Projekt 1 OMP

Laboratorium z przetwarzania równoległego

Adrian Piniek 144226

Dawid Królak 145383

Grupa L11, środa 9.45, tygodnie nieparzyste

Politechnika Poznańska

Informatyka, WIiT, semestr VI

Wymagany termin oddania sprawozdania: 11.05.2022 r.

Rzeczywisty termin oddania sprawozdania: 18.05.2022 r.

Wersja sprawozdania: 1.0

Opis zadania: Badanie efektywności przetwarzania równoległego w problemie znajdowania liczb pierwszych za pomocą Sita Eratostenesa w wariancie funkcyjnym i domenowym.

Kontakt:

adrian.piniek@student.put.poznan.pl

dawid.g.krolak@student.put.poznan.pl

Wykorzystany system obliczeniowy

Procesor Intel® Core™ i5-8300H

- Liczba rdzeni: 4
- Liczba wątków: 8
- Pamięć Cache: 8 MB (technologia Intel Smart Cache)
- Bazowa częstotliwość: 2,30 GHz
- Maksymalna częstotliwość Turbo: 4 GHz

System operacyjny: Manjaro Linux x86_64

Prezentacja wariantów kodu

Ze względu na charakterystykę problemu postanowiliśmy wykorzystać typ danych ‘unsigned long long int’, aby móc pomieścić rozpatrywane liczby, jednocześnie pozbywając się liczb ujemnych, które nie są w ogóle brane pod uwagę w problemie.

```
#define llint unsigned long long int
```

Kod 1. Definicja typu danych

Koncepcja prosta sekwencyjna (PS) :

Sprawdzenie każdej liczby w przedziale MIN .. MAX czy spełnia warunki bycia liczbą pierwszą.

Pierwszym etapem jest wyszukanie wszystkich liczb pierwszych mniejszych niż pierwiastek górnej granicy rozpatrywanego przedziału za pomocą najbardziej prymitywnej metody, a więc sprawdzania czy liczba x jest podzielna przez jakąkolwiek liczbę z przedziału 2 .. sqrt(x).

```
llint getPrimesLowerThanSqrtOfNumber(llint x, llint *primes) {
    llint sqrtOfNum = (int)sqrt(x);

    primes[0] = 2;
    int numOfPrimes = 1;
    for (llint i = 3; i <= sqrtOfNum; i+=2)
    {
        if(checkPrimeBasic(i)) {
            primes[numOfPrimes] = i;
            // printf("%lld\n", i);
            numOfPrimes++;
        }
    }

    return numOfPrimes;
}
```

Kod 2. Algorytm pierwszej iteracji

Zebrane w pierwszym etapie liczby pierwsze wykorzystujemy do szukania kolejnych w pozostałej części przedziału (MIN, MAX), tym razem sprawdzając podzielność tylko przez wyszukane już wcześniej liczby pierwsze.

```
int checkPrime(llint x, llint *primes, llint numOfPrimes) {
    llint sqrtX = (llint)sqrt(x);
    if(x == 1) {
        return 0;
    }
    if(x % 2 == 0) {
        return 0;
    }
    for (llint i = 0; i < numOfPrimes; i++)
    {
        if(primes[i] > sqrtX) {
            break;
        }
        if(x % primes[i] == 0) {
            return 0;
        }
    }
    return 1;
}
```

Kod 3. Zliczanie liczb pierwszych w zadanym przedziale liczbowym

Koncepcja prosta równoległa (PR):

Wstęp programu jest identyczny, ponieważ operacja pierwszego przeszukiwania jest prosta obliczeniowo, przez co lepiej nie wdrażać równoległości, która to mogłaby niekorzystnie wpłynąć na efektywność.

Zrównoleglenie programu następuje w momencie przeszukiwania pozostałej części zbioru. Podział wykonywania tej pętli na wątki pozwolił na około 3-krotne przyspieszenie w przypadku pracy na czterech procesorach i niemalże 6-krotne w przypadku ośmiu wątków.

```
#pragma omp parallel for
for (llint i = min; i <= max; i++)
{
    if(checkPrime(i, primes, numOfPrimesForSearching)) {
        isPrime[i-min] = 1;
    }
    else {
        isPrime[i-min] = 0;
    }
}
```

Kod 4. Dodanie dyrektywy równoległościowej i zapisywanie wyniku.

Koncepcja Sita Eratostenesa (SE):

Metoda polegająca na usuwaniu wielokrotności kolejnych liczb z tablicy w kolejności rosnącej. Sito Eratostenesa ma postać tablicy liczb, w której dla każdej liczby ***i*** przypisujemy wartość 1 jeśli jest liczbą pierwszą i 0 jeśli nią nie jest. Sito inicjujemy przypisując 1 do każdej liczby (oprócz zera i jedynki, które z definicji nie są liczbami pierwszymi), następnie iterując w przedziale 2 .. MAX "odsiewamy" wielokrotności kolejnych liczb przypisując im 0 w sicie.

```
llint min = atoll(argv[1]);
llint max = atoll(argv[2]);
llint maxSqrt = (llint)sqrt(max);
llint *primes = malloc(sizeof(llint) * max);
llint numOfPrimes = 0;
llint numOfPrimesInFirstRow = 0;
double startTime;
double stopTime;
//startTime = omp_get_wtime();
char* sieve = malloc(sizeof(char) * max);
for (llint i = 2; i < max; sieve[i++]=1);
```

Kod 5. Inicjalizacja algorytmu Eratostenesa wraz z wypełnieniem sita.

Sito Eratostenesa sekwencyjnie (SES):

```
for(llint i = 2; i <= maxSqrt; i++) {
    if(sieve[i] == 1) {
        primes[numOfPrimesInFirstRow] = i;
        numOfPrimesInFirstRow++;
        for (llint j = 2*i; j <= maxSqrt; j += i)
        {
            sieve[j] = 0;
        }
    }
}
```

Kod 6. Przesiewanie sita dla $\sqrt{max}$ liczb przedziału w pierwszej iteracji

W pierwszym etapie, podobnie jak w wersji prostej przesiewamy tylko liczby z zakresu 2 .. sqrt(MAX). Zebrane w ten sposób liczby służą optymalizacji następnego etapu, w którym odsiewamy już tylko wielokrotności wcześniej znalezionych liczb pierwszych.

```
for (llint i = 0; i < numOfPrimesInFirstRow; i++)
{
    for (llint j = 2; j*primes[i] <= max; j++)
    {
        sieve[j*primes[i]] = 0;
    }

}
```

Kod 7. Odsiewanie dla liczb obliczonych w pierwszej iteracji

Sito Eratostenesa równolegle funkcyjnie (SERF):

```
#pragma omp parallel for
for (llint i = 0; i < numOfPrimesInFirstRow; i++)
{
    for (llint j = 2; j*primes[i] <= max; j++)
    {
        sieve[j*primes[i]] = 0;
    }

}
```

Kod 8. Druga iteracja sita z pragmatem omp

Metoda jest zbliżona do działania wersji sekwencyjnej, jednak po obliczeniu liczb w pierwszym etapie na jednym procesorze pozostałe są już uruchamiane na wielu wątkach. Samo dodanie dyrektywy parallel spowodowało jednak niewielki przyrost przyspieszenia obliczeń (około 5%). Dopiero dodanie do pragmatu dyrektywy schedule(dynamic) zaowocowało zauważalnym już przyspieszeniem działania programu o około 20% w stosunku do wersji sekwencyjnej.

```
#pragma omp parallel for schedule(dynamic)
for (llint i = 0; i < numOfPrimesInFirstRow; i++)
{
    for (llint j = 2; j*primes[i] <= max; j++)
    {
        sieve[j*primes[i]] = 0;
    }

}
```

Kod 9. Finalna wersja wielowątkowego funkcyjnego przesiewania sita

Sito Eratostenesa równolegle domenowe (SERD):

Podejście to charakteryzuje się wykorzystywaniem bloków liczbowych aby zmniejszyć stopień złożoności wykreślania. Każdy wątek analizuje tylko swój przedział wykreślając wielokrotności liczb tak jak w podstawowym sicie.

```
llint blockSize = 100000;
llint blockStart;
llint blockRange;
llint blockEnd;
llint startFrom;

#pragma omp parallel for schedule(dynamic)
for (llint k = min/blockSize; k <= max/blockSize; k++)
{
    blockStart = k*blockSize > 2 ? k*blockSize : 2;
    blockRange = blockStart+blockSize;
    blockEnd = max < blockRange ? max : blockRange;

    for (llint i = 0; i < numOfPrimesInFirstRow; i++)
    {
        startFrom = blockStart % primes[i] == 0 ? blockStart/primes[i] : blockStart/primes[i] + 1;
        if(startFrom < 2) {
            startFrom = 2;
        }

        for (llint j = startFrom; j*primes[i] <= blockEnd; j++)
        {
            sieve[j*primes[i]] = 0;
        }

    }
}
```

Kod 10. Sito równoległe domenowe

Po kilku testach efektywności i zaznajomieniu się z tematem Intel Smart Cache ustaliliśmy wielkość bloku na sto tysięcy, pozwalając na odpowiednio równomiernie dzielenie dużych liczb zakresowych i jednocześnie pozwalając zachować pamięć dla reszty programu. Zarówno użycie mniejszych jak i większych wielkości bloków nie dawało dobrych rezultatów, a w niektórych przypadkach wręcz pogarszało efektywność programu.

Zagadnienie podziału pracy

Zagadnienie to najlepiej opisuje program sita funkcyjnego, gdzie najlepszą metodą jest przyznanie każdemu wątkowi po jednym zadaniu, aby jak najbardziej ograniczyć problem zbyt dużego przydziału. Jedno zadanie oznacza jedną liczbę pierwszą do odkrycia.

Metoda domenowa polega na podzielenie problemu na bloki i przydzieleniu ich odpowiednio wątkom. Wielkość tych bloków zależy od procesora który jest zainstalowany na komputerze. Posiadając 8 MB pamięci podręcznej i 8 wątków, uznaliśmy przyznanie 100 000 liczb na blok, wypełniając tymże około 10% pamięci Cache.

Zagadnienie przydziału zadań do procesów

W rozwiązaniu funkcyjnym przydział odbywa się dynamicznie, po skończonej jednej pętli wątek otrzymuje kolejny problem. Nie wykonuje bloku z góry już mu przypisanego, tylko otrzymuje nowe jeszcze dostępne zadanie po zakończeniu poprzedniego. Czas wykonywania każdego zadania jest zależny od stopnia skomplikowania liczby jaką otrzymał i może się diametralnie różnić pomiędzy wątkami, co może doprowadzić do nieefektywnej synchronizacji i oczekiwań na zadania.

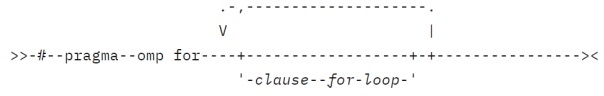
Podejście domenowe dzieląc bloki zakresowe pozwala na równomierne rozłożenie czasu wykonywania zadań. Mimo tego iż dalsze bloki posiadają większe liczby, nie skutkuje to zbytnim zwiększeniem czasu przetwarzania, ponieważ ilość operacji pozostaje ta sama.

Podział przetwarzania na zadania

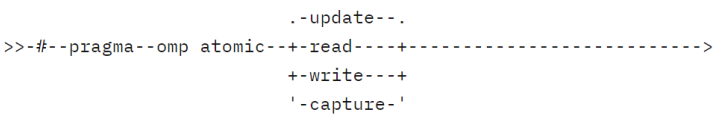
Zdecydowaliśmy się na rozłożenie po jednym zadaniu każdemu wątkowi, aby ułatwić zrozumienie działania programu. Dlatego też w algorytmach prostych 1 wątek oblicza jedną liczbę. W wersji funkcyjnej zadaniem jest wykreślenie wielokrotności jednej liczby ze zbioru, natomiast domenowa wersja problemu wskazuje jedno zadanie jako wykreślenie wszystkich liczb pierwszych z nadanego podzbioru.

Dyrektywy OMP

- #pragma omp parallel for - wykorzystuje statyczny podział liczby iteracji w pętli i rozdziela je pomiędzy dostępne wątki.



- #pragma omp parallel for schedule(dynamic) - Pętla for jest dzielona dynamicznie aby wątki mogły się ubiegać ciągle o nowe dane do liczenia, po obliczeniu wpierw przypisanych. Dyrektywa ta znacząco przyśpiesza wykonywanie obliczeń, ponieważ pierwsze wątki nie ‘zasypiają’ po skończeniu swojej części, która to jest z reguły mniej złożona czasowo, tylko ciągle dostarczane są nowe zadania od łatwych do trudnych.
- #pragma atomic - (użyty w pierwszych wersjach kodu, wersja 1.0 niewykorzystywany) eliminuje problem wyścigu, pozwalając na automatyczny dostęp do konkretnych miejsc pamięci, blokując inne wątki przed wpisaniem się na zadany blok.



- omp_set_num_threads(unsigned int) - wskazuje ile wątków ma zostać wykonanych dla bloku kodu

Problem poprawnościowy:

- Wyścig - problem występujący w przypadku gdy wątek ubiega się o pamięć, w czasie kiedy przynajmniej jeden z pozostałych wątków wykonuje zapis (np. do tablicy)

Potencjalne problemy obliczeniowe:

- False sharing - (pl. fałszywe udostępnianie) występuje w sytuacji gdy wątek niepotrzebnie wymaga odświeżenia bloku pamięci w którym nie występuje taka konieczność, przez wątek który aktualnie modyfikuje dany blok pamięciowy.
- Synchronizacja - Nasze programy nie wymagają użycia instrukcji synchronizacyjnych. Jedynie gdzie faktycznie można by wykorzystać tę metodę jest SERF, dzięki posłużeniu się wielowątkowością dla przedziału już pierwiastkowego. Jednak może to powodować i wymuszenie na wątku, aby czekał na zadaną liczbę pierwszą do wykreślania.
- Brak zrównoważenia procesora obliczeniami - występuje w przypadku nieprawidłowego przypisania wątków do zadawanych prac. Problem ten pozwala rozwiązać dyrektywa ‘dynamic’.

Prezentacja wyników

Rodzaje badanych algorytmów i ich efektywność najlepiej opisuje poniższa tabela z wynikami.

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)
Algorym prosty	1	2 ... 200 000 000	116,852	185,191,400,000	445,468,600,000	41,80%	56,90%	0,80%
		2 ... 100 000 000	46,078	72,785,800,000	174,689,600,000	42,50%	58,30%	0,00%
		100 mln ... 200 mln	69,793	112,398,700,000	267,595,800,000	41,80%	57,40%	0,50%
	4	2 ... 200 000 000	37,925	185,368,500,000	445,038,500,000	42,80%	56,40%	0,20%
		2 ... 100 000 000	14,781	72,870,900,000	174,510,200,000	43,50%	57,10%	0,00%
		100 mln ... 200 mln	19,512	112,518,300,000	272,727,100,000	44,00%	57,00%	0,00%
	8	2 ... 200 000 000	22,535	185,508,800,000	468,077,600,000	65.4%	34,90%	0,00%
		2 ... 100 000 000	8,673	72,937,600,000	184,736,000,000	67.0%	32.6%	0.0%
		100 mln ... 200 mln	11,847	112,697,700,000	287,803,600,000	73.5%	30.0%	0.0%
Sito Eratostenesa (sekwencyjnie)	1	2 ... 2 mld	28,322	59,889,700,000	107,196,100,000	13.8%	4.9%	79.4%
		2 ... 1 mld	13,066	28,499,300,000	49,723,700,000	13.5%	5.4%	79.2%
		1 mld ... 2 mld	26,261	48,127,500,000	99,652,100,000	11.3%	3.4%	84.1%
Sito Eratostenesa (funkcyjnie)	4	2 ... 2 mld	25,288	63,972,200,000	297,353,200,000	8.6%	3.9%	85.6%
		2 ... 1 mld	11,085	28,676,400,000	144,762,000,000	7.4%	3.7%	87.0%
		1 mld ... 2 mld	21,406	45,797,600,000	285,250,600,000	5.8%	3.2%	90.0%
	8	2 ... 2 mld	22,046	57,667,900,000	401,133,800,000	6.0%	3.5%	89.2%
		2 ... 1 mld	11,254	28,807,500,000	182,178,400,000	7.4%	4.1%	86.7%
		1 mld ... 2 mld	21,149	46,052,900,000	398,498,000,000	5.1%	3.3%	91.1%
Sito Eratostenesa (domenowo)	4	2 ... 2 mld	4,709	58,553,400,000	31,077,600,000	62.6%	17.7%	7.1%
		2 ... 1 mld	2,382	29,088,100,000	15,506,600,000	72.4%	18.3%	2.0%
		1 mld ... 2 mld	2,833	30,923,500,000	17,245,400,000	50.3%	17.4%	21.9%
	8	2 ... 2 mld	5,048	60,046,100,000	39,166,700,000	55.5%	17.5%	15.0%
		2 ... 1 mld	2,277	29,118,000,000	18,579,400,000	56.2%	18.1%	13.6%
		1 mld ... 2 mld	2,614	30,447,400,000	20,357,300,000	51.2%	17.1%	18.6%

Tabela 1. Pierwsze sześć parametrów dla finalnych wersji.

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
Algorym prosty	1	2 ... 200 000 000	0,30%	0,60%	23,00%	Nie dotyczy	1,71E+06	Nie dotyczy
		2 ... 100 000 000	0,00%	0,00%	22,50%	Nie dotyczy	2,17E+06	Nie dotyczy
		100 mln ... 200 mln	0,10%	0,30%	23,80%	Nie dotyczy	1,43E+06	Nie dotyczy
	4	2 ... 200 000 000	0,10%	0,20%	69,90%	3,0811	5,27E+06	0,770275
		2 ... 100 000 000	0,00%	0,00%	69,00%	3,1174	6,77E+06	0,77935
		100 mln ... 200 mln	0,00%	0,00%	81,10%	3,5769	5,13E+06	0,894225
	8	2 ... 200 000 000	0,00%	0,00%	81,10%	5,1854	8,88E+06	0,648175
		2 ... 100 000 000	0,00%	0,00%	76,10%	5,3128	1,15E+07	0,6641
		100 mln ... 200 mln	0,00%	0,00%	84,50%	5,8912	8,44E+06	0,7364
Sito Eratostenesa (sekwencyjnie)	1	2 ... 2 mld	28.3%	51.1%	23,00%	Nie dotyczy	7,06E+07	Nie dotyczy
		2 ... 1 mld	29.4%	49.8%	23,30%	Nie dotyczy	7,65E+07	Nie dotyczy
		1 mld ... 2 mld	30.1%	54.1%	23,10%	Nie dotyczy	3,81E+07	Nie dotyczy
Sito Eratostenesa (funkcyjnie)	4	2 ... 2 mld	33.6%	52.0%	42,90%	1,12	7,91E+07	0,28
		2 ... 1 mld	35.5%	51.5%	52,20%	1,1787	9,02E+07	0,294675
		1 mld ... 2 mld	36.1%	53.8%	50,70%	1,2268	4,67E+07	0,3067
	8	2 ... 2 mld	36.2%	53.0%	60,00%	1,2847	9,07E+07	0,1605875
		2 ... 1 mld	36.9%	49.8%	53,90%	1,161	8,89E+07	0,145125
		1 mld ... 2 mld	36.2%	54.8%	61,40%	1,2417	4,73E+07	0,1552125
Sito Eratostenesa (domenowo)	4	2 ... 2 mld	2.0%	5.1%	35,80%	6,0144	4,25E+08	1,5036
		2 ... 1 mld	0.6%	1.5%	35,50%	5,4853	4,20E+08	1,371325
		1 mld ... 2 mld	6.0%	15.8%	34,00%	9,2697	3,53E+08	2,317425
	8	2 ... 2 mld	4.9%	10.1%	32,90%	5,6105	3,96E+08	0,7013125
		2 ... 1 mld	4.1%	9.5%	34,90%	5,7383	4,39E+08	0,7172875
		1 mld ... 2 mld	5.7%	12.9%	33,70%	10,0463	3,83E+08	1,2557875

Tabela 2. Dalsze finalne wyniki programów.

Powyższe dane zostały zebrane za pomocą programu Intel® VTune™. Program ten jest wykorzystywany do badania wydajności i wykrywania tzw. “wąskich gardeł” (ang. *hotspots*) w aplikacjach wykorzystujących architekturę mikroprocesorową.

VTune wykorzystuje jednostkę monitorującą wydajność procesora aby wskazać niewykorzystane ścieżki, braki w pamięci cache (cache misses), przestoje w pracy rdzeni, etc.. Do badania takich zdarzeń wykorzystywany jest Event Based Sampling. Kiedy licznik zdarzeń osiągnie określony

poziom (SAV - *sample-after value*) następuje przerwanie i zaczyna się analizowanie danych w programie i przypisanie ich do odpowiedniej instrukcji.

Program zbiera wiele statystyk na temat analizowanego kodu, pozwala wykryć mało wydajne miejsca i pokazuje jaki procent możliwości architektury komputera zostało wykorzystane w czasie analizy.

Wnioski

Dzięki dzieleniu problemu na bloki najbardziej wydajnym programem okazał się SERD (maksymalnie 5 sekund dla 2 mld liczb). Względem wersji sekwencyjnej, podejście domenowe w najlepszym przypadku okazuje się nawet 10 razy szybsze w wersji działającej na ośmiu wątkach. Metoda ta zdecydowanie pokonuje pozostałe sposoby otrzymywania liczb pierwszych, którym nie udało się zejść poniżej 10 sekund.

Fragmentem dla kodów równoległych spowalniającym szybkość całkowitą programu może być pierwsza iteracja wykonywana sekwencyjnie dla wszystkich wersji, jednak próby jej zrównoleglenia jedynie wydłużają czas trwania algorytmu, bądź zwracają błędne wyniki.

Najgorsze efekty pod kątem przetwarzania równoległego posiada SERF, wymagając ciągłych dostępów do pamięci oraz tablicy niemieszczącej się w pamięci CACHE dając tak niskie wartości dla efektywności przetwarzania równoległego. Podejście to nie spowodowało znacznej poprawy efektywności względem wersji jednowątkowej, ucinając jedynie około 1 do 2 sekund (w skali 11 .. 28 sekund)

Instancją, która najefektywniej wykorzystała wielowątkowość okazał się 4 wątkowy SERD obliczający w przedziale 1:2 miliarda. Osiągając ponad 9-krotne przyspieszenie algorytmu równoległego względem jednowątkowego oraz 34% wykorzystania fizycznego procesora.

Co ciekawe, dla wszystkich algorytmów najbardziej efektywne okazało się zrównoleglenie na 4 procesorach fizycznych, co prawdopodobnie wynika z nadmiernego podziału pracy w przypadku użycia 8 wątków.

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)
Algorytm prosty	4	100 mln ... 200 mln	19,512	112,518,300,000	272,727,100,000	44,00%	57,00%	0,00%
Sito Eratostenesa (funkcyjnie)	4	1 mld ... 2 mld	21,406	45,797,600,000	285,250,600,000	5.8%	3.2%	90.0%
Sito Eratostenesa (domenowo)	4	1 mld ... 2 mld	2,833	30,923,500,000	17,245,400,000	50.3%	17.4%	21.9%

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
Algorytm prosty	4	100 mln ... 200 mln	0,00%	0,00%	81,10%	3,5769	5,13E+06	0,894225
Sito Eratostenesa (funkcyjnie)	4	1 mld ... 2 mld	36.1%	53.8%	50,70%	1,2268	4,67E+07	0,3067
Sito Eratostenesa (domenowo)	4	1 mld ... 2 mld	6.0%	15.8%	34,00%	9,2697	3,53E+08	2,317425

Tabela 3. Najlepsze wyniki przetwarzań