Sprawozdanie z projektu i eksperymentu obliczeniowego

Laboratorium z przetwarzania równoległego

Szymon Sroka 141312, Weronika Radzi 141303
Grupa L8, piątki w tygodniach nieparzystych godzina 8:00

Termin oddania sprawozdania: 8 maja 2021r.
Rzeczywisty termin oddania sprawozdania: 7 maja 2021r.
Wersja pierwsza

szymon.j.sroka@student.put.poznan.pl, weronika.radzi@student.put.poznan.pl

1. Temat zadania

Realizowane zadanie rozwiązuje problem znajdowania liczb pierwszych w podanym za pomocą parametrów problemu przedziału .

2. Opis wykorzystanego systemu obliczeniowego i systemu operacyjnego:

Intel Core i5-5200U CPU @ 2.20GHz (4CPUs), ~2.2GHz
Liczba procesorów logicznych: 4
Liczba procesorów fizycznych: 2
3MB Cache
System Windows 10 Pro 64-bit
Oprogramowanie Visual Studio Code 2017 oraz Intel vTune

3. Prezentacja przygotowanych wariantów kodów

Jako badany przedział przyjęliśmy zakres liczb <2,90000000>. Znalezione liczby pierwsze przechowujemy w tablicy statycznej *matrix* o typie *boolean.* Programy wyświetlają liczbę znalezionych liczb pierwszych w danym przedziale, a ponadto, po ustawieniu zmiennej *enable_debug* na true, wypisują znalezione liczby pierwsze.

a. KOD NR 1: wersja sekwencyjna - modulo:

```
#include <stdio.h>
#include <omp.h>
#include <time.h>
#include <math.h>

#define MAX 90000000
#define MIN 45000000

bool matrix[MAX + 1] = { 0 }, enable_debug = false;

int main(int argc, char* argv[])
{
 for (int num = MIN; num <= MAX; num++)
 {
		bool is_prime = true;
		for (int i = 2; i <= pow(num, 1.0 / 2); i++)
		{
			if (num%i == 0)
			{
				is_prime = false;
				break;
			}
		}
		if (is_prime)
		{
```

```
                matrix[num] = 1;
            }
        }
        ...
    }
```

b. **KOD NR 2: wariant zrównoleglenia modulo: zastosowanie podejścia domenowego.** Wątek bada podzbiór liczb pierwszych. Uzyskaliśmy taki efekt poprzez zastosowanie klauzuli #pragma omp parallel for num_threads(4) przy pętli iterującej po tablicy matrix:

```
int main(int argc, char* argv[])
{
        #pragma omp parallel for num_threads(4)
        for (int num = MIN; num <= MAX; num++)
        {
                …
         }
}
```

- W tej wersji kodu zastosowaliśmy domyślny sposób podziału pracy między wątkami, w naszym przypadku jest to podział statyczny z rozmiarem *chunk* równym *#iterations / #threads,* na przykład w przypadku przedziału *2...90000000* i *4* wątków każdy z procesów dostanie do wykonania w przybliżeniu *(90000000-2)/4=22500000* iteracji.
- Przydział pracy odbywa się statycznie. Spodziewamy się zrównoważenia procesorów przetwarzaniem, ponieważ każdy z wątków dostaje podobną ilość obliczeń do wykonania. Wątki pracują na różnych elementach tablicy wykreśleń, a liczba liczb pierwszych jest określana poza blokiem równoległym, stąd nie ma potrzeby zapewnienia synchronizacji na przykład za pomocą dyrektywy *atomic*, nie ma zagrożenia false-sharingiem. Z tych również powodu użycie dyrektywy dynamicznie przydzielającej zadania jest tutaj zbędne i zbyt kosztowne.

Definicja false-sharingu:

Zjawisko to polega na wielokrotnym, cyklicznym (wywołanym przez wielokrotne zapisy w różnych procesorach wartości do tej samej linii pamięci podręcznej procesora): • unieważnianiu powielonych linii pamięci w pp procesorów, które

przestają być aktualne w wyniku zapisu danych do jednej z wielu kopii tej linii, • konieczności sprowadzenia do pamięci podręcznej procesora realizującego kod wątku (korzystającego z danych sąsiednich) aktualnej kopii danych z unieważnionej linii."

c. KOD NR 3: wersja sekwencyjna - sito ('dodawanie'):

```
#include <stdio.h>
#include <omp.h>
#include <time.h>
#include <math.h>

#define MAX 90000000
#define MIN 2

bool matrix[MAX + 1], final_matrix[MAX + 1], enable_debug = false;

int main(int argc, char* argv[])
{
   //prepare arrays
   for (int i = 0; i < MAX; i++)
   {
            matrix[i] = 0;
            final_matrix[i] = 1;
   }

   final_matrix[0] = 0;
   final_matrix[1] = 0;


   for (int num = 2; num*num <= MAX; num++)
   {
            bool is_prime = true;
            for (int i = 2; i <= pow(num, 1.0 / 2); i++)
            {
                     if (num%i == 0)
                     {
                              is_prime = false;
                              break;
                     }
            }
            if (is_prime)
            {
                     matrix[num] = 1;
            }
   }

//wykreslamy liczby pierwsze

   for (int i = 0; i <= MAX; i++)
   {
            if (matrix[i])
            {
                     for (int j = i + i; j < MAX; j = j + i)
                     {
                              //printf("wykreslam: %d \n,", j);
                              final_matrix[j] = 0;
                     }
            }
   }
   ....
}
```

W przypadku wersji sekwencyjnej 'sita' spodziewamy się znacznie krótszego czasu przetwarzania niż w przypadku wersji 'modulo'.

d. **KOD NR 4: wariant zrównoleglenia sita - zastosowanie podejścia funkcyjnego**, w którym procesy otrzymują całą tablicę wykreśleń i fragment zbioru liczb pierwszych, których wielokrotności są usuwane. Uzyskaliśmy taki efekt przy zastosowaniu dyrektywy #pragma omp parallel for z parametrem schedule(static,2), to znaczy ze statycznym podziałem iteracji między wątki, w blokach o wielkości 10 iteracji:

```
#pragma omp parallel for schedule(static, 10) num_threads(4)
for (int i = 0; i <= sqr_variable; i++)
{
        ...
}
```

- W przypadku tego wariantu satysfakcjonujące wyniki uzyskaliśmy stosując statyczny podział pracy z rozmiarem *chunk* równym *10.* Oznacza to, że przy pracy na przykład 4 wątków pierwszy z nich otrzyma pierwsze 10 iteracji, drugi kolejne 10, trzeci kolejne 10, czwarty kolejne 10, a następnie znów pierwszy wątek 10 iteracji i cykl się powtarza.
- Spodziewamy się, że będzie to odpowiedni przydział dla tego problemu ze względu na dość nierównomierny rozkład liczb pierwszych wśród liczb całkowitych; na początku pętli for następuje sprawdzenie w statycznej tablicy matrix, czy liczba *i* jest pierwsza. Jeśli okaże się, że nie jest, wątek kończy iterację i przechodzi do kolejnej. Uważamy, że cykliczny podział iteracji pozwoli na uniknięcie sytuacji, gdy potencjalnie jeden wątek nie wykonuje pracy, bo trafia wyłącznie na liczby złożone i sprawi, że podział pracy między wątkami będzie zrównoważony.
- Wątki pracują na różnych elementach tablicy wykreśleń, a liczba liczb pierwszych jest określana poza blokiem równoległym, stąd nie ma potrzeby zapewnienia synchronizacji na przykład za pomocą dyrektywy *atomic*, nie ma zagrożenia false-sharingiem. Ponadto tablica wykreślanych liczb jest współdzielona tylko do odczytu - takie współdzielenie jest bezpieczne.

W celu przyspieszenia pracy programu, zrównolegliliśmy również wyszukiwanie liczb pierwszych z zakresu <2...$\sqrt{MAX}$>, używanych do wykreślania w celu uzyskania finalnego zbioru liczb:

```
#pragma omp parallel for num_threads(2)
        for (int num = 2; num <= sqr_variable; num++)
        {
        ...
        }
```

- Taki zabieg ma na celu poprawę efektywności przetwarzania. Podział pracy jest domyślny - statyczny z chunkami wielkości *#iterations / #threads,* analogicznie jak w przypadku domenowej wersji ‘modulo’.
- Podobnie jak w podpunkcie wyżej, również tutaj wątki pracują na różnych elementach tablicy wynikowej, stąd nie ma zagrożenia wyścigiem lub false-sharingiem.

e. **KOD NR 5: wariant zrównoleglenia sita - zastosowanie podejścia domenowego**, w którym procesy otrzymują fragment tablicy wykreśleń i całą tablicę liczb pierwszych (do pierwiastka z maksimum zakresu). W tym celu określiliśmy rozmiar fragmentów tablicy wykreśleń, na którym ma wykonywać operację poszczególny wątek (w naszym wypadku chunk_size=1000000 okazał się wystarczający) oraz za pomocą dyrektywy #pragma omp parallel rozdzieliliśmy te fragmenty dla poszczególnych wątków:

```
#pragma omp parallel for schedule(static, 1) num_threads(4)
for (int chunk_num = 0; chunk_num <= (MAX-MIN)/chunk_size ; chunk_num++)
{
        …
}
```

- W tym podejściu ręcznie wyznaczyliśmy przedziały wielkości miliona liczb, w których wątki mają wykreślać wielokrotności dostarczonych liczb pierwszych. Następnie za pomocą dyrektywy *#parallel* przydzieliliśmy statycznie wyznaczone przedziały do poszczególnych wątków. Pierwszy wątek otrzymał pierwszy z przedziałów, drugi - drugi z przedziałów, taki przydział powtarza się cyklicznie do osiągnięcia progu MAX.
- Zastosowaliśmy statyczny podział iteracji do procesorów, ponieważ nie jest w tym wariancie wymagana komunikacja między wątkami, a na przykład dynamiczny podział spowodowałby spadek efektywności przetwarzania
- W naszej ocenie statyczne, cykliczne przypisywanie zadań do wątków będzie w tym wariancie odpowiednie, ponieważ wątki pracują na zupełnie odrębnych

fragmentach tablicy wykreśleń, ponadto tablica liczb pierwszych, których wielokrotności są wykreślane, jest współdzielona jedynie do odczytu. Nie ma ryzyka wyścigu czy false-sharingu.

W tym podejściu, aby ograniczyć liczbę wykonywanych operacji, zmodyfikowaliśmy program tak, aby wątek, działając na fragmencie tablicy wykreśleń, nie musiał dodawać wielokrotności wykreślanej liczby, gdy znajdują się ona poza badanym przedziałem, to znaczy aby działając na przykład na fragmencie liczb 9...20 nie wykreślał niepotrzebnie kolejnych wielokrotności 2 takich jak (4, 6, 8), ale rozpoczął wykreślanie od razu od liczby 10. Dla badanego przez nas zakresu 2...90000000 pozwoliło to na znacznie przyspieszenie pracy programu (KOD NR 6):

```
if (matrix[i]) //tablica liczb pierwszych z zakresu <2...pierwiastek(MAX)>
{
		int closest_num = my_start;
		while (closest_num%i != 0)closest_num++;

		for (int j = closest_num; j <= my_end; j = j + i)
			{
				if (j == i)continue;
				final_matrix[j] = 0;
			}
	}
```

Ponadto początkowo do zainicjowania wartości w tablicach potrzebnych do obliczeń używaliśmy następującej pętli:

```
for (int i = 0; i < MAX; i++)
	{
		matrix[i] = 0;
		final_matrix[i] = 1;
	}
```

jednak takie podejście okazało się zbyt kosztowne pod kątem dostępu do pamięci. Stąd zdecydowaliśmy się na usprawnienie obliczeń przez zastosowanie

```
std::fill(std::begin(matrix), std::end(matrix), 0);
std::fill(std::begin(final_matrix), std::end(final_matrix), 1);
```

takie podejście okazało się znacznie lepsze pod kątem efektywności przetwarzania.

4. **Prezentacja wyników i omówienie przebiegu eksperymentu obliczeniowo-pomiarowego.**
   a. sposób zbierania przez oprogramowanie Intel Vtune informacji o efektywności przetwarzania

W celu zebrania informacji o efektywności przetwarzania użyliśmy programu Intel VTune. Głównym wykorzystywanym przez nas trybem był *Microarchitecture Exploration.* który pozwolił nam na zebranie wyników, zaprezentowanych później w tabeli poniżej. Ponadto, gdy uzyskane wyniki budziły nasze wątpliwości lub gdy chcieliśmy dowiedzieć się, jaka dokładnie przyczyna stoi za uzyskanymi wartościami parametrów, używaliśmy trybów *Hotspots* oraz *Memory Access*, które pozwoliły nam m. in. na zlokalizowanie fragmentów kodu odpowiadających za największą część czasu przetwarzania.

b. tabela z wynikami przetwarzania sekwencyjnego:

	sekwencyjne					
	modulo			sito		
	1 wątek			1 wątek		
MIN=2, MAX=90000000	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
Czas przetwarzania [s]	145,26	65,611	126,562	2,061	1,066	2,017
Instructions retired	4,19E+11	1,58E+09	2,62E+11	3,32E+09	1,71E+09	3,36E+09
Clockticks	3,79E+11	1,41E+09	2,45E+11	5,10E+09	2,57E+09	4,99E+09
Retiring	49,90%	49,30%	49,40%	15,80%	12,30%	15,00%
Front-end bound	48,40%	48,10%	50,40%	3,70%	5,80%	3,30%
Back-end bound	1,20%	2,20%	0,00%	80,20%	80,60%	82,20%
Memory bound	0,00%	0,00%	0,00%	33,30%	31,90%	33,60%
Core bound	1,20%	2,10%	0,00%	46,90%	48,60%	48,60%
Effective physical core utilization	47,70%	48,50%	44,50%	40,70%	32,80%	43,10%
Przyspieszenie	x	x	x	x	x	x
Prędkość przetwarzania	6,20E+05	6,86E+05	3,56E+05	4,37E+07	4,22E+07	2,23E+07
Efektywność	x	x	x	x	x	x

Tabela 1: wyniki przetwarzania sekwencyjnego

MIN=2, MAX=90000000	równoległe																	
	modulo - domenowe						sito - funkcyjne						sito - domenowe					
	2 wątki			4 wątki			2 wątki			4 wątki			2 wątki			4 wątki		
	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
Czas przetwarzania [s]	119,735	44,796	87,24	80,798	34,692	47,342	1,86	0,981	1,718	1,639	0,851	1,703	1,03	0,483	0,601	0,867	0,398	0,569
Instructions retired	4,19E+11	1,58E+11	2,62E+11	4,20E+11	1,58E+11	2,63E+11	2,23E+09	1,15E+09	2,07E+09	2,38E+09	1,45E+09	2,15E+09	4,05E+09	1,72E+09	2,22E+09	4,32E+09	1,90E+09	2,43E+09
Clockticks	4,21E+11	1,59E+11	3,31E+11	5,33E+11	2,06E+11	3,57E+11	7,17E+09	3,72E+09	6,82E+09	1,21E+10	5,44E+09	9,66E+09	4,25E+09	1,84E+09	2,25E+09	5,94E+09	2,37E+09	3,42E+09
Retiring	52,60%	52,70%	58,70%	59,90%	61,80%	62,70%	9,10%	10,20%	9,50%	9,20%	11,90%	8,10%	40,50%	39,90%	39,10%	46,10%	41,70%	45,20%
Front-end bound	46,50%	45,40%	46,40%	39,50%	41,60%	37,90%	5,90%	7,00%	6,10%	7,00%	12,30%	10,00%	35,00%	36,90%	35,60%	34,10%	33,90%	31,00%
Back-end bound	0,70%	1,50%	0,00%	0,20%	0,00%	0,00%	84,30%	82,50%	84,10%	83,50%	74,70%	81,20%	23,40%	22,40%	23,70%	18,40%	21,20%	22,90%
Memory bound	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	36,00%	37,80%	35,20%	35,30%	63,40%	35,10%	6,90%	8,50%	8,20%	6,40%	9,10%	9,00%
Core bound	0,70%	1,50%	0,00%	0,20%	0,00%	0,00%	48,20%	44,70%	48,90%	48,20%	11,30%	46,10%	16,50%	13,90%	15,50%	11,90%	12,10%	13,90%
Effective physical core utilization	67,10%	67,20%	52,60%	86,90%	73,90%	88,80%	58,40%	44,90%	59,40%	78,00%	63,40%	58,60%	60,50%	55,60%	60,60%	70,50%	62,00%	62,70%
Przyspieszenie	1,213	1,465	1,451	1,798	1,891	2,673	1,108	1,087	1,174	1,257	1,253	1,184	2,001	2,207	3,356	2,377	2,678	3,545
Prędkość przetwarzania	7,52E+05	1,00E+06	5,16E+05	1,11E+06	1,30E+06	9,51E+05	4,84E+07	4,59E+07	2,62E+07	5,49E+07	5,29E+07	2,64E+07	8,74E+07	9,32E+07	7,49E+07	1,04E+08	1,13E+08	7,91E+07
Efektywność	0,607	0,732	0,725	0,449	0,473	0,668	0,554	0,543	0,587	0,314	0,313	0,296	1,000	1,104	1,678	0,594	0,670	0,886

Tabela 2: wyniki przetwarzania równoległego

5. Wnioski

O ile nie wspomniano inaczej, wnioski prezentujemy w oparciu o instancję (2...MAX) przy zastosowaniu 1 wątku (sekwencyjne) lub 4 wątków (wersje równoległe)

a. Czas przetwarzania:

- Zastosowane rozwiązania znacznie różniły się między sobą czasem przetwarzania.
- Najwolniejszy okazał się wariant ‘modulo’ - sekwencyjny. W przypadku tego wariantu najdłuższy czas przetwarzania wyniósł aż 145 sekund. Dla porównania, sekwencyjna wersja ‘sita eratostenesa’ okazała się znacznie szybsza, przetwarzanie zakończyło się w nieco ponad 2 sekundy, czyli czas przetwarzania był ponad 70 razy krótszy niż w sekwencyjnej wersji ‘modulo’.
- Zrównoleglona domenowo wersja ‘modulo’ przynosi krótsze czasy przetwarzania niż jej sekwencyjny odpowiednik, ale wyniki nie są wciąż zadowalające. W porównaniu z wersją sekwencyjną, 4-wątkowe

modulo-domenowe charakteryzuje niemal dwukrotnie krótszy czas przetwarzania, jednak w porównaniu z opisanymi niżej wariantami zrównoleglenia, jest to czas bardzo długi.

- Zrównoleglona wersja ‘sita eratostenesa’ - funkcyjna - pozwoliła na uzyskanie 1.25x krótszego czasu niż jej sekwencyjny odpowiednik. Podejście domenowe pozwoliło na uzyskanie około 2,4x krótszego czasu niż w przypadku wersji sekwencyjnej.
- Równoległa, funkcyjna wersja ‘sita eratostenesa’ pozwoliła na uzyskanie 49x krótszego - a wersja domenowa 92x krótszego - czasu przetwarzania niż równoległa wersja modulo.

b. Wykorzystanie struktur wewnętrznych, analiza wartości wskaźnika *retiring*:

- wskaźnik retiring oznacza udział procentowy wykorzystanych zasobów procesora do przetwarzania kodu
- Zauważyliśmy, że zazwyczaj 4-wątkowe przetwarzanie przekładało się na nieco wyższą efektywność wykorzystania struktur wewnętrznych procesora.
- Najwyższe wartości wskaźnika *Retiring* (na poziomie ok. 60%) uzyskaliśmy w podejściu modulo-domenowym 4-wątkowym. Domenowy wariant ‘sita eratostenesa’ odznaczał się niewiele niższym wskaźnikiem retiring niż wersja modulo (rzędu ok. 45%), a wersja funkcyjna charakteryzuje się niskimi wartościami *retiring* (rzędu ok. 10%)
- W przypadku wersji modulo-domenowej wąskim gardłem okazało się *Front-end bound,* czyli w ograniczenie efektywności przetwarzania części wejściowej procesora. Pozostałe wartości wskaźników ‘bound’ były bliskie lub równe zeru
- W przypadku funkcyjnej wersji ‘sita eratostenesa’ mamy do czynienia z silnym zjawiskiem wąskiego gardła na który składają się wartości Back-end bound, Memory bound oraz Core bound. Udało nam się nieco zmniejszyć wartości tych wskaźników poprzez modyfikacje opisane w punkcie 3.e, to znaczy dzięki zastosowania *std::fill* oraz zwolnienia wątku, działającego na fragmencie tablicy wykreśleń, z konieczności dodawania wielokrotności wykreślnej liczby, gdy znajdują się ona poza badanym przedziałem.

- W domenowej wersji ‘sita eratostenesa’ wąskie gardło nie ma tak dużego znaczenia jak w przypadku funkcyjnego wariantu, jednak jest ono zauważalne. Głównym czynnikiem jest Front-end bound, nieco mniej istotnym Back-end bound oraz Core bound, najmniejszy udział w ograniczeniu efektywności przetwarzania ma Memory bound.
- Staraliśmy się zmniejszyć wartości wskaźników oznaczających ograniczenie efektywności przetwarzania między innymi stosując różne typy podziału zadań między wątki (guided, static, dynamic), różne wielkości bloków, które otrzymują poszczególne wątki; wartości podane w tabeli to najlepsze spośród tych, jakie zdołaliśmy uzyskać
- Największą wartością wskaźnika retiring wykazał się wariant modulo-domenowy (ok. 60%), wariant sita eratostenesa - domenowego wykazał się wskaźnikiem *retiring* na poziomie około 45%, najgorsza efektywność (ok. 10%) cechowała wariant sita funkcyjnego.

c. Zrównoleglenie:

- Najwyższe wartości zarówno przyspieszenia, jak i efektywności przetwarzania obserwujemy dla wariantu równoległego sita domenowego.
- Dla opisywanej instancji (2...MAX) przy zastosowaniu 1 wątku (sekwencyjne) lub 4 wątków (wersje równoległe), wspomniany wariant wykazał się 1,9x większym przyspieszeniem niż funkcyjny wartian sita oraz 1,3x większym przyspieszeniem niż równoległa wersja modulo-domenowa.
- Wariant równoległego sita domenowego okazał się lepszy również pod kątem efektywności: jest on 1,9x bardziej efektywny niż wariant równoległego sita funkcyjnego oraz 1,3x bardziej efektywny niż równoległe modulo domenowe.
- Wartości Effective Physical core utilization okazały się najwyższe w wariancie modulo-domenowym (w ‘sicie’ funkcyjnym i domenowym były odpowiednio 1,11x oraz 1,23 raza mniejsze). Wysoka wartość wspomnianego wskaźnika nie przełożyła się na uzyskanie lepszego czasu przetwarzania niż inne równoległe warianty, które uzyskały niższą wartość tego wskaźnika. Dochodzimy do wniosku, że efektywne/zrównoważone wykorzystanie rdzeni fizycznych nie zawsze doprowadza do obliczeń o najkrótszym czasie.

d. Prędkość przetwarzania:

- W każdym z badanych wariantów prędkość przetwarzania była większa w przypadku zastosowania 4 procesorów, niż w przypadku zastosowania 2 procesorów.
- Zdecydowanie największą prędkością przetwarzania wykazał się wariant równoległego sita domenowego. Dla każdej z instancji prędkość przetwarzania w tym wariancie jest lepsza niż w pozostałych podejściach, jednak w obrębie jednego wariantu prędkość różni się w zależności od instancji. Największą prędkością cechuje się przetwarzanie liczb w przedziale 2...MAX/2, w wariancie równoległego sita domenowego jest ono 1,09x szybsze niż przetwarzanie 2...MAX oraz 1,43x szybsze niż przetwarzanie liczb z przedziału MAX/2...MAX. Bardzo podobne zależności obserwujemy również w przypadku pozostałych wariantów zrównoleglenia badanego problemu.

e. Efektywność:

- najbardziej efektywne okazało się podejście ‘równoległe sito - domenowe’. Uważamy, że jest to zasługa ograniczenia liczby iteracji potrzebnych przy wykreślaniu liczb, opisana w punkcie 3.e tego sprawozdania. Dla opisywanej instancji ten wariant okazał się 1,89x bardziej efektywny niż funkcyjna wersja zrównoleglenia sita oraz 1,23x bardziej efektywny niż domenowa wersja zrównoleglenia wariantu modulo.
- zauważyliśmy, że efektywność skaluje się ze wzrostem liczby procesorów: efektywność przetwarzania zazwyczaj była wyższa w przypadku zastosowania 2 procesorów niż w przypadku użycia 4 procesorów. Najprawdopodobniej jest to koszt współdzielenia pracy między wątkami, który jednak staraliśmy się zminimalizować stosując odpowiednie modyfikacje kodu oraz właściwy podział pracy między wątkami.

f. Ograniczenia efektywnościowe:

- W równoległym wariancie modulo - domenowym dominuje ograniczenie *front-end* (ok. 45%), to znaczy pochodzące od części wejściowej procesora
- W równoległym wariancie sita - funkcyjnym dominuje ograniczenie *back-end* (ok. 84%), czyli ograniczenie przetwarzania części wyjściowej procesora. Nieco mniej znaczące, jednak zauważalne, jest również Core Bound (ok.

48%, ograniczenie jednostek wykonawczych) oraz Memory Bound (35%, ograniczenie ograniczenie systemu pamięci)

- W wariancie domenowy sita eratostenesa dominują ograniczenia front-end (34%), poza tym widoczne jest ograniczenie back-end (ok. 18%), a także memory oraz core bound, oba o wartości około 10%.

6. Tabela podsumowująca:

W celu ułatwienia analizy zawartości tabel zastosowaliśmy formatowanie warunkowe. Kolor biały oznacza najmniejsze prędkości przetwarzania, kolor ciemnoniebieski - największe.

	sekwencyjne					
	modulo			sito		
	1 wątek			1 wątek		
MIN=2, MAX=90000000	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
Prędkość przetwarzania	6,20E+05	6,86E+05	3,56E+05	4,37E+07	4,22E+07	2,23E+07

Tabela 3: Podsumowanie prędkości przetwarzania sekwencyjnego

	równoległe																	
	modulo - domenowe						sito - funkcyjne						sito - domenowe					
	2 wątki			4 wątki			2 wątki			4 wątki			2 wątki			4 wątki		
MIN=2, MAX=90000000	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
Prędkość przetwarzania	7,52E+05	1,00E+06	5,16E+05	1,11E+06	1,30E+06	9,51E+05	4,84E+07	4,59E+07	2,62E+07	5,49E+07	5,29E+07	2,64E+07	8,74E+07	9,32E+07	7,49E+07	1,04E+08	1,13E+08	7,91E+07

Tabela 4: Podsumowanie prędkości przetwarzania równoległego