Sprawozdanie na temat
«Znajdowanie liczb pierwszych w zakresie liczb»

Wersja №1
Grupa L6 Środa 9:45
Wymagany Termin oddania: 10.05.2024

Heorhi Zakharkevich 153992
heorhi.zakharkevich@student.put.poznan.pl
Jakub Korcz 151855
jakub.korcz@student.put.poznna.pl

1. Specyfikacja

- Intel(R) Core(TM) i5-8365U CPU @ 1.60GHz
- Ubuntu 22.04
- 4 proc. fizycznych
- 8 proc. logicznych
- Pamięc podręczna:
    - L1d: 128 KiB (4 instances)
    - L1i: 128 KiB (4 instances)
    - L2: 1 MiB (4 instances)
    - L3: 6 MiB (1 instance)

2. Kluczowe fragmenty i wyjaśnienia kodów algorytmów

- Liczby pierwsze wyznaczane sekwencyjnie przez dzielenie w zakresie <m,n> [k1]

```
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;
        }
    }
}

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;
        }
    }
}
```

Ten kod implementuje sekwencyjną metodę znajdowania liczb pierwszych w zakresie od m do n. Polega to na podziale każdej liczby w tym zakresie przez liczby pierwsze, które są możliwymi dzielnikami (od 2 do pierwiastka kwadratowego danej liczby). Wartość "false" oznacza, że liczba jest złożona.

- Liczby pierwsze wyznaczane równolegle przez dzielenie w zakresie <m,n> [k2]

```
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;
        }
    }
}

#pragma omp parallel
{
    #pragma omp for
    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;
            }
        }
    }
}
```

Ten kod oblicza liczby pierwsze w danym zakresie liczb od 1 do n przy użyciu obliczeń równoległych. Opiera się na tej samej metodzie, co poprzedni kod, ale teraz wykorzystuje rzetwarzanie równoległe w celu przyspieszenia obliczeń. Odbywa się to za pomocą dyrektywy równoległej #pragma omp w celu uruchomienia równoległego bloku kodu. Wewnątrz tego bloku kodu dyrektywa #pragma omp for służy do równego podziału pętli wyszukiwania liczb pierwszych pomiędzy wątki. Każdy wątek pracuje z własną częścią zakresu liczb, co pozwala na szybsze przetwarzanie.

- Sito sekwencyjne bez lokalności dostępu do danych [k3]

```
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) {
            if (j - m >= 0) {
                result[j - m] = false;
            }
        }
    }
}
```

Ten kod oblicza liczby pierwsze w danym zakresie od 1 do n za pomocą sekwencyjnego algorytmu bez uwzględnienia lokalności dostępu do danych. Drugim cyklem jest sekwencyjna obróbka zakresu liczb od m do n. W każdej iteracji sprawdzane są kolejne liczby z tego zakresu, aby określić, czy są one liczby pierwsze. Dla każdej sprawdzanej liczby, iteruje się przez potencjalne dzielniki (liczby od 2 do pierwiastka z tej liczby) i sprawdza się, czy liczba jest podzielna przez któryś z tych dzielników. Jeśli tak, oznacza to, że liczba nie jest liczbą pierwszą, a więc jej odpowiadający element w tablicy wynikowej jest ustawiany na false.

- Sito sekwencyjne z lokalnością dostępu do danych [k3a]

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

for (int start = 0; start <= n - m; start += blockSize) {
    int low = m + start;
    int high = m + start + blockSize - 1;
    if (high > n) high = n;

    for (int j = 2; j * j <= high; j++) {
        if (primeArray[j]) {
            int firstMultiple = (low / j) * j;
            if (firstMultiple < low) firstMultiple += j;
            if (firstMultiple < j * j) firstMultiple = j * j;

            for (int k = firstMultiple; k <= high; k += j) {
                result[k - m] = false;
            }
        }
    }
}
```

Ten kod dzieli obliczenia na bloki, co poprawia lokalność dostępu do danych poprzez ograniczenie liczby dostępów do pamięci o dużym czasie dostępu. Każdy blok obejmuje określony zakres liczb, a następnie w tym bloku wykreślane są liczby złożone przez odznaczenie ich wielokrotności, które zostały wcześniej zidentyfikowane jako liczby pierwsze i zapisane w tablicy primeArray. Ważne jest odpowiednie dobranie rozmiaru bloku, aby zoptymalizować wydajność poprzez minimalizację liczby dostępów do pamięci i operacji obliczeniowych.

- Sito równoległe funkcyjne bez lokalności dostępu do danych [k4]

```
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 N = ((int)sqrt( x: n));
#pragma omp parallel
{
    #pragma omp for
    for (int i = 2; 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) {
                if (j - m >= 0) {
                    result[j - m] = false;
                }
            }
        }
```

Ten kod prezentuje sposób wykorzystania OpenMP do równoległego przetwarzania w celu znalezienia liczb pierwszych za pomocą sita Eratostenesa. Jednakże, wątki operują na oddalonych od siebie indeksach tablicy, co prowadzi do problemów związanych z brakiem lokalności przestrzennej danych. Jest to spowodowane częstym odwoływaniem się do pamięci i konfliktami w dostępie do pamięci między wątkami, co znacznie obniża wydajność programu poprzez narzut synchronizacji i konflikty w dostępie do danych.

- Sito równoległe funkcyjne bez lokalności dostępu do danych [k4a]

```
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 N = ((int)sqrt( x n));
#pragma omp parallel for
for (int i = 2; 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) {
            if (j - m >= 0) {
                if (result[j-m]) result[j-m] = false;
            }
        }
    }
}
```

Ten kod jest bardzo podobny do tego, który występuje w kodzie wyżej. Jednakże, jedną istotną różnicą jest to, że przed zapisaniem, że liczba jest pierwsza, kod sprawdza, czy inny wątek już tego nie zrobił. Dzięki temu unikamy wielu niepotrzebnych operacji unieważniania linii pamięci, co może znacznie poprawić wydajność programu poprzez redukcję konfliktów w dostępie do danych i synchronizacji między wątkami.

- Sito równoległe domenowe z potencjalną lokalnością dostępu do danych [k5]

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

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

Ten kod dzieli całą przestrzeń obliczeń na mniejsze bloki, co jest skuteczną strategią do równomiernego rozłożenia pracy między wątki. Poprzez odpowiednie dostosowanie rozmiaru bloku staramy się zmniejszyć problem unieważniania linii danych, co może poprawić efektywność działania programu. Kluczowe fragmenty kodu to: obliczenie liczby bloków na podstawie wcześniej określonej wielkości bloku, określenie krawędzi bloku analizowanego w danej iteracji oraz wykonanie algorytmu w ramach każdego bloku, korzystając z wcześniejszych elementów kodu.

3. Tabele z przetestowanymi danymi

- Prędkość obliczeń (Mliczb/sek)

Zakres	k1	k2	k3	k3a	k4	k4a	k5
2, max	3.315433	18.934912	197.004635	680.749770	208.795263	316.243141	2.977113
max/2, max	3.334110	11.476978	310.617681	585.192274	334.012291	567.135309	2.132584
2, max/2	4.527165	21.066148	388.922384	730.332649	370.235668	573.022311	4.641901

- Tabela z znajdowaniem efektywnego rozmiara bloka danych dla K3a

Czas	Block Size
13.062223	2
6.880681	4
3.662629	8
1.943432	16
1.079469	32
0.614411	64
0.354062	128
0.139942	512
0.065942	1024
0.042614	4096
0.038779	8192
0.030266	32768
0.024151	65536
0.033531	131072
0.036334	262144
0.037126	524288

Najszybsze przetwarzanie jest przy blokie 65536: 0.024151

- Tabela z znajdowaniem efektywnego rozmiara bloka danych dla K5

Czas	Block Size
22.522664	2
11.529256	4
6.061992	8
3.422607	16
1.764045	32
1.044847	64
0.629721	128
0.283299	512
0.167573	1024
0.055748	4096
0.046102	8192
0.044062	32768
0.039177	65536
0.042371	131072
0.053614	262144
524288	524288

- Tabela dla algorytmów K1 oraz K3 (sekwencyjnych) z przyspiszeniem od K3a (najlepszy blok)

	k1	k3
czas	3,20875	0,06383
przyspieszenie	0,007526606934	0,3783644055

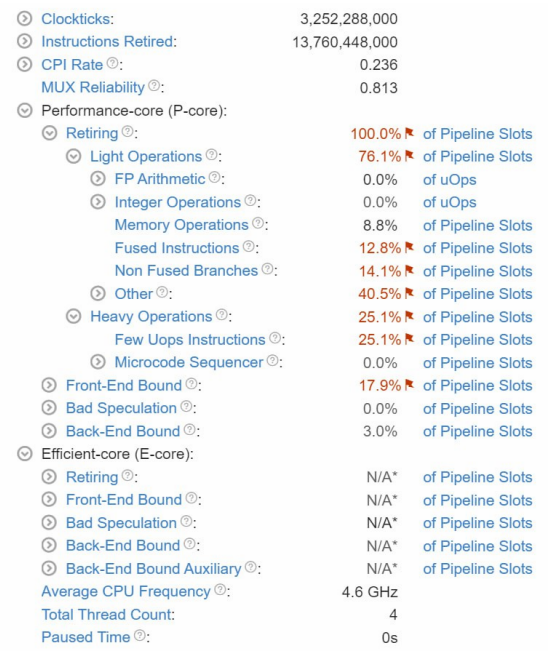
- Tabela dla algorytmów K2, K4, K4a z przyspiszeniem od K3a (najlepszy blok)

#pragma omp for shedule()		k2	k4	k4a
static, 2	czas	3,84446	0,59536	0,403469
	przyspieszenie	0,006282026605	0,04056537221	0,05985837821
static, 4	czas	3,761449	0,201546	0,391194
	przyspieszenie	0,006420663952	0,119828724	0,06173663195
static, 8	czas	3,82482	0,217835	0,385643
	przyspieszenie	0,006314284071	0,1108683178	0,06262527778
static, 16	czas	3,867928	0,279204	0,414316
	przyspieszenie	0,006243911469	0,08649947708	0,05829125595
static, 128	czas	3,841992	0,31259	0,392228
	przyspieszenie	0,006286062022	0,07726094885	0,0615738805

static, 512	czas	3,988156	0,276388	0,321009
	przyspieszenie	0,006055680871	0,08738078354	0,07523465074
static, 1024	czas	3,894602	0,280224	0,328316
	przyspieszenie	0,006201147126	0,08618462373	0,07356022856
static, 4096	czas	4,542767	0,185642	0,291521
	przyspieszenie	0,005316363353	0,1300944829	0,08284480363
static, 8192	czas	3,875817	0,175696	0,294741
	przyspieszenie	0,00623120235	0,1374590201	0,08193973692
static, 32768	czas	4,11979	0,296897	0,308645
	przyspieszenie	0,005862192005	0,08134470877	0,07824847316
static, 65536	czas	3,97542	0,278888	0,291495
	przyspieszenie	0,006075081375	0,08659748716	0,08285219301
static, 131072	czas	3,9541	0,294289	0,300227
	przyspieszenie	0,006107837435	0,08206558859	0,0804424652
• static, **524288**	czas	3,835643	0,266194	0,296742
	przyspieszenie	0,006296467111	0,09072706372	0,08138719831
dynamic	czas	4,214597	0,223248	0,285425
	przyspieszenie	0,005730322496	0,1081801405	0,08461417185

4. Screenshoty z VTune

- Dla K1



- Dla K2

Clockticks:	25,628,928,000	
Instructions Retired:	15,200,640,000	
CPI Rate:	1.686	
Performance-core (P-core):	1.540	
Efficient-core (E-core):	2.272	
MUX Reliability:	0.896	
Performance-core (P-core):		
Retiring:	23.2%	of Pipeline Slots
Front-End Bound:	37.4%	of Pipeline Slots
Bad Speculation:	0.5%	of Pipeline Slots
Back-End Bound:	38.9%	of Pipeline Slots
Efficient-core (E-core):		
Retiring:	75.8%	of Pipeline Slots
Front-End Bound:	5.1%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	41.2%	of Pipeline Slots
Core Bound:	0.0%	of Clockticks
Memory Bound:	41.2%	of Clockticks
Store Bound:	0.3%	of Clockticks
L1 Bound:	3.0%	of Clockticks
L2 Bound:	3.8%	of Clockticks
L3 Bound:	7.7%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Other Load Store:	26.5%	of Clockticks
Back-End Bound Auxiliary:	41.2%	of Pipeline Slots
Average CPU Frequency:	4.0 GHz	
Total Thread Count:	26	
Paused Time:	0s	

- Dla K3

Clockticks:	152,256,000	
Instructions Retired:	137,280,000	
CPI Rate:	1.109	
Performance-core (P-core):	1.109	
Efficient-core (E-core):	0.000	
MUX Reliability:	N/A*	
Performance-core (P-core):		
Retiring:	0.0%	of Pipeline Slots
Front-End Bound:	0.0%	of Pipeline Slots
Bad Speculation:	100.0%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Efficient-core (E-core):		
Retiring:	0.0%	of Pipeline Slots
Front-End Bound:	0.0%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Back-End Bound Auxiliary:	0.0%	of Pipeline Slots
Average CPU Frequency:	4.1 GHz	
Total Thread Count:	3	
Paused Time:	0s	

- Dla K3a

Clockticks:	109,824,000	
Instructions Retired:	192,192,000	
CPI Rate:	0.571	
MUX Reliability:	0.164	
Performance-core (P-core):		
Retiring:	0.0%	of Pipeline Slots
Front-End Bound:	0.0%	of Pipeline Slots
Bad Speculation:	100.0%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Efficient-core (E-core):		
Retiring:	24.0%	of Pipeline Slots
Front-End Bound:	24.0%	of Pipeline Slots
Bad Speculation:	51.9%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Back-End Bound Auxiliary:	0.0%	of Pipeline Slots
Average CPU Frequency:	4.8 GHz	
Total Thread Count:	3	
Paused Time:	0s	

*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

- Dla K4

Clockticks:	401,856,000	
Instructions Retired:	234,624,000	
CPI Rate:	1.713	
Performance-core (P-core):	1.438	
Efficient-core (E-core):	3.286	
MUX Reliability:	0.403	
Performance-core (P-core):		
Retiring:	0.0%	of Pipeline Slots
Front-End Bound:	2.9%	of Pipeline Slots
Bad Speculation:	97.1%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Efficient-core (E-core):		
Retiring:	0.0%	of Pipeline Slots
Front-End Bound:	0.0%	of Pipeline Slots
Bad Speculation:	100.0%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Back-End Bound Auxiliary:	0.0%	of Pipeline Slots
Average CPU Frequency:	4.2 GHz	
Total Thread Count:	25	
Paused Time:	0s	

- Dla K4a

Clockticks:	277,056,000	
Instructions Retired:	257,088,000	
CPI Rate:	1.078	
Performance-core (P-core):	1.021	
Efficient-core (E-core):	1.857	
MUX Reliability:	0.502	
Performance-core (P-core):		
Retiring:	0.0%	of Pipeline Slots
Front-End Bound:	22.5%	of Pipeline Slots
Bad Speculation:	77.5%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Efficient-core (E-core):		
Retiring:	0.0%	of Pipeline Slots
Front-End Bound:	0.0%	of Pipeline Slots
Bad Speculation:	100.0%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Back-End Bound Auxiliary:	0.0%	of Pipeline Slots
Average CPU Frequency:	4.5 GHz	
Total Thread Count:	16	
Paused Time:	0s	

- Dla K5

Clockticks:	247,104,000	
Instructions Retired:	539,136,000	
CPI Rate:	0.458	
MUX Reliability:	0.870	
Performance-core (P-core):		
Retiring:	N/A*	of Pipeline Slots
Front-End Bound:	N/A*	of Pipeline Slots
Bad Speculation:	N/A*	of Pipeline Slots
Back-End Bound:	N/A*	of Pipeline Slots
Efficient-core (E-core):		
Retiring:	0.0%	of Pipeline Slots
Front-End Bound:	0.0%	of Pipeline Slots
Bad Speculation:	100.0%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Back-End Bound Auxiliary:	0.0%	of Pipeline Slots
Average CPU Frequency:	4.3 GHz	
Total Thread Count:	25	
Paused Time:	0s	

**N/A is applied to metrics with undefined value. There is no data to calculate the metric.*