

Oprogramowanie użyte do przygotowania kodu: Visual Studio Code

Oprogramowanie użyte do testowania kodu: Intel VTune Profiler

Wykorzystane warianty kodu

[K1] Liczby pierwsze wyznaczane sekwencyjnie przez dzielenie w zakresie <m,n>

```
bool* result = (bool*)malloc((n - m + 1) * sizeof(bool));
memset(result, true, (n - m + 1) * sizeof(bool));
bool* primeArray = (bool*)malloc((sqrt(n) + 1) * sizeof(bool));
memset(primeArray, true, (sqrt(n) + 1) * sizeof(bool));

for (int i = 2; i * i <= n; i++) {
    for (int j = 2; j * j <= i; j++) {
        if (primeArray[j] == true && i % j == 0) {
            primeArray[i] = false;
            break;
        }
    }
}

int numOfPrimesInBetween = 0;

for (int i = m; i <= n; i++) {
    bool isPrime = true;
    for (int j = 2; j * j <= i; j++) {
        if (primeArray[j] == true && i % j == 0) {
            result[i - m] = false;
            isPrime = false;
            break;
        }
    }
    if (isPrime && i >= 2) {
        numOfPrimesInBetween++;
    }
}
```

Kod 1. Proste sito sekwencyjne

Ten fragment kodu najpierw generuje tablicę `primeArray`, która zawiera informacje o liczbach pierwszych w przedziale od 2 do pierwiastka kwadratowego z `n`, wykorzystując algorytm sita Eratostenesa. Następnie iteruje przez liczby w zadanym zakresie od `m` do `n`, sprawdzając ich podzielność przez liczby pierwsze z `primeArray`, aby określić, czy dana liczba jest pierwsza. Wyniki są zapisywane w tablicy `result`, a jednocześnie zliczane jest także wystąpienie liczb pierwszych w zakresie. Ostatecznie, liczba znalezionych liczb pierwszych w przedziale jest zwracana jako wartość `numOfPrimesInBetween`.

[K2] Liczby pierwsze wyznaczane równolegle przez dzielenie w zakresie <m,n>

```
bool* result = (bool*)malloc((n - m + 1) * sizeof(bool));
memset(result, true, (n - m + 1) * sizeof(bool));
bool* primeArray = (bool*)malloc((sqrt(n) + 1) * sizeof(bool));
memset(primeArray, true, (sqrt(n) + 1) * sizeof(bool));

for (int i = 2; i * i <= n; i++) {
    for (int j = 2; j * j <= i; j++) {
        if (primeArray[j] == true && i % j == 0) {
            primeArray[i] = false;
            break;
        }
    }
}

omp_set_num_threads(8);
#pragma omp parallel
{
    #pragma omp for schedule(dynamic)
    for (int i = m; i <= n; i++) {
        for (int j = 2; j * j <= i; j++) {
            if (primeArray[j] == true && i % j == 0) {
                result[i - m] = false;
                break;
            }
        }
    }
}
```

Kod 2. Proste sito równoległe

Ten fragment kodu rozszerza poprzednią wersję o wielowątkowość przy wyznaczaniu liczb pierwszych w zadanym zakresie. Za pomocą dyrektywy OpenMP, program równolegle przetwarza pętlę `for`, iterującą przez liczby w zadanym zakresie od `m` do `n`, sprawdzając ich podzielność przez liczby pierwsze z `primeArray`. Natomiast dyrektywa `schedule(dynamic)` umożliwia elastyczne przydzielanie iteracji do wątków.

[K3] Sito sekwencyjne bez lokalności dostępu do danych

```
bool* result = (bool*)malloc((n - m + 1) * sizeof(bool));
memset(result, true, (n - m + 1) * sizeof(bool));
bool* primeArray = (bool*)malloc((n + 1) * sizeof(bool));
memset(primeArray, true, (n + 1) * sizeof(bool));

for (int i = 2; i * i * i * i <= n; i++) {
    if (primeArray[i] == true) {
        for (int j = i * i; j * j <= n; j += i) {
            primeArray[j] = false;
        }
    }
}


for (int i = 2; i * i <= n; i++) {
    if (primeArray[i]) {
        int firstMultiple = (m / i);
        if (firstMultiple <= 1) {
            firstMultiple = i + i;
        } else if (m % i) {
            firstMultiple = (firstMultiple * i) + i;
        } else {
            firstMultiple = (firstMultiple * i);
        }
        for (int j = firstMultiple; j <= n; j += i) {
            result[j - m] = false;
        }
    }
}
```

Kod 3. Sito sekwencyjne bez lokalności dostępu do danych

Pierwsza pętla iteruje przez liczby do czwartej potęgi mniejsze lub równe n, eliminując wielokrotne potęgi liczb pierwszych z primeArray. Następnie druga pętla sprawdza liczby pierwsze z primeArray w zakresie od 2 do pierwiastka kwadratowego z n, a następnie dla każdej z tych liczb znajduje pierwszą wielokrotność większą lub równą m, iterując przez kolejne wielokrotności i oznaczając odpowiednie wartości w tablicy result jako fałszywe.

[K3A] Sito sekwencyjne z lokalnością dostępu do danych

```
bool* result = (bool*)malloc((n - m + 1) * sizeof(bool));
memset(result, true, (n - m + 1) * sizeof(bool));
bool* primeArray = (bool*)malloc((n + 1) * sizeof(bool));
memset(primeArray, true, (n + 1) * sizeof(bool));

int blockSize = (int)(0.375 * 1024 * 1024);
int numberOfBlocks = (n - m) / blockSize;
if ((n - m) % blockSize != 0) {
    numberOfBlocks++;
}

for (int i = 0; i < numberOfBlocks; i++) {
    int low = m + i * blockSize;
    int high = m + i * blockSize + blockSize;
    if (high > n) {
        high = n;
    }
    for (int j = 2; j * j <= high; j++) {
        if (primeArray[j]) {
            int firstMultiple = (low / j);
            if (firstMultiple <= 1) {
                firstMultiple = j + j;
            } else if (low % j) {
                firstMultiple = (firstMultiple * j) + j;
            } else {
                firstMultiple = (firstMultiple * j);
            }
            for (int k = firstMultiple; k <= high; k += j) {
                result[k - m] = false;
            }
        }
    }
}
```

Kod 4. Sito sekwencyjne z lokalnością dostępu do danych

Ten fragment kodu wprowadza lokalność dostępu do danych poprzez podział zakresu liczb na bloki o określonym rozmiarze. Każdy blok obejmuje zakres liczb od `low` do `high`, gdzie `low` i `high` są granicami bieżącego bloku. Dla każdego bloku, iteruje się przez liczby pierwsze z `primeArray` w zakresie od 2 do pierwiastka kwadratowego z `high`, a następnie znajduje pierwszą wielokrotność większą lub równą `low` dla każdej z tych liczb pierwszych, iterując przez kolejne wielokrotności i oznaczając odpowiednie wartości w tablicy `result` jako fałszywe. Wielkość bloku ‘blocksize’ została oszacowana poprzez eksperymentowanie nad jej wartością w granicach <0.0, 5.0 * 1024 * 1024> (górna wartość tego przedziału to rozmiar pamięci podręcznej poziomu 2).

[K4] Sito równoległe funkcyjne bez lokalności dostępu do danych

```
bool* result = (bool*)malloc((n - m + 1) * sizeof(bool));
memset(result, true, (n - m + 1) * sizeof(bool));
bool* primeArray = (bool*)malloc((n + 1) * sizeof(bool));
memset(primeArray, true, (n + 1) * sizeof(bool));

for (int i = 2; i * i * i * i <= n; i++) {
    if (primeArray[i] == true) {
        for (int j = i * i; j * j <= n; j += i) {
            primeArray[j] = false;
        }
    }
}

int sqrt_n = (int)sqrt(n);

omp_set_num_threads(8);
#pragma omp parallel for schedule(dynamic)
for (int i = 2; i <= sqrt_n; i++) {
    if (primeArray[i]) {
        int firstMultiple = (m / i);
        if (firstMultiple <= 1) {
            firstMultiple = i + i;
        } else if (m % i) {
            firstMultiple = (firstMultiple * i) + i;
        } else {
            firstMultiple = (firstMultiple * i);
        }
        for (int j = firstMultiple; j <= n; j += i) {
            result[j - m] = false;
        }
    }
}
```

Kod 5. Sito równoległe funkcyjne bez lokalności dostępu do danych

W kodzie 4, w porównaniu do kodu 3, dodano wielowątkowość przy obliczaniu liczb pierwszych. Pierwsza pętla pozostaje taka sama, eliminując wielokrotne potęgi liczb pierwszych z `primeArray`. Jednakże, w drugiej pętli równoległej, iteruje się przez liczby pierwsze od 2 do pierwiastka kwadratowego z `n`, znajdując pierwsze wielokrotności większe lub równe `m` dla każdej z tych liczb pierwszych i oznaczając odpowiednie wartości w tablicy `result` jako fałszywe.

[K4A] Sito równoległe funkcyjne bez lokalności dostępu do danych z dodatkowym warunkiem

```
for (int i = 2; i * i * i * i <= n; i++) {
    if (primeArray[i] == true) {
        for (int j = i * i; j * j <= n; j += i) {
            primeArray[j] = false;
        }
    }
}

int sqrt_n = (int)sqrt(n);

omp_set_num_threads(8);
#pragma omp parallel for schedule(dynamic)
for (int i = 2; i <= sqrt_n; i++) {
    if (primeArray[i]) {
        int firstMultiple = (m / i);
        if (firstMultiple <= 1) {
            firstMultiple = i + i;
        } else if (m % i) {
            firstMultiple = (firstMultiple * i) + i;
        } else {
            firstMultiple = (firstMultiple * i);
        }
        for (int j = firstMultiple; j <= n; j += i) {
            if (result[j-m]) {
                result[j - m] = false;
            }
        }
    }
}
```

Kod 6. Sito równoległe funkcyjne bez lokalności dostępu do danych (z dodatkowym warunkiem)

W porównaniu do kodu 4, w kodzie 4a dodano warunek sprawdzający, czy wartość w tablicy `result` jest prawdziwa, zanim zostanie oznaczona jako fałszywa. W praktyce oznacza to, że tylko jeśli wartość w `result[j - m]` jest prawdziwa, zostanie ona zmieniona na fałszywą. Dzięki temu unika się nadpisania wartości, które zostały już oznaczone jako fałszywe w poprzednich iteracjach. Ta modyfikacja może zmniejszyć liczbę operacji zapisu do tablicy `result`, co może przyspieszyć wykonywanie się kodu, szczególnie dla dużych zakresów liczb.

[K5] Sito równoległe domenowe z potencjalną lokalnością dostępu do danych

```
int blockSize = (int)(0.325 * 1024 * 1024);
int numberOfBlocks = (n - m) / blockSize;
if ((n - m) % blockSize != 0) {
    numberOfBlocks++;
}

omp_set_num_threads(8);
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < numberOfBlocks; i++) {
    int low = m + i * blockSize;
    int high = m + i * blockSize + blockSize;
    if (high > n) {
        high = n;
    }
    for (int j = 2; j * j <= high; j++) {
        if (primeArray[j]) {
            int firstMultiple = (low / j);
            if (firstMultiple <= 1) {
                firstMultiple = j + j;
            } else if (low % j) {
                firstMultiple = (firstMultiple * j) + j;
            } else {
                firstMultiple = (firstMultiple * j);
            }
            for (int k = firstMultiple; k <= high; k += j) {
                result[k - m] = false;
            }
        }
    }
}
```

Kod 7. Sito równoległe domenowe z potencjalną lokalnością dostępu do danych

W porównaniu z kodem 3a, kod 5 wykorzystuje wielowątkowość za pomocą biblioteki OpenMP, co pozwala na równoległe przetwarzanie zadania znajdowania liczb pierwszych w zadanym zakresie. Dzięki temu każdy blok poszukiwań może być wykonywany niezależnie przez oddzielny wątek, co w większości przypadków skraca czas wykonania programu. Rozmiar bloku `blockSize` jest kluczowy dla efektywnej pracy algorytmu i został przez nas dobrany poprzez eksperymentowanie z różnymi jego wartościami, na tej samej podstawie co w kodzie 3a.

Zagadnienie podziału pracy

Zagadnienie podziału pracy dotyczy efektywnego rozdzielenia zadań do wykonania równolegle przez wątki lub procesy. W przypadku programu sita funkcyjnego, najlepszym podejściem jest przydzielenie każdemu wątkowi pojedynczego zadania, co oznacza odkrycie jednej liczby pierwszej.

Innym podejściem jest metoda domenowa, polegająca na podziale problemu na bloki zadań i odpowiednim przydzieleniu ich wątkom. Wielkość tych bloków zależy od charakterystyki procesora, co pozwala na optymalne wykorzystanie zasobów obliczeniowych. Ważne jest unikanie zbyt dużego przydziału zadań, aby nie obciążać systemu i zapewnić równomierną dystrybucję pracy.

Zagadnienie przydziału zadań do procesorów

W rozwiązaniu funkcyjnym, zadania są przydzielane dynamicznie, co oznacza, że po zakończeniu przetwarzania jednego zadania, wątek otrzymuje kolejne dostępne zadanie. Nie ma sztywnego podziału na bloki z góry, co może prowadzić do nierównomiernego obciążenia wątków. Ponadto, czas wykonywania każdego zadania może się znacząco różnić, co może prowadzić do nieefektywnej synchronizacji i oczekiwania na zadania.

Natomiast podejście domenowe polega na podziale zadaniowego na bloki zakresowe, co pozwala na równomierne rozłożenie czasu wykonywania zadań. Nawet jeśli kolejne bloki mają większe liczby do przetworzenia, to ilość operacji pozostaje taka sama, co minimalizuje zbyt duże zróżnicowanie czasu przetwarzania. Dzięki temu, każdy proces może równomiernie wykorzystać dostępne zasoby obliczeniowe.

Podział przetwarzania na zadania

Podział przetwarzania na pojedyncze zadania dla każdego wątku został wybrany z myślą o prostocie implementacji i zrozumieniu działania programu. W algorytmach prostych, taka jak w przypadku wersji funkcyjnej, każdy wątek zajmuje się obliczaniem jednej liczby, co ułatwia śledzenie przebiegu działania. Wersja funkcyjna polega na eliminowaniu wielokrotności danej liczby z określonego zbioru liczb.

Natomiast podejście domenowe, które dzieli problem na bloki zadań, również zostało zastosowane dla ułatwienia równomiernego rozłożenia pracy na poszczególne wątki. Każde zadanie w tym przypadku polega na wyeliminowaniu wszystkich liczb pierwszych z określonego podzbioru. To podejście pozwala na efektywne wykorzystanie dostępnych zasobów

obliczeniowych, nawet w przypadku różnic w złożoności obliczeniowej poszczególnych zadań.

Dyrektywy OMP

Dyrektywa `#pragma omp parallel for` używa statycznego podziału liczby iteracji w pętli, rozdzielając je między dostępne wątki. `#pragma omp parallel for schedule(dynamic)` dzieli pętlę dynamicznie, umożliwiając wątkom ciągłe ubieganie się o nowe dane do obliczeń po zakończeniu przypisanych zadań. Dyrektywa `omp_set_num_threads(unsigned int)` określa liczbę wątków do wykonania dla danego bloku kodu.

Problem poprawnościowy

W kodzie, który wykorzystuje wielowątkowość, potencjalny problem wyścigu może wystąpić, gdy dwa lub więcej wątków próbuje równocześnie modyfikować wspólne zasoby, takie jak tablice lub zmienne. W takim przypadku może dojść do nieprzewidywalnych wyników ze względu na niemożność przewidzenia, który wątek dokona swojej modyfikacji jako pierwszy. Wyścigi mogą prowadzić do błędów w wynikach obliczeń oraz mogą wpływać na poprawność działania programu oraz jego prędkość przetwarzania, ponieważ wymagają dodatkowych mechanizmów synchronizacji, co może obniżyć wydajność.

Potencjalne problemy obliczeniowe

Potencjalne problemy efektywnościowe, takie jak false sharing, synchronizacja i brak zrównoważenia procesorów obliczeniami, nie występują w naszych kodach w znaczącym stopniu.

1. False sharing: W naszych kodach nie ma zauważalnego ryzyka false sharing, ponieważ operacje wykonywane przez różne wątki rzadko dotyczą tych samych obszarów pamięci, co minimalizuje potencjalne konflikty o dostęp do tych obszarów.

2. Synchronizacja: Programy nie wymagają instrukcji synchronizacyjnych, ponieważ operacje wykonywane przez wątki nie kolidują ze sobą. W przypadku algorytmu K5 możliwe jest zastosowanie synchronizacji dla obszaru już pierwiastkowego, ale w praktyce nie jest to konieczne, a użycie synchronizacji mogłoby wprowadzić dodatkowe opóźnienia.

3. Brak zrównoważenia procesora obliczeniami: W naszych kodach, w szczególności dzięki użyciu dyrektywy `schedule(dynamic)` praca jest równomiernie rozłożona na dostępne wątki, co zapobiega brakowi zrównoważenia obciążenia procesorów.

Prezentacja wyników

Rodzaje badanych algorytmów i ich efektywność przedstawiają poniższe tabele:

Algorytm	Algorytm	Liczba wątków	Zakres badanych liczb	Czas przetwarzania (s)	Liczba instrukcji kodu asemblera (instructions retired)	Liczba cykli procesora (clockticks)	Udział procentowy wykorzystanych zasobów procesora do przetwarzania kodu (retiring)	Ograniczenie wejścia (front-end bound)	Ograniczenie wyjścia (back-end bound)
K1	Proste sito sekwencyjne	1	<2, 100 000 000>	58,062	405,064,800,000	229,778,400,000	32.7%	25.7%	9.4%
			<50 000 000, 100 000 000>	38,669	253,960,800,000	153,571,200,000	30.7%	26.2%	8.0%
			<2, 50 000 000>	19,175	151,082,400,000	76,219,200,000	37.3%	23.2%	12.1%
K2	Proste sito równoległe	4	<2, 100 000 000>	20,074	423,237,600,000	291,242,400,000	33.4%	23.9%	8.6%
			<50 000 000, 100 000 000>	12,873	263,090,400,000	186,259,200,000	31.9%	25.0%	8.3%
			<2, 50 000 000>	7,954	160,236,000,000	109,480,800,000	37.6%	24.0%	10.7%
		8	<2, 100 000 000>	12,79	423,326,400,000	363,292,800,000	42.6%	21.6%	6.8%
			<50 000 000, 100 000 000>	8,609	263,222,400,000	231,888,000,000	40.9%	22.1%	6.5%
			<2, 50 000 000>	4,831	160,233,600,000	130,332,000,000	45.1%	22.4%	7.0%
K3	Sito sekwencyjne bez lokalności dostępu do danych	1	<2, 100 000 000>	1,109	1,641,600,000	4,077,600,000	4.8%	3.7%	92.6%
			<50 000 000, 100 000 000>	0,574	856,800,000	2,136,000,000	3.0%	2.8%	57.3%
			<2, 50 000 000>	0,498	804,000,000	1,867,200,000	11.5%	1.1%	98.3%
K3A	Sito sekwencyjne z lokalnością dostępu do danych	1	<2, 100 000 000>	0,662	3,967,200,000	2,536,800,000	27.4%	5.0%	64.5%
			<50 000 000, 100 000 000>	0,38	2,085,600,000	1,408,800,000	28.4%	6.9%	80.7%
			<2, 50 000 000>	0,329	1,922,400,000	1,248,000,000	21.2%	4.7%	67.0%
K4	Sito równoległe funkcyjne bez lokalności dostępu do danych	4	<2, 100 000 000>	0,597	1,608,000,000	7,243,200,000	0.8%	2.6%	83.4%
			<50 000 000, 100 000 000>	0,308	859,200,000	3,583,200,000	2.2%	2.4%	86.7%
			<2, 50 000 000>	0,292	832,800,000	3,489,600,000	0.8%	2.0%	66.3%
		8	<2, 100 000 000>	0,645	1,648,800,000	13,437,600,000	4.1%	4.0%	87.3%
			<50 000 000, 100 000 000>	0,269	859,200,000	4,977,600,000	0.5%	2.7%	79.4%
			<2, 50 000 000>	0,267	820,800,000	4,831,200,000	3.6%	3.7%	77.2%
K4A	Sito równoległe funkcyjne bez lokalności dostępu do danych (z dodatkowym warunkiem)	4	<2, 100 000 000>	0,459	1,968,000,000	4,444,800,000	5.2%	21.1%	69.4%
			<50 000 000, 100 000 000>	0,25	1,027,200,000	2,368,800,000	5.4%	11.7%	33.5%
			<2, 50 000 000>	0,248	998,400,000	2,277,600,000	4.0%	10.8%	35.9%
		8	<2, 100 000 000>	0,425	1,960,800,000	7,226,400,000	4.8%	21.7%	46.5%
			<50 000 000, 100 000 000>	0,242	1,032,000,000	3,273,600,000	8.4%	20.5%	38.8%
			<2, 50 000 000>	0,21	984,000,000	3,288,000,000	8.7%	21.5%	33.9%
K5	Sito równoległe domenowe z potencjalną lokalnością dostępu do danych	4	<2, 100 000 000>	0,319	4,034,400,000	3,273,600,000	31.0%	11.4%	41.8%
			<50 000 000, 100 000 000>	0,193	2,116,800,000	1,857,600,000	13.7%	10.5%	22.8%
			<2, 50 000 000>	0,15	1,927,200,000	1,562,400,000	26.0%	13.3%	40.2%
		8	<2, 100 000 000>	0,277	4,010,400,000	3,736,800,000	31.2%	18.2%	40.2%
			<50 000 000, 100 000 000>	0,14	2,080,800,000	2,198,400,000	22.9%	22.1%	40.9%
			<2, 50 000 000>	0,131	1,927,200,000	1,946,400,000	17.0%	14.2%	37.8%

Tabela 1. Wyniki eksperymentów

Algorytm	Algorytm	Liczba wątków	Zakres badanych liczb	Ograniczenie systemu pamięci (memory bound)	Ograniczenie jednostek wykonawczych (core bound)	Wykorzystanie rdzeni procesora (effective physical core utilization)	Przyspieszenie przetwarzania równoległego	Prędkość przetwarzania (liczby / s)	Efektywność przetwarzania równoległego
K1	Proste sito sekwencyjne	1	<2, 100 000 000>	1.0%	8.4%	23.0%	-	1,72E+06	-
			<50 000 000, 100 000 000>	0.9%	7.1%	22.7%	-	2,59E+06	-
			<2, 50 000 000>	1.2%	10.9%	23.3%	-	5,22E+06	-
K2	Proste sito równoległe	4	<2, 100 000 000>	1.3%	7.3%	75.5%	0,3457	4,98E+06	0,0864
			<50 000 000, 100 000 000>	1.1%	7.1%	78.4%	0,3329	7,77E+06	0,0832
			<2, 50 000 000>	2.0%	8.7%	65.9%	0,4148	1,26E+07	0,1037
		8	<2, 100 000 000>	1.2%	5.5%	93.6%	0,2203	7,82E+06	0,0275
			<50 000 000, 100 000 000>	1.1%	5.4%	88.4%	0,2226	1,16E+07	0,0278
			<2, 50 000 000>	1.5%	5.5%	88.9%	0,2519	2,07E+07	0,0315
K3	Sito sekwencyjne bez lokalności dostępu do danych	1	<2, 100 000 000>	48.2%	44.4%	18.1%	-	9,02E+07	-
			<50 000 000, 100 000 000>	29.9%	27.4%	14.4%	-	1,74E+08	-
			<2, 50 000 000>	45.9%	52.3%	24.9%	-	2,01E+08	-
K3A	Sito sekwencyjne z lokalnością dostępu do danych	1	<2, 100 000 000>	30.7%	33.8%	22,00%	-	1,51E+08	-
			<50 000 000, 100 000 000>	38.5%	42.2%	26.8%	-	2,63E+08	-
			<2, 50 000 000>	37.9%	29.1%	15.1%	-	3,04E+08	-
K4	Sito równoległe funkcyjne bez lokalności dostępu do danych	4	<2, 100 000 000>	41.7%	41.7%	54.5%	0,5383	1,68E+08	0,1346
			<50 000 000, 100 000 000>	44.3%	42.4%	44.1%	0,5366	3,25E+08	0,1341
			<2, 50 000 000>	32.4%	33.9%	54.5%	0,5863	3,42E+08	0,1466
		8	<2, 100 000 000>	44.0%	43.3%	69.6%	0,5816	1,55E+08	0,0727
			<50 000 000, 100 000 000>	38.0%	41.4%	48.9%	0,4686	3,72E+08	0,0586
			<2, 50 000 000>	39.7%	37.5%	61.5%	0,5361	3,75E+08	0,0670
K4A	Sito równoległe funkcyjne bez lokalności dostępu do danych (z dodatkowym warunkiem)	4	<2, 100 000 000>	56.2%	13.2%	57.4%	0,4139	2,18E+08	0,1035
			<50 000 000, 100 000 000>	27.4%	6.1%	29.6%	0,4355	4,00E+08	0,1089
			<2, 50 000 000>	30.6%	5.2%	42.1%	0,4980	4,03E+08	0,1245
		8	<2, 100 000 000>	36.5%	10.0%	66.5%	0,3832	2,35E+08	0,0479
			<50 000 000, 100 000 000>	30.3%	8.6%	52,00%	0,4216	4,13E+08	0,0527
			<2, 50 000 000>	24.1%	9.8%	62.6%	0,4217	4,76E+08	0,0527
K5	Sito równoległe domenowe z potencjalną lokalnością dostępu do danych	4	<2, 100 000 000>	18.8%	23.0%	39.1%	0,4819	3,13E+08	0,1205
			<50 000 000, 100 000 000>	10.2%	12.5%	24,00%	0,5079	5,18E+08	0,1270
			<2, 50 000 000>	18.9%	21.3%	46,30%	0,4559	6,67E+08	0,1140
		8	<2, 100 000 000>	16.2%	24.0%	51.3%	0,4184	3,61E+08	0,0523
			<50 000 000, 100 000 000>	17.2%	23.7%	45.8%	0,3684	7,14E+08	0,0461
			<2, 50 000 000>	16.0%	21.8%	49,00%	0,3982	7,63E+08	0,0498

Tabela 2. Dalsza część wyników

Intel® VTune™ wykorzystuje jednostkę monitorującą wydajność procesora, aby identyfikować nieefektywne ścieżki kodu, braki w pamięci podręcznej (cache misses), przestoje w pracy rdzeni itp. Do tego celu stosuje się Event Based Sampling. Gdy licznik zdarzeń osiągnie określony poziom (SAV - sample-after value), program przerywa działanie i przystępuje do analizy danych w aplikacji, przypisując je do odpowiednich instrukcji.

Program zbiera różnorodne statystyki dotyczące analizowanego kodu, co pozwala wykryć mało wydajne fragmenty i przedstawia procentowe wykorzystanie możliwości architektury komputera w trakcie analizy.

Wnioski

Analizując badane metody przetwarzania sieciowego, można stwierdzić, że skuteczność rozwiązania problemu jest zależna od wielu czynników, w tym struktury danych, optymalizacji dostępu do pamięci oraz wykorzystania zasobów procesora.

Metoda K5, czyli Sito równoległe domenowe z potencjalną lokalnością dostępu do danych, wykazuje się najwyższą prędkością przetwarzania dla wszystkich zakresów liczb i liczby wątków, co sugeruje jej uniwersalność i efektywność.

Natomiast metody K3 i K4 wydają się być mniej efektywne, ze względu na niskie wykorzystanie zasobów procesora oraz wąskie gardła związane z dostępem do danych.

Metoda K2, czyli proste sito równoległe, osiąga konkurencyjne wyniki przetwarzania równoległego dla większych zakresów liczb i liczby wątków, co wskazuje na skuteczność wykorzystania struktur wewnętrznych procesora.

Podsumowując, zastosowanie odpowiedniej metody przetwarzania zależy od charakterystyki danych oraz struktury procesora, przy czym metoda K5 wydaje się być najbardziej wszechstronna i efektywna.