

Sprawozdanie z Projektu Pierwszego

Laboratorium Przetwarzania Równoległego

Piotr Tylczyński

L7 / 141331

Środa, 11:45

`piotr.tylczynski@student.put.poznan.pl`

Zuzanna Rękawek

L7 / 141304

Środa, 11:45

`zuzanna.rekawek@student.put.poznan.pl`

Oddane: 29.04.2021

Deadline: 29.04.2021

Wersja 1

Contents

1	Motywacja	2
2	Uwagi Ogólne	2
3	Specyfikacja platformy uruchomieniowej	2
4	Problemy i zjawiska występujące w programach równoległych	2
4.1	Problemy Poprawnościowe	2
4.1.1	Zjawisko wyścigu	2
4.2	Problemy Efektywnościowe	3
4.2.1	Problem False Sharing	3
5	Zastosowane Algorytmy	4
5.1	Opis teoretyczny	4
5.1.1	Sito Erastotenesa	4
5.1.2	Przegląd Listy Liczb	4
5.1.3	Określanie pierwszości liczb	4
5.2	Realizacja praktyczna	5
5.2.1	Wykorzystane klauzule OMP	5
5.2.2	Sekwencyjne Określanie Pierwszości Liczb	5
5.2.3	Sekwencyjny Pełny Przegląd Listy Liczb - SPPLL	5
5.2.4	Sekwencyjne Sito Erastotenesa - SSE	6
5.3	Równoległy Pełny Przegląd Listy Liczb	6
5.3.1	Równoległe Sito Erastotenesa - wersja domenowa	9
5.3.2	Równoległe Sito Erastotenesa - wersja funkcyjna	12
5.3.3	Sposób Podziału pracy w programach	16
6	Prezentacja i Omówienie eksperymentu obliczeniowo-pomiarowego	16
6.1	Metodyka wykonania eksperymentów	16
6.2	Charakterystyka Programu Testującego	17
6.3	Wyniki Uzyskane Dla Algorytmów Sekwencyjnych	17
6.4	Wyniki Uzyskane Dla Algorytmów Równoległych	17
6.4.1	Tablica Zbiorcza wszystkich uzyskanych wyników	19
6.5	Analiza Udziału Poszczególnych Procesorów	19
6.5.1	Sekwencyjny Pełny Przegląd Listy Liczb - SPPLL	20
6.5.2	Sekwencyjne Sito Erastotenesa - SSE	21
6.5.3	Równoległe Naiwne Sito Erastotenesa wersja Funkcyjna	23
6.5.4	Równoległe Optymalne Sito Erastotenesa wersja Funkcyjna	26
6.5.5	Równoległe Optymalne Sito Erastotenesa wersja Domenowa	29
6.5.6	Równoległe Naiwne Pełne Przeglądanie Listy Liczb	32
6.5.7	Równoległe Optymalne Pełne Przeglądanie Listy Liczb	35
7	Wnioski	37

1 Motywacja

Celem niniejszego projektu jest stworzenie efektywnego programu wyszukującego liczby pierwsze w zadanym przedziale. W tym celu wykorzystamy programowanie równoległe. Pozwoli to na efektywniejsze wykorzystanie zasobów komputerowych jakimi dysponujemy. W wyniku otrzymamy program mogący wykorzystywać do 100% mocy obliczeniowej procesora komputera, na którym zostanie uruchomiony. Pozwoli to nam na znaczącą redukcję czasu wykonania programu względem standardowej wersji sekwencyjnej programu.

W rozwiązaniu wykorzystamy algorytm Sita Erastotenesa (SE). Jest to algorytm pozwalający na osiągnięcie znaczącego przyspieszenia, po jego zrównolegleniu. Dodatkowo rozważymy wykorzystanie i zrównoleglanie algorytmu Pełnego Przeglądu List Liczb (PPLL).

2 Uwagi Ogólne

Niestety po zakończeniu pisania tego sprawozdania zdaliśmy sobie sprawę z pewnego błędu jaki wkradł się podczas tworzenia i eksperymentowania z poszczególnymi algorytmami. Niestety pomyliliśmy ze sobą nazwy podejścia domenowego i funkcyjnego. Z tego powodu każde wystąpienie słowa *domenowy* powinno zostać zastąpione słowem *funkcyjny* i na odwrót.

3 Specyfikacja platformy uruchomieniowej

Procesor Intel Core i5-9300H

Procesorów Fizycznych 4

Procesorów Logicznych 8

Pamięć Cache 8 MB Intel® Smart Cache

System Operacyjny Windows 10 Pro 20H2

IDE Visual Studio 2019

Oprogramowanie Testujące Intel® VTune™ Profiler

4 Problemy i zjawiska występujące w programach równoległych

4.1 Problemy Poprawnościowe

4.1.1 Zjawisko wyścigu

Jest to zjawisko polegające na ubieganiu się o dostęp do pojedynczego zasobu przez wiele wątków. Jest to warunek konieczny wystąpienia w.w. zjawiska.

Jeżeli nie zastosujemy, żadnego systemu zarządzania dostępem okaże się, że wynik wykonania programu nie będzie deterministyczny. Będzie to spowodowane przez używanie jednego zasobu przez wiele wątków. Będą one potencjalnie zapisywać do zasobu równocześnie. Spowoduje to, że wartość zapisana będzie w najlepszym przypadku wartością zapisaną przez ostatni wątek. W gorszym przypadku powstanie losowa kombinacja wartości zapisywanych przez poszczególne wątki. Kolejność informacji zapisanych w zasobie będzie zależała od czasu dostępu, stąd nazwa wyścig.

Oczywiście występowanie zjawiska wyścigu nie jest porządana. Może ona doprowadzić do powstania złych wyników. W najgorszym przypadku będą one wypadkową, lub permutacją poprawnych wyników.

Rozwiązaniem problemu wyścigu jest wprowadzenie ograniczeń w dostępie do współdzielonego zasobu. W najprostszy sposób można to osiągnąć blokując zasób w czasie jego wykorzystywania przez wątek. Wtedy pozbedziemy się problemu wielodostępu i nigdy nie dopuścimy do spełnienia warunku koniecznego. Narzędziem, które może okazać się przydatne mogą być monitory, lub semaforey. W naszym projekcie zastosujemy odpowiednie klauzule, które zapobiegają opisywanemu zjawisku. Będą to klauzule synchronizujące i dzielące prace między wątkami, które zostały szczegółowo opisane poniżej, jaki i w opisach samych algorytmów.

4.2 Problemy Efektywnościowe

4.2.1 Problem False Sharing

Polega na unieważnieniu potencjalnie nie współdzielonych danych w pamięci podręcznej procesora. Dzieje się tak, ponieważ leżą one na tej samej linii adresowej procesora. Z tego powodu jeżeli procesor wykonuje zapis do jednej z komórek pamięci to może się okazać, że narusza inną zmienną leżącą na tej samej linii pamięci. W takim wypadku, zmienna ta zostanie uznana za "*brudną*" i będzie wymagała ponownego pobrania z pamięci.

Zjawisko False Sharing prowadzi do znacznego spadku efektywności programu. W skrajnych przypadkach może okazać się, że program jest znacząco wolniejszy od swojej wersji sekwencyjnej. Jest to bezpośrednie następstwo wielokrotnego i niepotrzebnego unieważniania linii pamięci. Co sprawia, że wymagany jest dodatkowy narzut czasowy związany z transferem danych pomiędzy poszczególnymi poziomami pamięci komputera.

Rozwiązaniem problemu False Sharingu jest odpowiednia separacja przestrzenna danych. W przypadku tabel można to zapewnić przez umieszczenie wolnych miejsc pomiędzy danymi do zapisu. W ten sposób dane nie będą leżały na tej samej linii pamięci.

W wyniku eksperymentu przeprowadzonego w ramach zadania pierwszego wyznaczyliśmy wielkość pojedynczej linii pamięci na 64B

5 Zastosowane Algorytmy

5.1 Opis teoretyczny

5.1.1 Sito Erastotenesa

W swojej najprostszej formie SE jest algorytmem, który przyjmuje na swoje wejście wektor liczb naturalnych, uporządkowanych rosnąco z krokiem jeden - w dalszych częściach tego dokumentu nazywanych Wektorem Liczb Uporządkowanych - (WLU). Zastosowanie WLU pozwala na dokonanie pewnej optymalizacji. Polega ona na sprawdzaniu tylko liczb znajdujących się przed połową takiego wektora. Optymalizacja taka jest możliwa, ponieważ WLU jest rosnąco uporządkowany, więc wiemy, że wszystkie liczby złożone w drugiej połowie są, wielokrotnością, liczb znajdujących się w pierwszej połowie. Biorąc pod uwagę sposób działania algorytmu, który wykreśla z WLU wszystkie wielokrotności liczb, mamy pewność, że po przejrzaniu wszystkich liczb z pierwszej połowy, wyeliminowaliśmy wszystkie pozostałe z drugiej połowy.

Sam sposób działania SE jest prosty. Dla każdej znalezionej liczby w WLU wykreśl z WLU wszystkie jej wielokrotności - w ten sposób pozbywamy się liczb złożonych. Następnie przejdź do kolejnej liczby w WLU i powtórz poprzedni krok. Algorytm ten działa o ile przeglądamy liczby z zakresu 1 do N (N dowolna liczba całkowita). Jeżeli pierwszą liczbą w WLU nie jest 1 to należy stworzyć sztuczny iterator, który będzie przechodził przez dodatkową tablicę zawierającą wszystkie liczby pierwsze (TLP). Kolejną optymalizacją jaką można dokonać jest ograniczenie wielkości tablicy liczb pierwszych. Maksymalna wymagana liczba pierwsza to pierwiastek kwadratowy z ostatniej liczby wchodzącej w skład WLU. Tą własność można łatwo udowodnić, ponieważ największy dzielnik dowolnej liczby naturalnej nie może być większy niż pierwiastek kwadratowy z niej samej.

5.1.2 Przegląd Listy Liczb

Jest to jeden z najprostszych algorytmów wyszukiwania liczb pierwszych. W swojej sekwencyjnej wersji polega na pełnym przejrzaniu WLU i znalezieniu w nim liczb pierwszych. Jest to algorytm mniej skuteczny niż SE, jednak szybszy w implementacji.

5.1.3 Określanie pierwszości liczyb

Działa na zasadzie iterowania się po wszystkich liczbach z zakresu 2 do wartości liczby. Jednocześnie sprawdzamy, czy któraś z liczb nie jest dzielnikiem sprawdzanej liczby. Jeżeli tak to wiemy, że dana liczba jest liczbą złożoną. Po zakończeniu iterowania i nie znalezieniu dzielnika, możemy stwierdzić, że sprawdzana liczba jest pierwsza. Optymalizacja algorytmu polega na sprawdzaniu liczb tylko i wyłącznie nie większych niż pierwiastek kwadratowy z testowanej liczby. Uzasadnienie tego faktu można znaleźć w algorytmie SE.

5.2 Realizacja praktyczna

5.2.1 Wykorzystane klauzule OMP

`omp_get_max_threads()` podaje maksymalną dostępną w systemie liczbę procesorów logicznych

`omp_get_wtime()` zwraca czas pracy wszystkich wątków

`#pragma omp parallel` rozpoczyna obszar wykonania Równoległego

`#pragma omp for` pozwala na zrównoleglanie pętli dzieląc wykonywaną przez nią pracę pomiędzy dostępne wątki

`omp_get_thread_num()` zwraca numer aktualnie wykonywanego wątku

`omp_set_num_threads()` ustawia maksymalną ilość wątków jakiej może używać program

5.2.2 Sekwencyjne Określanie Pierwszości Liczb

Dokładne działanie algorytmu zostało zaprezentowane w części teoretycznej. Jedyną różnicą jest zastosowanie reszty z dzielenia jako metody określania dzielnika. Jeżeli wynikiem reszty z dzielenia jest zero to wiadomo, że *div* jest jednym z dzielników liczby *p*, co oznacza, że liczba *p* nie jest pierwsza.

```
1  bool isPrime(int p)
2  {
3      int div = 2;
4      double sqrtP = (int)sqrt(p) + 1;
5
6      for (; div < sqrtP; ++div){
7          if(p % div == 0)
8              return false;
9      }
10
11      return true;
12 }
```

Figure 1: Sekwencyjne Określanie Pierwszości Liczb

5.2.3 Sekwencyjny Pełny Przegląd Listy Liczb - SPPLL

Tak jak opisano w części teoretycznej polega na pełnym przeglądzie wszystkich liczb z podanego zakresu. Aby zoptymalizować wykonanie programu ustalamy wielkość wynikowego wektora na samym początku. Następnie w pętli

sprawdzamy wszystkie liczby w zakresie od *start* do *end*. Samo sprawdzanie odbywa się w funkcji *isPrime*, która została szczegółowo opisana wcześniej.

```

1  std::vector<int> sequential::withoutSieve(int start, int end, bool printOutput){
2      int size = end - start + 1;
3      std::vector<int> result;
4      double timeStart, timeStop;
5
6      result.reserve(size);
7      timeStart = omp_get_wtime();
8      for (size_t i = start; i <= end; ++i){
9          if (isPrime(i)) {
10             result.push_back(i);
11         }
12     }
13     timeStop = omp_get_wtime();
14     if(printOutput) period(timeStart, timeStop, "sequential::withoutSieve");
15
16     return result;
17 }

```

Figure 2: Sekwencyjny Przegląd Listy Liczb

5.2.4 Sekwencyjne Sito Erastotenesa - SSE

Dokładne działanie tego algorytmu zostało także zaprezentowane w części teoretycznej. Algorytm rozpoczyna od wyznaczenia wszystkich liczb pierwszych, aż do pierwiastka górnej granicy przeszukiwania (*wyjaśnienie w części teoretycznej*). Następnie w pierwszej pętli *for* usuwamy wszystkie wielokrotności kolejnych liczb pierwszych znajdujących się w tablicy *primes*. Jeżeli dana liczba została oznaczona jako *true* to mamy pewność, że jest liczbą złożoną. Po sprawdzeniu ostatniej liczby z wektora *primes*, możemy przepisać wszystkie liczby, które nie zostały zakwalifikowane jako złożone.

5.3 Równoległy Pełny Przegląd Listy Liczb

Zrównoleglenie Sekwencyjnego algorytmu Przeglądania Listy Liczb, opiera się na podziale WLU pomiędzy wszystkie wątki. W takim wypadku każdy wątek będzie odpowiedzialny za przeszukiwanie swojej i tylko swojej części WLU. Sercem algorytmu jest funkcja *isPrime* wywoływana dla każdej liczby w celu sprawdzenia, czy jest pierwsza. Jeżeli taka jest to należy ją zapisać do wektora wyjściowego.

Podejście Naiwne - RNPLL W wersji naiwnej algorytm zapisuje wyniki do jednowymiarowego wektora. Takie rozwiązanie nie jest poprawne. W skra-

```

1  std::vector<int> sequential::withSieve(int start, int end, bool printOutput){
2      int sqrtEnd = (int)sqrt(end);
3      double timeStart, timeStop;
4      timeStart = omp_get_wtime();
5      std::vector<int> primes = sequential::withoutSieve(2, sqrtEnd, false);
6      std::vector<bool> isPrime(end + 1, false);
7      int primesSize = primes.size();
8      for (auto& p: primes){
9          int multiple = p * 2;
10         while (multiple <= end) {
11             isPrime[multiple] = true;
12             multiple += p;
13         }
14     }
15     int size = end - start + 1;
16     std::vector<int> result;
17     result.reserve(size);
18     for (size_t i = start; i <= end; i++){
19         if (!isPrime[i]) result.push_back(i);
20     }
21     timeStop = omp_get_wtime();
22     if (printOutput) period(timeStart, timeStop, "sequential::withSieve");
23     return result;
24     return std::vector<int>();
25 }

```

Figure 3: Sekwencyjne Sito Erastotenesa

jnych przypadkach może wystąpić zjawisko wyścigu. Okaże się, że istnieją momenty, w których wiele wątków stara się zapisać do wektora wynikowego efekty swojego przetwarzania. Spowoduje to, że wynikowy wektor może zostać łatwo uszkodzony. Dodatkowo program jest wrażliwy na wystąpienie zjawiska false sharingu. Nie spowoduje on niepoprawnego działania kodu, jednak znacząco wpłynie na prędkość przetwarzania. Stanie się tak, ponieważ elementy wektora wyjściowego są ułożone w pamięci jeden obok drugiego, co łatwo prowadzi do uniważniania sąsiednich linii pamięci.

```

1  std::vector<int> primes(int start, int end, bool printOutput){
2      int size = end - start + 1;
3      std::vector<int> result;
4      double timeStart, timeStop;
5      result.reserve(size);
6      timeStart = omp_get_wtime();
7      #pragma omp parallel
8      {
9          #pragma omp for
10         for (int i = start; i <= end; i++){
11             if (isPrime(i)) result.push_back(i);
12         }
13     }
14     timeStop = omp_get_wtime();
15     if (printOutput) period(timeStart, timeStop, "domain::primes");
16     return result;
17 }
```

Podejście Optymalne - ROPPLL Poprawa wersji naiwnej jest prosta. Wystarczy dodać strukturę pośrednią. W taki sposób, że wątki równolegle będą do niej zapisywać, a później w sposób sekwencyjny spłaszczymy ją do jednowymiarowego wektora. Wspomnianą strukturą może być dwuwymiarowy wektor. W takim wypadku każdy z procesów będzie zapisywał do swojego własnego wiersza, a na końcu w sposób sekwencyjny połączymy wszystkie wiersze w jedną tabelę.

Zjawisko false sharingu nie zajdzie, ponieważ komórki pamięci będą znacząco od siebie oddzielone - conajmniej o jeden wiersz w wektorze. Zjawisko wyścigu także zostanie wyeliminowane. Żadne dwa wątki nie mogą zapisywać do tego samego wiersza, co oznacza, że mają swoje osobne przestrzenie robocze.

```

1  std::vector<int> primes(int start, int end, bool printOutput){
2      int size = end - start + 1;
3      std::vector<int> result;
4      double timeStart, timeStop;
5      int threadsCount = omp_get_max_threads();
6      result.reserve(size);
7      timeStart = omp_get_wtime();
8      std::vector<std::vector<int>> privateResults(threadsCount, std::vector<int>());
9      for (int i = 0; i < threadsCount; i++) privateResults[i].reserve(end - start + 1);
10     #pragma omp parallel
11     {
12         #pragma omp for
13         for (int i = start; i <= end; i++){
14             if (isPrime(i))
15                 privateResults[omp_get_thread_num()].push_back(i);
16         }
17     }
18     for (size_t i = 0; i < threadsCount; i++)
19         result.insert(
20             result.end(), privateResults[i].begin(), privateResults[i].end()
21         );
22     timeStop = omp_get_wtime();
23     if (printOutput) period(timeStart, timeStop, "funcional::primes");
24     return result;
25 }

```

5.3.1 Równoległe Sito Erastotenesa - wersja domenowa

Zrównoleglenie SE dla podejścia domenowego polega na przekazaniu całego WLU, ale tylko części wektora liczb pierwszych. W taki sposób każdy wątek będzie odpowiedzialny za sprawdzenie, czy elementy WLU są podzielne przez liczby pierwsze znane dla niego. Tak samo jak w sekwencyjnym SE i tutaj można zastosować optymalizację w postaci ograniczenia największej liczby pierwszej do sprawdzenia do pierwiastka kwadratowego największej z liczb w WLU.

Podejście Naiwne - RNSD W tej wersji naiwnej algorytmu rozpoczynamy od wygenerowania wszystkich potrzebnych liczb pierwszych. Następnie inicjalizujemy odpowiednio duży wektor przetrzymujący rozwiązanie. Następnie w pętli za pomocą odpowiedniej dyrektywy *OMP* przydzielamy liczby pierwsze do wątków. Następnie każdy z wątków sprawdza całe WLU w celu sprawdzenia, czy nie znajdują się w nim wielokrotności przydzielonych mu liczb pierwszych. Jeżeli takie występują to są oznaczane jako liczby pierwsze - wpis do wektora *isPrime* wartości *true* w odpowiednie miejsce. Na końcu powstały wektor *isPrime* przetwarzamy na wektor wynikowy.

Problemem wersji naiwnej jest niepoprawność generowanych wyników, oraz

występujące zjawisko false sharingu - oba spowodowane używaniem wspólnej listy *isPrime*. Jego geneza jest dokładnie taka sama jak w przypadku naiwej wersji algorytmu PPLL

Podejście Optymalne - RNSED W przypadku podejścia domenowego znaczącej zmianie ulega sposób przetrzymywania rozwiązań. Od teraz przechowujemy je w dwuwymiarowym wektorze. Na końcu spłaszczamy tę dwuwymiarową strukturę do jednowymiarowego wektora. Takie rozwiązanie pomoże nam zapobiec zjawisku false sharingu, oraz zjawisku wyścigu. Dzięki temu wyniki generowane przez program będą poprawne i wykonane w sposób efektywny.

```

1  std::vector<int> domain::basicSieve(int start, int end, bool printOutput){
2      double timeStart, timeStop;
3      int sqrtEnd = (int)sqrt(end);
4      std::vector<int> primes =
5          domain::primes(2, sqrtEnd, false);
6      std::vector<bool> isPrime(end + 1, false);
7      timeStart = omp_get_wtime();
8      #pragma omp parallel
9      {
10     #pragma omp for
11         for (int i = 0; i < primes.size(); i++){
12             int multiple = primes[i] * 2;
13             while (multiple <= end) {
14                 isPrime[multiple] = true;
15                 multiple += primes[i];
16             }
17         }
18     }
19     int size = end - start + 1;
20     std::vector<int> result;
21     result.reserve(size);
22     for (int i = start; i <= end; i++){
23         if (!isPrime[i]) {
24             result.push_back(i);
25         }
26     }
27     timeStop = omp_get_wtime();
28     if (printOutput) period(timeStart, timeStop, "domain::basicSieve");
29     return result;
30 }

```

Figure 4: Równoległe Naiwne Sito Erastotenesa w Wersji Domenowej

```

1  std::vector<int> domain::optimizedSive(int start, int end, bool printOutput){
2      double timeStart, timeStop;
3      int threadsCount = omp_get_max_threads();
4      int sqrtEnd = (int)sqrt(end);
5      std::vector<int> primes = domain::primes(2, sqrtEnd, false);
6      std::vector<std::vector<bool>> isPrime(threadsCount, std::vector<bool>(end + 1, false));
7      timeStart = omp_get_wtime();
8      #pragma omp parallel
9      {
10     #pragma omp for schedule(dynamic)
11     for (int i = 0; i < primes.size(); i++){
12         int multiple = primes[i] * 2;
13         while (multiple <= end) {
14             isPrime[omp_get_thread_num()][multiple] = true;
15             multiple += primes[i];
16         }
17     }
18     }
19     timeStop = omp_get_wtime();
20     int size = end - start + 1;
21     std::vector<int> result;
22     result.reserve(size);
23     for (int i = start; i <= end; i++){
24         int sum = false;
25         for (int j = 0; j < threadsCount; j++){
26             sum |= isPrime[j][i];
27         }
28         if (!sum) {
29             result.push_back(i);
30         }
31     }
32     }
33     period(timeStart, timeStop, "domain::optimizedSive");
34     return result;
35 }

```

Figure 5: Równoległe Optymalne Sito Erastotenesa w Wersji Domenowej

5.3.2 Równoległe Sito Erastotenesa - wersja funkcyjna

Zrównoleglenie Sita Erastotenesa w wersji funkcyjnej polega na podzieleniu WLU na części i rozdystrybuowanie ich pomiędzy poszczególne wątki. Natomiast każdy z wątków dostanie cały zbiór liczb pierwszych. Także i w podejściu funkcyjnym można zastosować optymalizacje ograniczające ilości liczb pierwszych do testowania. Dokładne uzasadnienie można znaleźć w części teoretycznej niniejszego

sprawozdania.

Podejście Naiwne - RNSEF Naiwna implementacja rozpoczyna od wygenerowania odpowiedniej ilości liczb pierwszych. Następnie jest deklarowana odpowiednio duża tablica do przechowywania wyników przetwarzania. Jest to wektor jedno wymiarowy, co będzie miało swoje poważne konsekwencje w przyszłości. Następnie w regionie równoległym każdy z wątków dostaje swoją część tablicy do przeszukania. Po wyjściu z regionu równoległego wynikowy wektor zostaje zamieniony na wektor liczb pierwszych.

W przypadku tej implementacji wyniki zawsze wychodzą poprawne. Dzieje się tak, ponieważ nie może tutaj wystąpić zjawisko wyścigu - zauważmy, że każdy z wątków działa na swojej części WLU i dlatego nigdy nie będą działały na tym samym obszarze wektora *isPrime*.

Z drugiej strony należy pamiętać, że wszystkie wątki działają na tej samej strukturze - *isPrime*, co doprowadza do zjawiska false sharingu i powtarzającego się unieważniania linii danych, a to doprowadza do strat w efektywności.

Podejście Optymalne - RNSED Uoptymalnienie

```

1  std::vector<int> funcional::basicSieve(int start, int end, bool printOutput){
2      double timeStart, timeStop;
3      int sqrtEnd = (int)sqrt(end);
4      std::vector<int> primes =
5          funcional::primes(2, sqrtEnd, false);
6      std::vector<bool> isPrime(end + 1, false);
7      int threadsCount = omp_get_max_threads();
8      timeStart = omp_get_wtime();
9      #pragma omp parallel
10     {
11         int privateA =
12             funcional::getStart(start, end, omp_get_thread_num(), threadsCount);
13         int privateB =
14             funcional::getEnd(start, end, omp_get_thread_num(), threadsCount);
15         for (int i = 0; i < primes.size(); i++){
16             int multiple = primes[i] * 2;
17             int expr = ((privateA - primes[i] * 2) / primes[i]) * primes[i];
18             if (expr > 0) multiple += expr;
19             while (multiple <= privateB) {
20                 isPrime[multiple] = true;
21                 multiple += primes[i];
22             }
23         }
24     }
25     int size = end - start + 1;
26     std::vector<int> result;
27     result.reserve(size);
28     for (int i = start; i <= end; i++) if (!isPrime[i]) result.push_back(i);
29     timeStop = omp_get_wtime();
30     if (printOutput) period(timeStart, timeStop, "funcional::basicSieve")
31     return result;
32 }

```

Figure 6: Równoległe Naiwne Sito Erastotenesa w Wersji Funkcyjnej

```

1  std::vector<int> funcional::optimizedSive(int start, int end, bool printOutput){
2      double timeStart, timeStop;
3      int threadsCount = omp_get_max_threads();
4      int sqrtEnd = (int)sqrt(end);
5      std::vector<int> primes =
6          funcional::primes(2, sqrtEnd, false);
7      std::vector<std::vector<bool>> isPrime(threadsCount, std::vector<bool>(end + 1, false));
8      timeStart = omp_get_wtime();
9      #pragma omp parallel
10     {
11         int privateA =
12             funcional::getStart(start, end, omp_get_thread_num(), threadsCount);
13         int privateB =
14             funcional::getEnd(start, end, omp_get_thread_num(), threadsCount);
15         for (int i = 0; i < primes.size(); i++)
16         {
17             int multiple = primes[i] * 2;
18             int expr = ((privateA - primes[i] * 2) / primes[i]) * primes[i];
19             if (expr > 0) {
20                 multiple += expr;
21             }
22             while (multiple <= privateB) {
23                 isPrime[omp_get_thread_num()][multiple] = true;
24                 multiple += primes[i];
25             }
26         }
27     }
28     timeStop = omp_get_wtime();
29     int size = end - start + 1;
30     std::vector<int> result;
31     result.reserve(size);
32     for (int i = start; i <= end; i++){
33         int sum = false;
34         for (int j = 0; j < threadsCount; j++){
35             sum |= isPrime[j][i];
36         }
37         if (!sum){
38             result.push_back(i);
39         }
40     }
41     period(timeStart, timeStop, "funcional::optimizedSive");
42     return result;
43 }

```

Figure 7: Równoległe Optymalne Sito Erastotenesa w Wersji Funkcyjnej

5.3.3 Sposób Podziału pracy w programach

Klauzula `schedule` określa sposób, w jaki iteracja `for` pętli jest dzielona między wątki w pętli `for`. Możliwe rodzaje podziału pracy:

static podział dokonany przed uruchomieniem pętli, najmniejszy narzut czasu wykonania

dynamic wątki wykonują kolejno „pierwszy wolny” segment w przestrzeni instrukcji `for`, jednak powoduje to opóźnienia z powodu konieczności

guided trochę jak w `dynamic`, wielkość segmentu może ulegać zmniejszaniu

runtime podział zależy od wartości zmiennej środowiskowej `OMP_SCHEDULE`

Praca dzielona jest zawsze na tyle części, ile jest wątków. W kodach programów większości mamy x iteracji, które dzielimy na n wątków więc każdy dostaje po x/n zadań; wyjątek stanowi kod `RNSED`, gdzie wprowadziliśmy konstrukcję `schedule`.

6 Prezentacja i Omówienie eksperymentu obliczeniowo-pomiarowego

6.1 Metodyka wykonania eksperymentów

Wszystkie prezentowane eksperymenty zostały uruchomione na maszynie opisanej na początku tego sprawozdania. Do testów zostały użyte wszystkie algorytmy opisane w powyższej części sprawozdania.

Rozmiary instancji testowych to: ($MAX = 7 * 10^7$)

- $2..MAX$
- $MAX/2..MAX$
- $2..MAX/2$

Ilości używanych wątków to:

jeden wątek podejście sekwencyjne

połowa dostępnych procesorów logicznych każdy z rdzeni procesora powinien dostać swój wątek, w tym wypadku ilość wątków będzie równała się ilości procesorów fizycznych

dostępne rdzenie logiczne każdy z procesorów logicznych powinien dostać dokładnie po 2 wątki, jest to możliwe dzięki wykorzystaniu technologii `HyperThreading` implementowanej dla procesorów Intel

6.2 Charakterystyka Programu Testującego

Intel®VTune jest oprogramowaniem służącym do analizy programów sekwencyjnych oraz równoległych. Pozwala on na określenie:

- miejsc oraz funkcji, w których program spędził najwięcej czasu
- sekcji kodu, które nieefektywnie wykorzystują zasoby procesora
- miejsc w kodzie, które należy zoptymalizować dla lepszego działania sekwencyjnego oraz wielowątkowego
- punktów synchronizacji, które wprowadzają największe opóźnienia
- gdzie, kiedy i dlaczego program spędza duże ilości czasu na operacjach I/O
- czy program jest ograniczany przez CPU czy GPU
- wpływ różnych metod synchronizacji na efektywność programu
- problemów sprzętowych, np. false sharing

W naszym projekcie nie skorzystaliśmy ze wszystkich dostępnych analiz i metryk oferowanych przez VTune, lecz skupiliśmy się na informacjach oferowanych przez Microarchitecture Exploration.

6.3 Wyniki Uzyskane Dla Algorytmów Sekwencyjnych

Pierwszą grupą testowanych algorytmów, będą algorytmy sekwencyjne. Przeprowadzimy dla nich eksperymenty na wszystkich dostępnych zestawach testowych, które opisaliśmy powyżej. Ma to na celu identyfikację wszystkich cech stworzonych przez nas algorytmów sekwencyjnych. Dane zebrane w tym punkcie pomogą nam w określeniu zmian wydajnościowych w dalszych analizach dla rozwiązań równoległych.

Z tabelki zaprezentowanej poniżej, możemy odczytać, że w porównaniu metod SPPLL i SSE zdecydowanym faworytem jest metoda Sita Erastotenesa.

6.4 Wyniki Uzyskane Dla Algorytmów Równoległych

Drugą grupą testowanych algorytmów są algorytmy równoległe. Do analizy uzyskanych w tym punkcie wyników posłużymy się dwoma parametrami:

Acceleration określa przyspieszenie algorytmu równoległego względem najszybszego odpowiadającego mu i przetestowanego algorytmu sekwencyjnego
- większa wartość = lepiej

Efektywność stosunek *Acceleration* względem ilości procesorów logicznych -
większa wartość = lepiej

Figure 8: Równoległe Naiwne Sito Erastotenesa wersja Domenowa - RNSED

method	Parallel domain naïve with sieve					
number of threads	4			8		
Range	2...MAX	2..MAX/2	MAX/2...MAX	2...MAX	2..MAX/2	MAX/2...MAX
acceleration	9,26E-01	7,25E-01	7,50E-01	8,77E-01	9,29E-01	7,51E-01
Speed	1,12E+08	1,07E+08	5,79E+07	1,06E+08	1,37E+08	5,80E+07
efficiency	2,32E-01	1,81E-01	1,88E-01	2,19E-01	2,32E-01	1,88E-01

Figure 9: Równoległe Optymalne Sito Erastotenesa wersja Domenowa - RNSED

method	Parallel domain optimal with sieve					
number of threads	4			8		
Range	2...MAX	2...MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
acceleration	5,29E-01	4,53E-01	4,70E-01	4,30E-01	3,76E-01	4,30E-01
Speed	6,39E+07	6,69E+07	3,63E+07	5,19E+07	5,56E+07	3,32E+07
efficiency	1,32E-01	1,13E-01	1,18E-01	1,07E-01	9,40E-02	1,08E-01

Figure 10: Równoległe Naiwne Sito Erastotenesa wersja Funkcyjna - RNSEF

method	Parallel functional naïve with sieve					
number of threads	4			8		
Range	2...MAX	2..MAX/2	MAX/2...MAX	2...MAX	2..MAX/2	MAX/2...MAX
acceleration	1,16E+00	1,39E+00	2,94E+00	1,87E+00	1,31E+00	2,65E+00
Speed	1,40E+08	2,05E+08	2,27E+08	2,26E+08	1,93E+08	2,05E+08
efficiency	2,90E-01	3,46E-01	7,35E-01	4,67E-01	3,27E-01	6,62E-01

Figure 11: Równoległe Optymalne Sito Erastotenesa wersja Funkcyjna - RNSED

method	Parallel functional optimal with sieve					
number of threads	4			8		
Range	2...MAX	2..MAX/2	MAX/2...MAX	2...MAX	2..MAX/2	MAX/2...MAX
acceleration	6,20E-01	5,90E-01	1,00E+00	6,59E-01	4,83E-01	8,78E-01
Speed	7,49E+07	8,71E+07	7,76E+07	7,96E+07	7,13E+07	6,78E+07
efficiency	1,55E-01	1,47E-01	2,51E-01	1,65E-01	1,21E-01	2,19E-01

Figure 12: Równoległe Naiwne Pełne Przeglądanie Listy Liczb - RNPPLL

method	Parallel naïve without sieve					
number of threads	4			8		
Range	2...MAX	2..MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
acceleration	2,48E-02	2,89E-02	3,55E-02	3,49E-02	3,89E-02	4,95E-02
Speed	3,00E+06	4,27E+06	2,74E+06	4,22E+06	5,75E+06	3,82E+06
efficiency	6,21E-03	7,22E-03	8,87E-03	8,73E-03	9,73E-03	1,24E-02

Figure 13: Równoległe Optymalne Pełne Przeglądanie Listy Liczb - ROPPLL

method	Parallel optimal without sieve					
number of threads	4			8		
Range	2...MAX	2..MAX/2	MAX/2...MAX	2...MAX	2...MAX/2	MAX/2...MAX
acceleration	2,56E-02	2,88E-02	3,61E-02	3,66E-02	3,96E-02	4,88E-02
Speed	3,10E+06	4,25E+06	2,79E+06	4,42E+06	5,84E+06	3,77E+06
efficiency	6,41E-03	7,19E-03	9,04E-03	9,14E-03	9,89E-03	1,22E-02

Z powyższych tabel można wyczytać, że najlepsze wyniki otrzymaliśmy dla programów RNSD, czyli Równoległe Optymalne Sieto Erastotenesa w wersji Domenowej, dla dowolnej instancji testowej oraz ilość procesorów logicznych.

6.4.1 Tablica Zbiorcza wszystkich uzyskanych wyników

method	number of threads	range	Elapsed time [s]	Instructions retired	Clockticks	Retiring	Front-end bound	Back-end bound	Memory bound	Core bound	effective physical core utilization	Speed [ilość/s]
Sequential without sieve	1	2...MAX	58,092	2,25E+11	2,29E+11	50.3%	50.9%	0.0%	0.0%	0.0%	24.0%	1,20E+06
		2...MAX/2	21,86	8,38E+10	8,62E+10	49.8%	49.4%	0.4%	0.0%	0.4%	23.5%	1,60E+06
		MAX/2...MAX	36,633	1,41E+11	1,45E+11	48.1%	48.4%	3.2%	0.0%	3.2%	23.7%	9,55E+05
Sequential with sieve	1	2...MAX	0,579	2,31E+09	1,98E+09	33.2%	13.0%	44.1%	38.3%	5.8%	21.1%	1,21E+08
		2...MAX/2	0,237	1,17E+09	8,59E+08	31.0%	13.0%	41.3%	0.0%	41.3%	24.2%	1,48E+08
		MAX/2...MAX	0,453	1,95E+09	1,58E+09	26.5%	1.8%	66.4%	66.4%	0.0%	24.6%	7,73E+07
Parallel domain naive with sieve	4	2...MAX	0,625	4,54E+09	6,86E+09	21.6%	5.4%	71.4%	17.9%	53.5%	26.5%	1,12E+08
		2...MAX/2	0,327	2,27E+09	2,94E+09	46.3%	7.6%	46.0%	12.5%	33.6%	20.5%	1,07E+08
		MAX/2...MAX	0,604	3,79E+09	6,52E+09	37.1%	11.7%	50.4%	26.2%	24.2%	24.9%	5,79E+07
	8	2...MAX	0,66	6,04E+09	1,19E+10	26.4%	10.1%	62.9%	41.3%	21.5%	36.3%	1,06E+08
		2...MAX/2	0,255	3,23E+09	5,60E+09	19.5%	6.0%	74.0%	43.0%	31.0%	28.7%	1,37E+08
		MAX/2...MAX	0,603	4,71E+09	9,24E+09	33.3%	8.5%	58.2%	44.8%	13.4%	32.4%	5,80E+07
Parallel domain optimal with sieve	4	2...MAX	1,095	1,26E+10	1,29E+10	35.1%	6.6%	57.1%	43.8%	13.3%	46.9%	6,39E+07
		2...MAX/2	0,523	6,66E+09	7,21E+09	30.5%	6.4%	63.4%	53.6%	9.8%	43.7%	6,69E+07
		MAX/2...MAX	0,963	1,04E+10	1,25E+10	36.3%	8.2%	53.9%	41.5%	12.3%	53.2%	3,63E+07
	8	2...MAX	1,348	1,71E+10	1,97E+10	50.3%	8.7%	38.8%	27.8%	11.0%	45.5%	5,19E+07
		2...MAX/2	0,63	9,52E+09	1,23E+10	32.8%	8.0%	58.7%	44.9%	13.9%	44.5%	5,56E+07
		MAX/2...MAX	1,053	1,29E+10	1,80E+10	35.2%	8.6%	56.8%	47.0%	9.8%	54.5%	3,32E+07
Parallel functional naive with sieve	4	2...MAX	0,5	3,46E+09	5,85E+09	15.4%	5.0%	78.4%	71.2%	7.2%	38.7%	1,40E+08
		2...MAX/2	0,171	1,76E+09	2,00E+09	60.2%	33.6%	4.8%	3.4%	1.4%	23.9%	2,05E+08
		MAX/2...MAX	0,154	1,75E+09	1,87E+09	58.3%	6.0%	34.2%	16.4%	17.8%	25.2%	2,27E+08
	8	2...MAX	0,31	4,00E+09	5,29E+09	46.1%	11.1%	40.2%	17.9%	22.3%	33.8%	2,26E+08
		2...MAX/2	0,181	1,89E+09	2,41E+09	75.4%	25.5%	0.0%	0.0%	0.0%	29.7%	1,93E+08
		MAX/2...MAX	0,171	1,91E+09	2,37E+09	29.5%	22.4%	45.7%	26.6%	19.1%	33.4%	2,05E+08
Parallel functional optimal with sieve	4	2...MAX	0,934	1,24E+10	1,01E+10	30.7%	9.4%	58.8%	32.2%	26.6%	40.2%	7,49E+07
		2...MAX/2	0,402	6,61E+09	5,42E+09	55.9%	18.9%	24.4%	8.2%	16.2%	32.3%	8,71E+07
		MAX/2...MAX	0,451	6,79E+09	5,91E+09	42.6%	11.0%	45.6%	14.0%	31.6%	29.6%	7,76E+07
	8	2...MAX	0,879	1,68E+10	1,27E+10	51.7%	11.4%	35.4%	9.8%	25.6%	38.4%	7,96E+07
		2...MAX/2	0,491	9,43E+09	9,20E+09	51.1%	8.5%	8.5%	18.1%	20.8%	34.1%	7,13E+07
		MAX/2...MAX	0,516	9,74E+09	9,74E+09	49.1%	11.5%	37.9%	8.9%	29.1%	32.5%	6,78E+07
Parallel naive without sieve	4	2...MAX	23,31	2,26E+11	2,63E+11	51.5%	51.6%	0.0%	0.0%	0.0%	59.0%	3,00E+06
		2...MAX/2	8,202	8,55E+10	9,92E+10	52.5%	50.8%	0.0%	0.0%	0.0%	58.5%	4,27E+06
		MAX/2...MAX	12,764	1,43E+11	1,69E+11	50.6%	48.7%	0.4%	0.0%	0.4%	70.4%	2,74E+06
	8	2...MAX	16,576	2,28E+11	3,24E+11	58.1%	41.9%	0.0%	0.0%	0.0%	77.6%	4,22E+06
		2...MAX/2	6,091	8,69E+10	1,26E+11	59.6%	42.0%	0.0%	0.0%	0.0%	78.2%	5,75E+06
		MAX/2...MAX	9,156	1,44E+11	2,14E+11	60.5%	38.8%	0.5%	0.0%	0.5%	84.9%	3,82E+06
Parallel optimal without sieve	4	2...MAX	22,589	2,27E+11	2,58E+11	49.4%	49.6%	0.8%	0.0%	0.8%	61.3%	3,10E+06
		2...MAX/2	8,242	8,55E+10	1,01E+11	48.4%	48.8%	2.3%	0.0%	2.3%	59.2%	4,25E+06
		MAX/2...MAX	12,532	1,43E+11	1,67E+11	50.0%	49.9%	0.0%	0.0%	0.0%	70.9%	2,79E+06
	8	2...MAX	15,832	2,29E+11	3,24E+11	58.8%	42.7%	0.0%	4.1%	0.0%	81.3%	4,42E+06
		2...MAX/2	5,992	8,68E+10	1,25E+11	55.7%	39.8%	4.1%	0.0%	4.1%	80.2%	5,84E+06
		MAX/2...MAX	9,291	1,44E+11	2,14E+11	61.8%	40.1%	0.0%	0.0%	0.0%	86.7%	3,77E+06

Figure 14: Zagregowana Tablica Uzyskanych Pomiarów

6.5 Analiza Udziału Poszczególnych Procesorów

W poniższym zestawieniu porównane zostaną procentowe udziały wykorzystania każdego z dostępnych procesorów logicznych platformy testującej.

6.5.1 Sekwencyjny Pełny Przegląd Listy Liczb - SPPLL

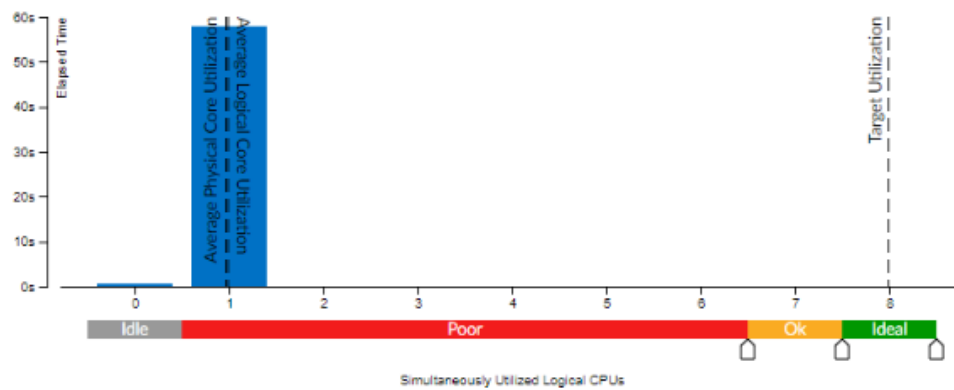


Figure 15: SPPLL dla liczb z zakresu 2..*MAX*

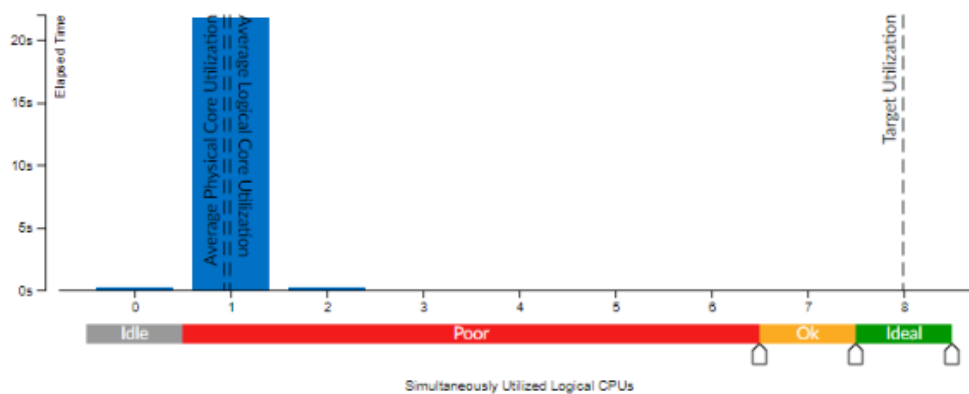


Figure 16: SPPLL dla liczb z zakresu 2..*MAX*/2

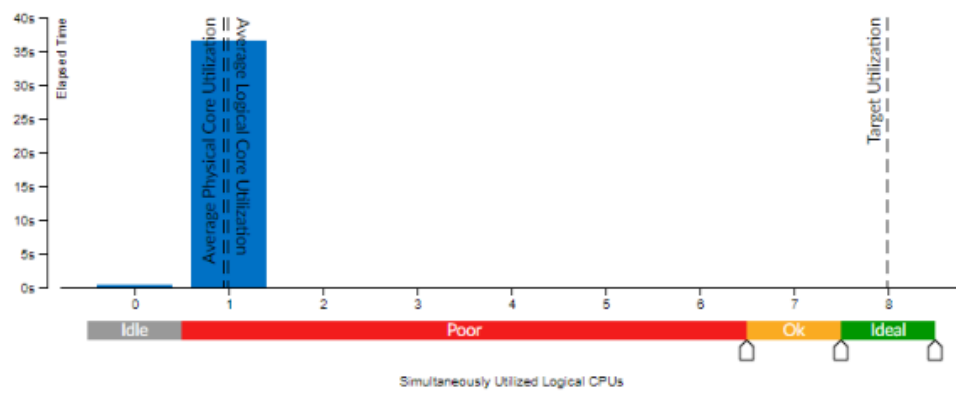


Figure 17: SPPLL dla liczb z zakresu $MAX/2..MAX$

6.5.2 Sekwencyjne Sito Erastotenesa - SSE

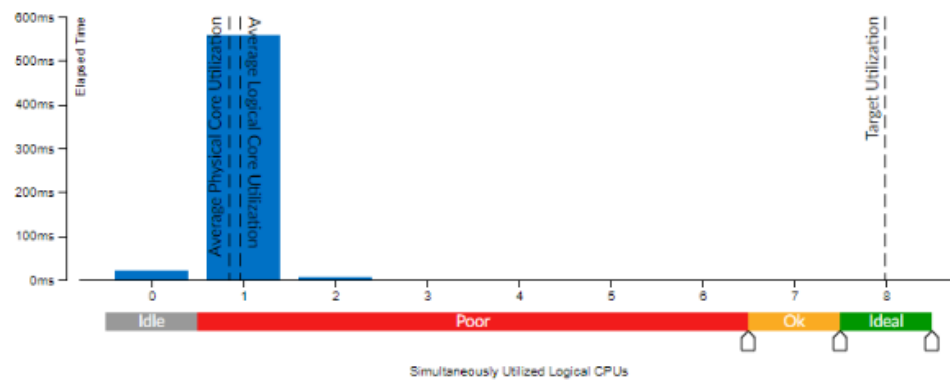


Figure 18: SSE dla liczb z zakresu $2..MAX$

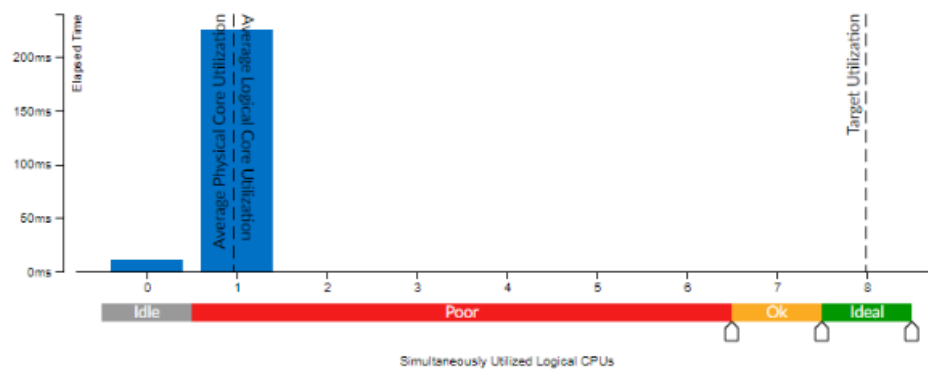


Figure 19: SSE dla liczb z zakresu $2..MAX/2$

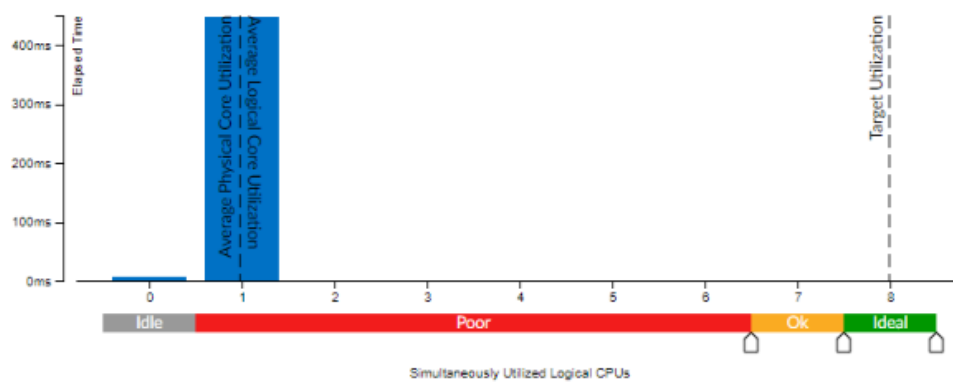


Figure 20: SSE dla liczb z zakresu $MAX/2..MAX$

6.5.3 Równoległe Naiwne Sito Erastotenesa wersja Funkcyjna

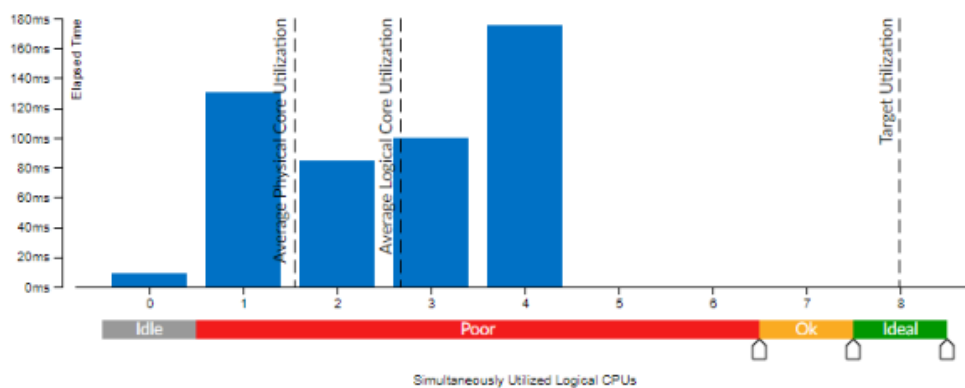


Figure 21: RNSEF dla liczb z zakresu $2..MAX$ dla 4 procesorów logicznych

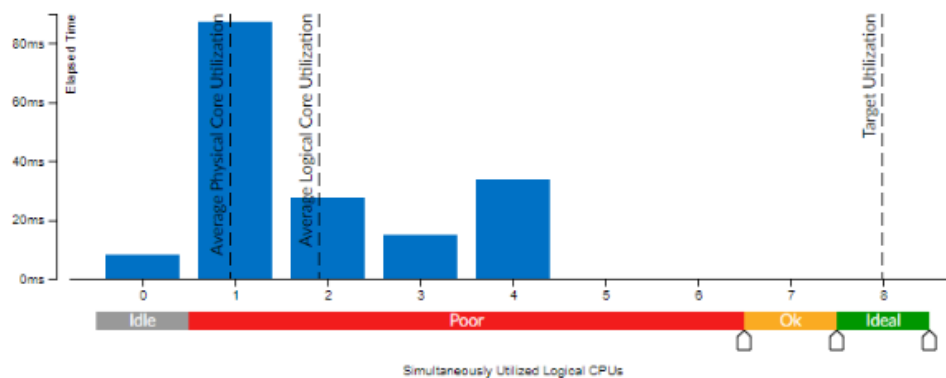


Figure 22: RNSEF dla liczb z zakresu $2..MAX/2$ dla 4 procesorów logicznych

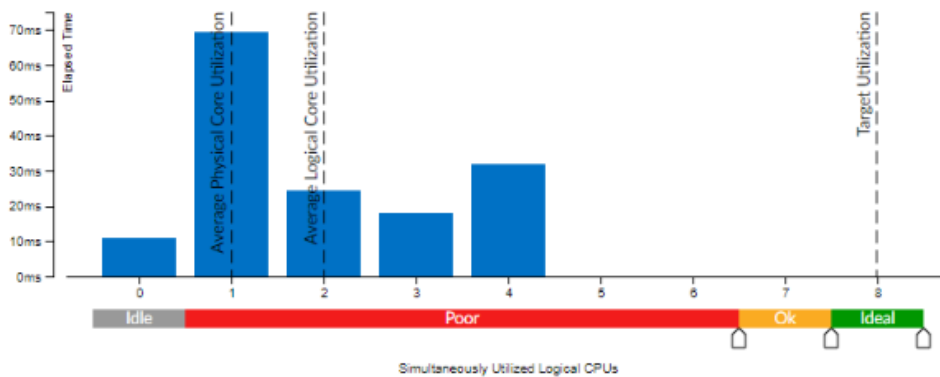


Figure 23: RNSEF dla liczb z zakresu $MAX/2..MAX$ dla 4 procesorów logicznych

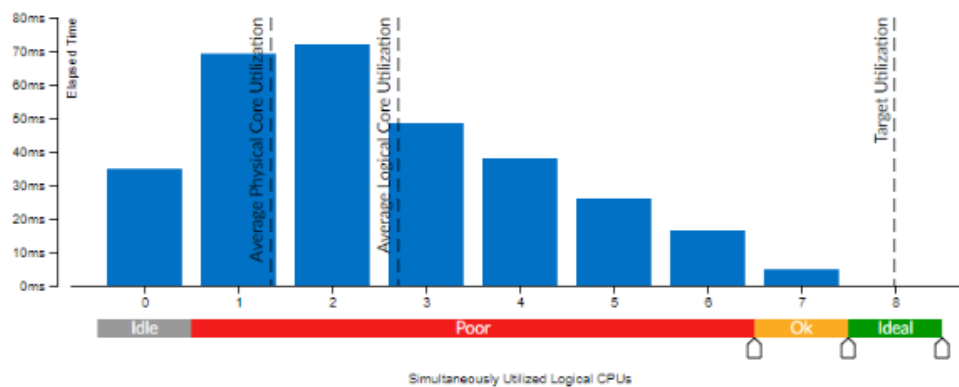


Figure 24: RNSEF dla liczb z zakresu $2..MAX$ dla 8 procesorów logicznych

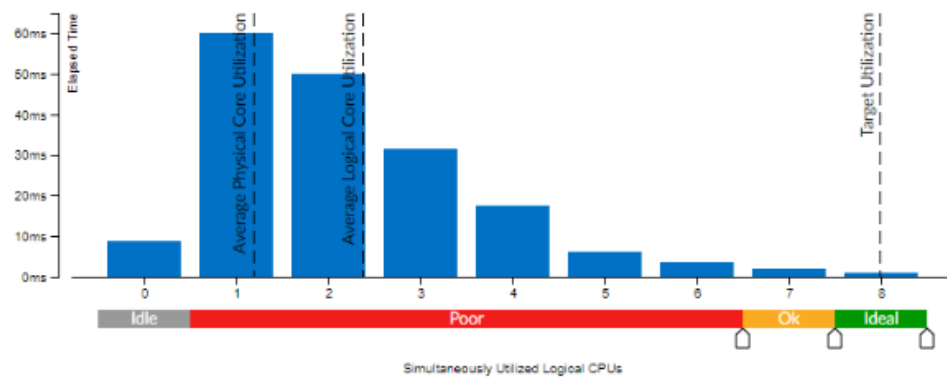


Figure 25: RNSEF dla liczb z zakresu $2..MAX/2$ dla 8 procesorów logicznych

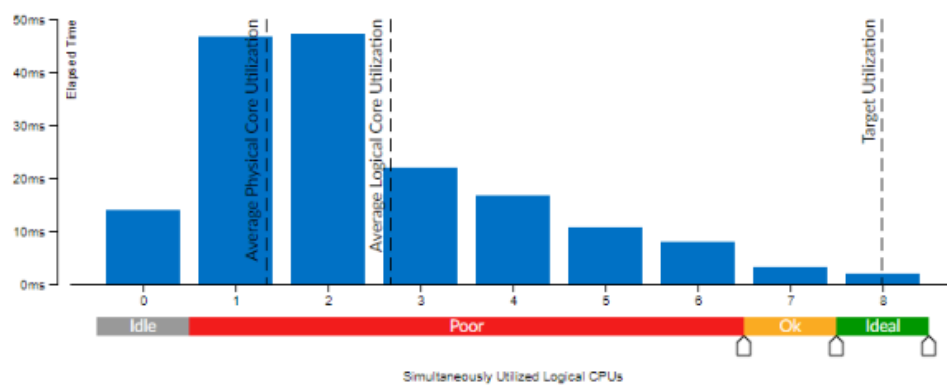


Figure 26: RNSEF dla liczb z zakresu $MAX/2..MAX$ dla 8 procesorów logicznych

6.5.4 Równoległe Optymalne Sito Erastotenesa wersja Funkcyjna

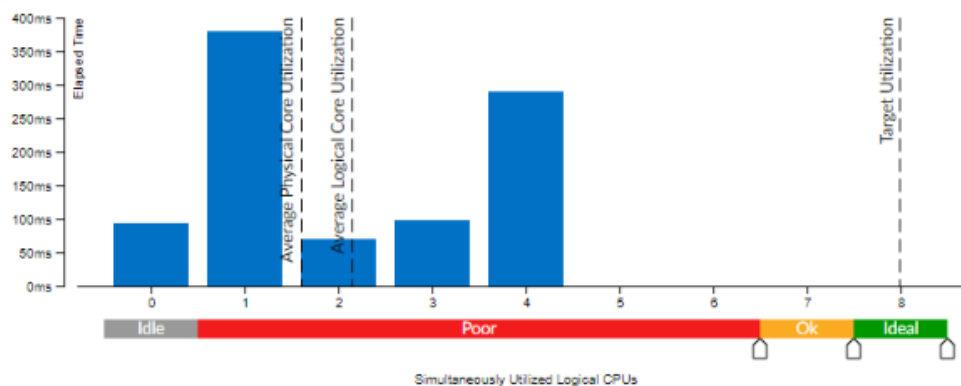


Figure 27: RNSD dla liczb z zakresu 2.. MAX dla 4 procesorów logicznych

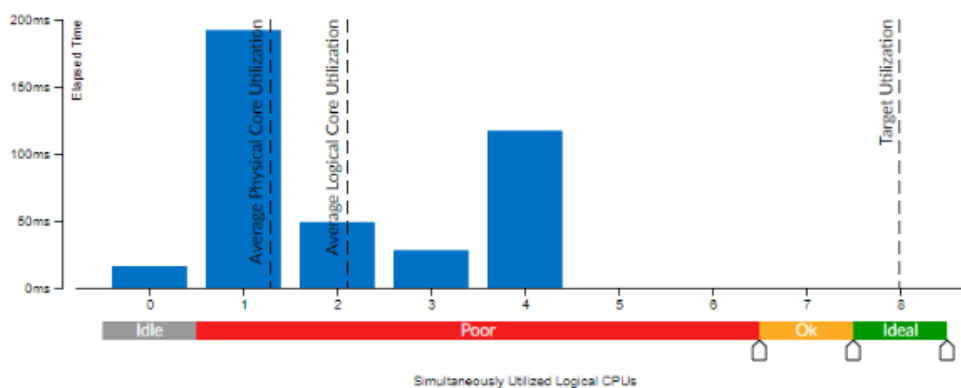


Figure 28: RNSD dla liczb z zakresu 2.. $MAX/2$ dla 4 procesorów logicznych

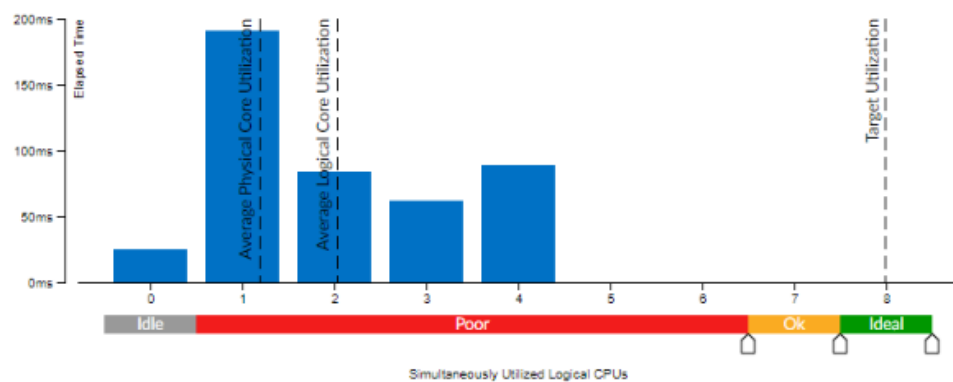


Figure 29: RNSD dla liczb z zakresu $MAX/2..MAX$ dla 4 procesorów logicznych

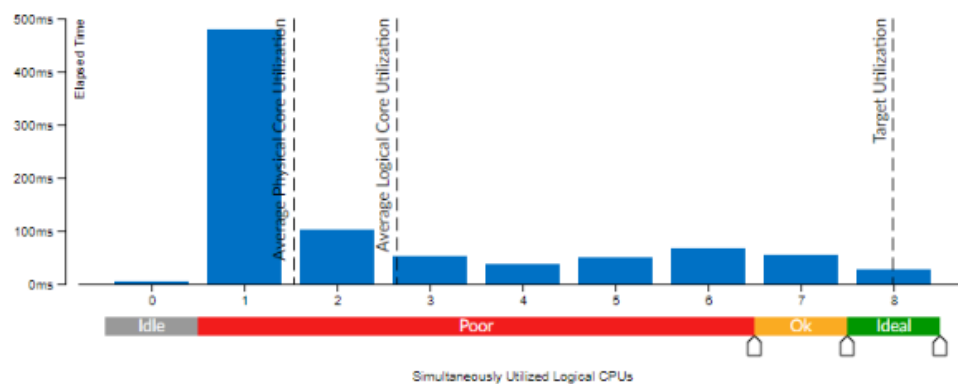


Figure 30: RNSD dla liczb z zakresu $2..MAX$ dla 8 procesorów logicznych

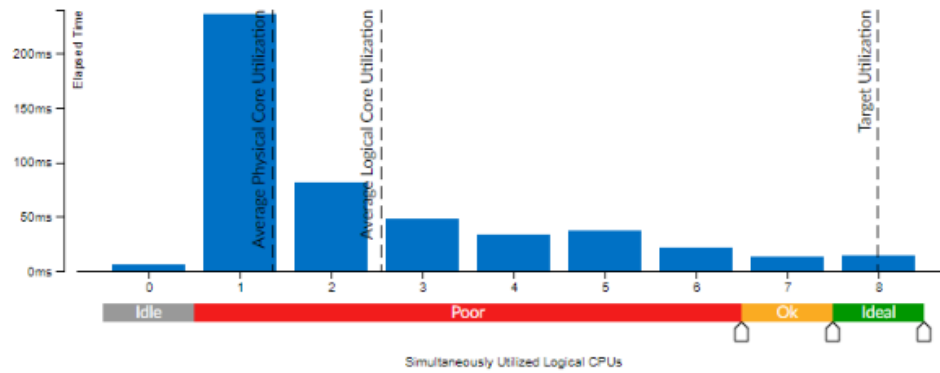


Figure 31: RNSD dla liczb z zakresu $2..MAX/2$ dla 8 procesorów logicznych

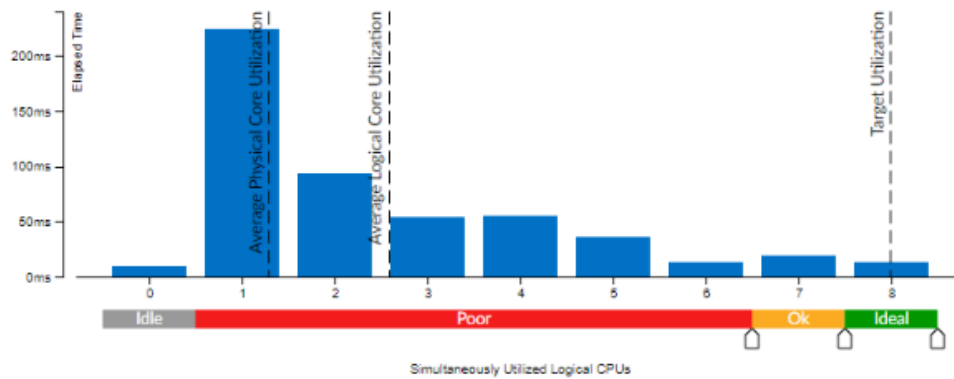


Figure 32: RNSD dla liczb z zakresu $MAX/2..MAX$ dla 8 procesorów logicznych

6.5.5 Równoległe Optymalne Sito Erastotenesa wersja Domenowa

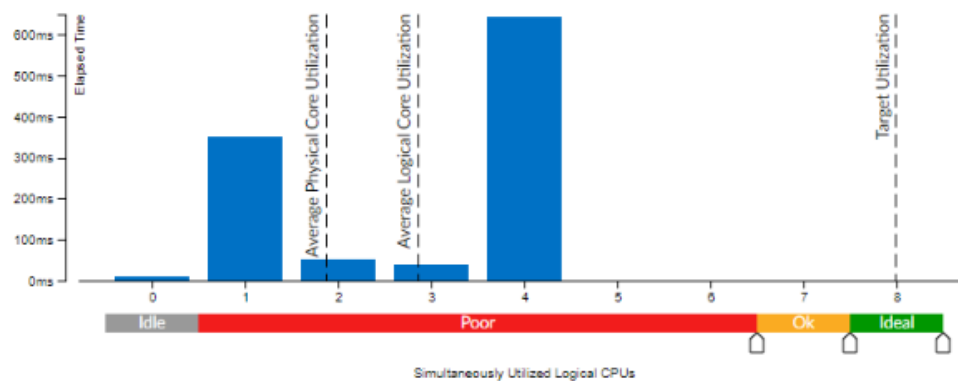


Figure 33: RNSD dla liczb z zakresu 2.. MAX dla 4 procesorów logicznych

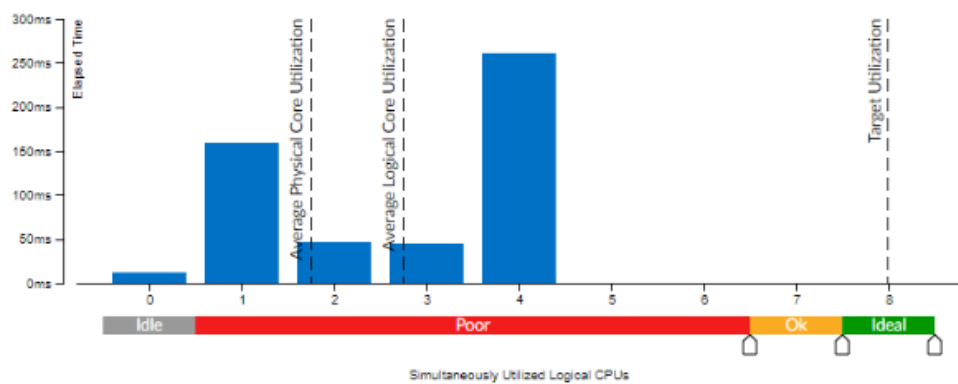


Figure 34: RNSD dla liczb z zakresu 2.. $MAX/2$ dla 4 procesorów logicznych

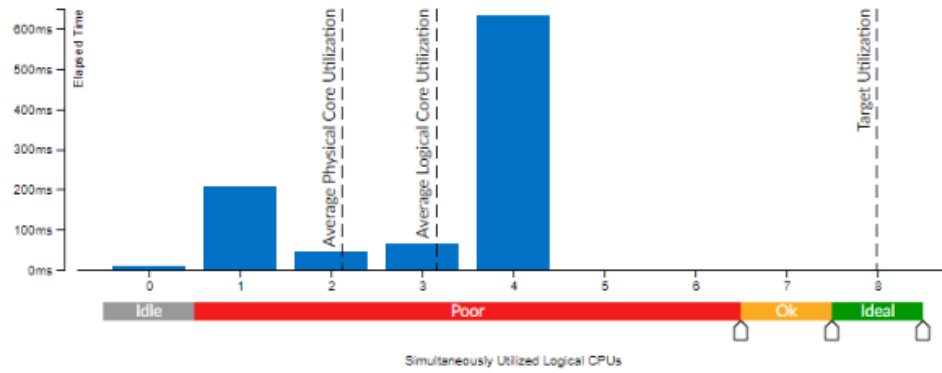


Figure 35: RNSD dla liczb z zakresu $MAX/2..MAX$ dla 4 procesorów logicznych

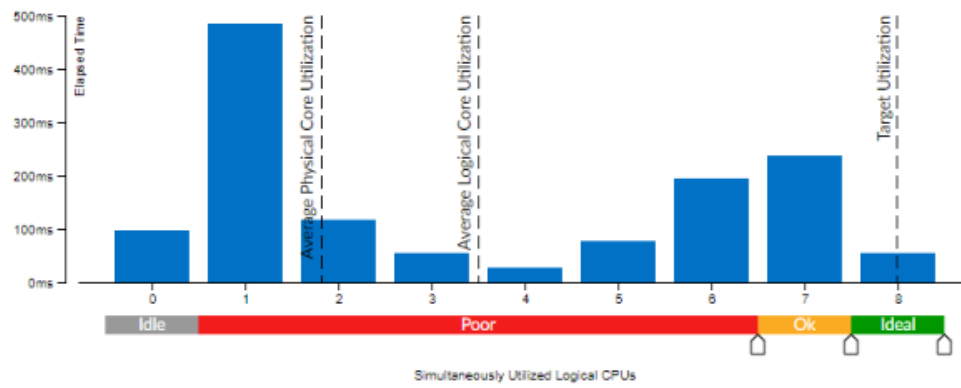


Figure 36: RNSD dla liczb z zakresu $2..MAX$ dla 8 procesorów logicznych

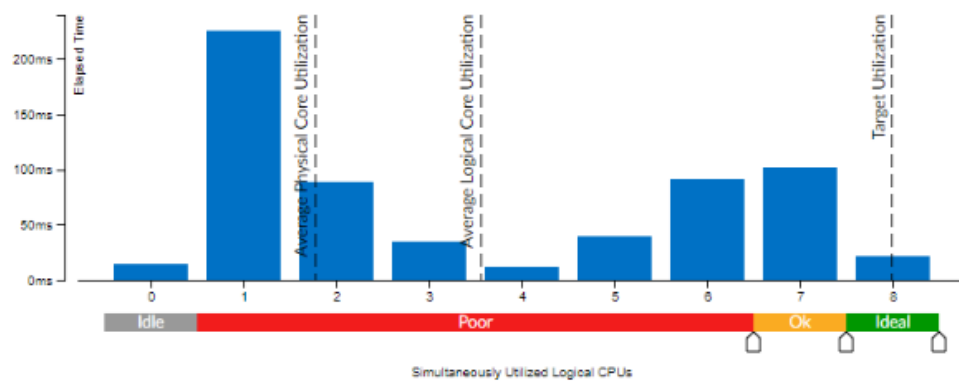


Figure 37: RNSED dla liczb z zakresu $2..MAX/2$ dla 8 procesorów logicznych

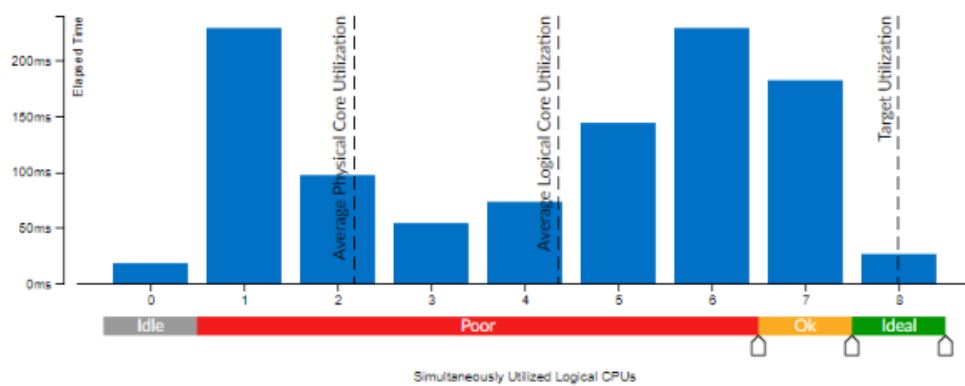


Figure 38: RNSED dla liczb z zakresu $MAX/2..MAX$ dla 8 procesorów logicznych

6.5.6 Równoległe Naiwne Pełne Przeglądanie Listy Liczb

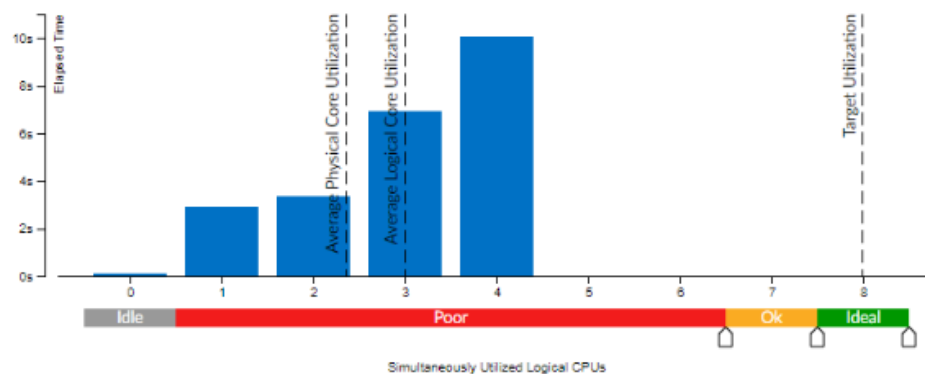


Figure 39: RNPPLL dla liczb z zakresu $2..MAX$ dla 4 procesorów logicznych

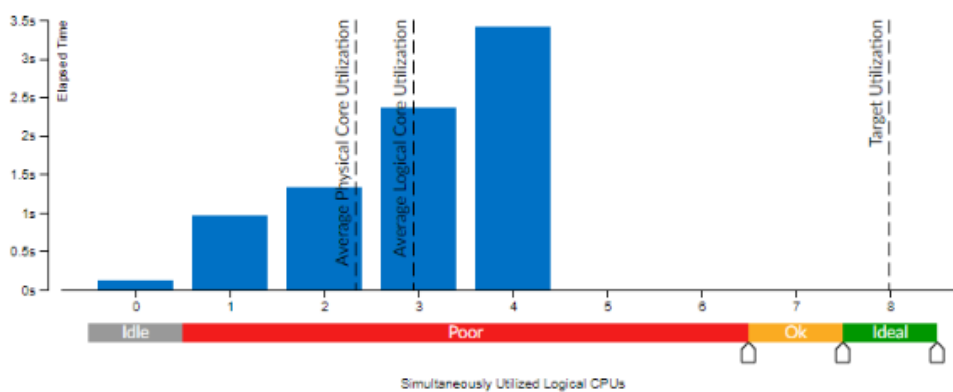


Figure 40: RNPPLL dla liczb z zakresu $2..MAX/2$ dla 4 procesorów logicznych

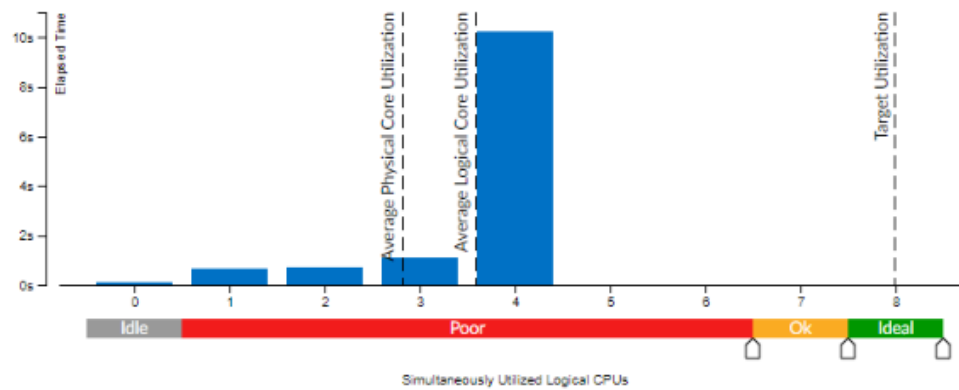


Figure 41: RNPPLL dla liczb z zakresu $MAX/2..MAX$ dla 4 procesorów logicznych

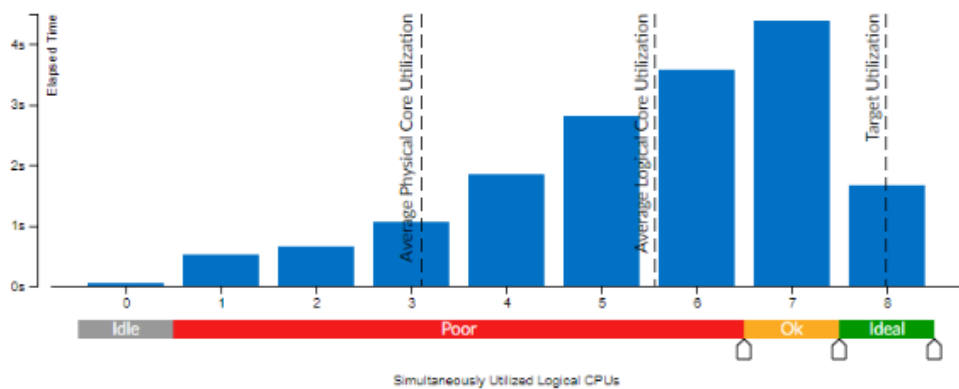


Figure 42: RNPPLL dla liczb z zakresu $2..MAX$ dla 8 procesorów logicznych

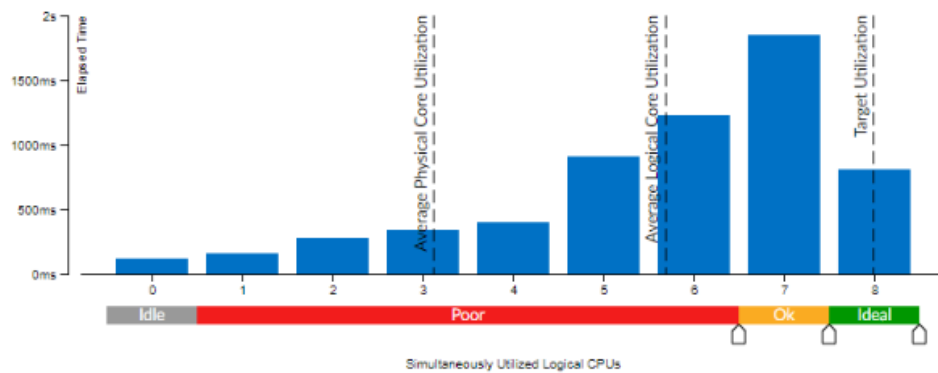


Figure 43: RNPPLL dla liczb z zakresu $2..MAX/2$ dla 8 procesorów logicznych

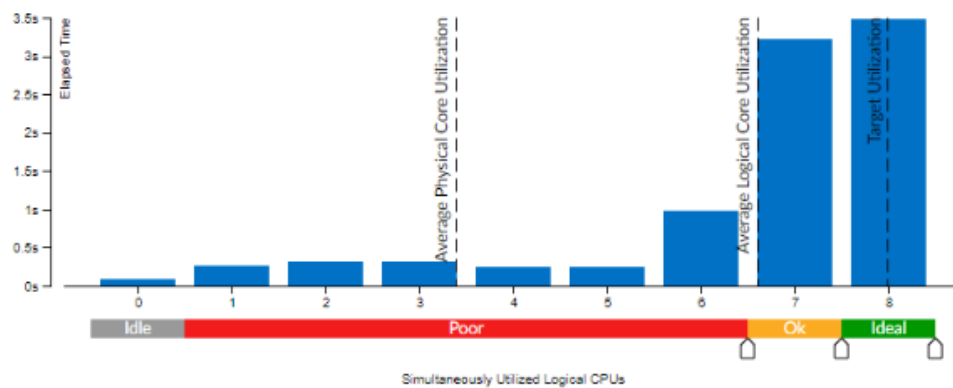


Figure 44: RNPPLL dla liczb z zakresu $MAX/2..MAX$ dla 8 procesorów logicznych

6.5.7 Równoległe Optymalne Pełne Przeglądanie Listy Liczb

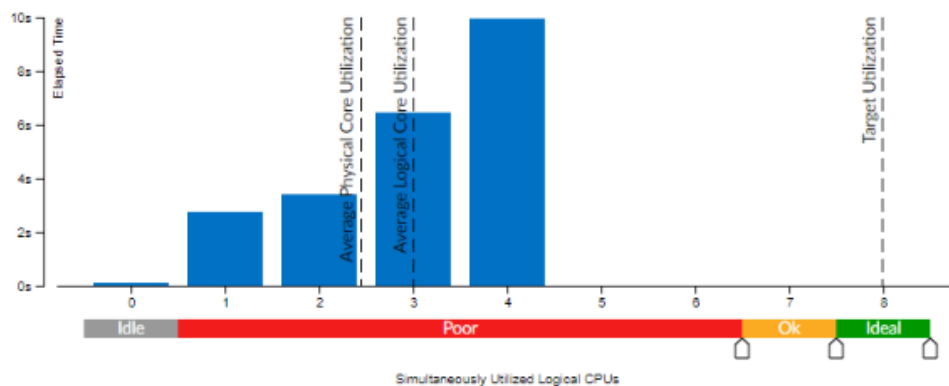


Figure 45: ROPPLL dla liczb z zakresu 2.. MAX dla 4 procesorów logicznych

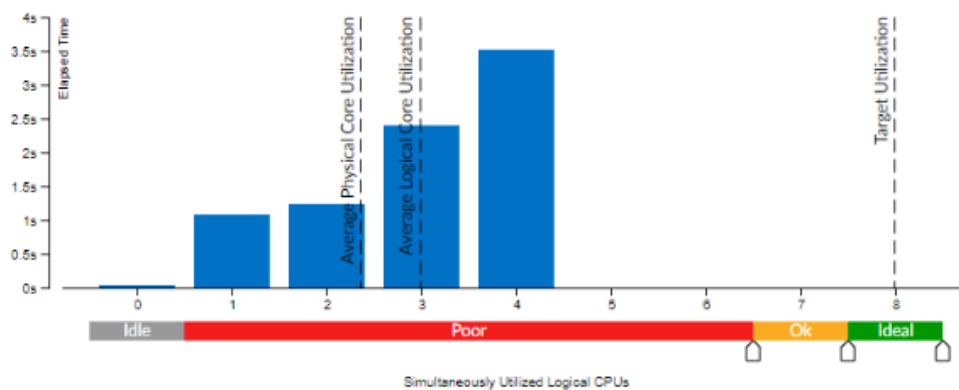


Figure 46: ROPPLL dla liczb z zakresu 2.. $MAX/2$ dla 4 procesorów logicznych

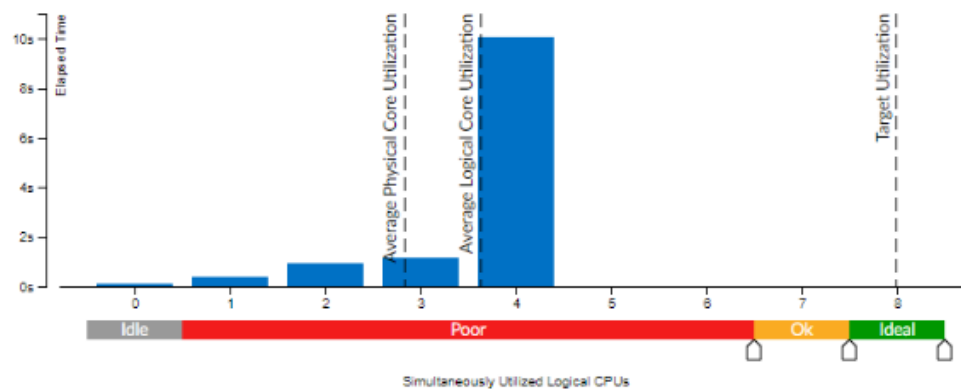


Figure 47: ROPPLL dla liczb z zakresu $MAX/2..MAX$ dla 4 procesorów logicznych

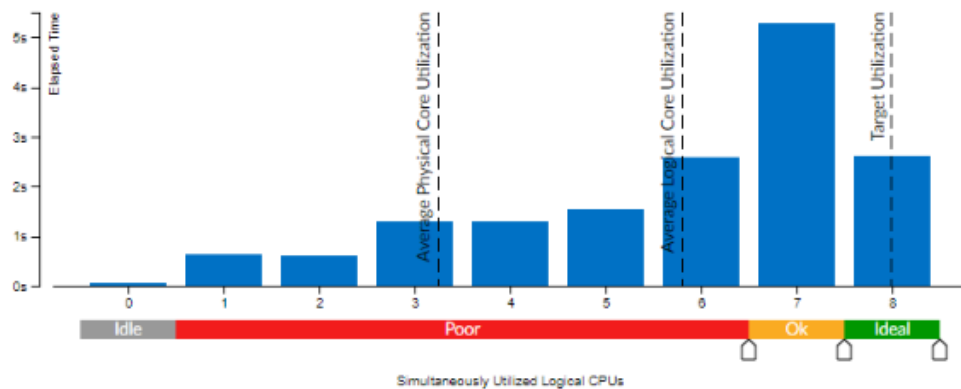


Figure 48: ROPPLL dla liczb z zakresu $2..MAX$ dla 8 procesorów logicznych

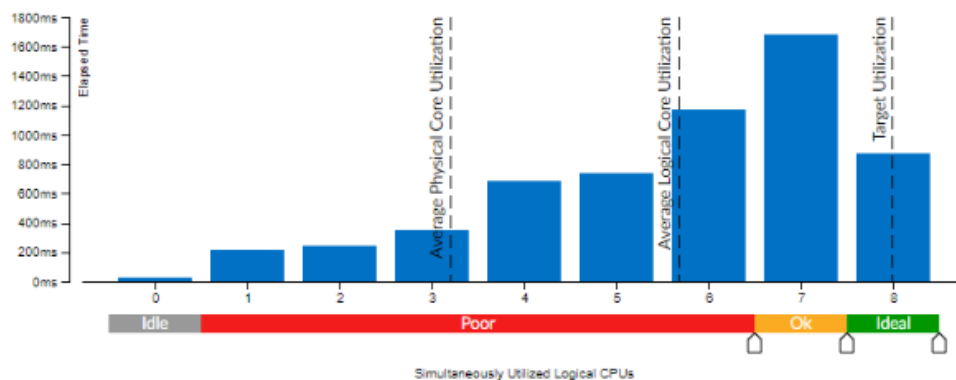


Figure 49: ROPPLL dla liczb z zakresu $2..MAX/2$ dla 8 procesorów logicznych

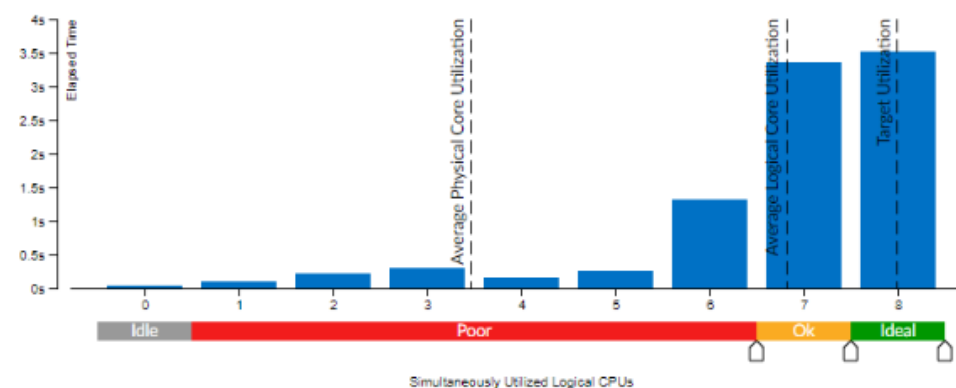


Figure 50: ROPPLL dla liczb z zakresu $MAX/2..MAX$ dla 8 procesorów logicznych

7 Wnioski

1. Niestety po przejrzeniu powyższych wyników, możemy dojść do konkluzji, że nie wszystkie programy w równym stopniu wykorzystują wewnętrzne struktury procesora. Najlepszy z analizowanych algorytmów (taki, który uzyskał najwyższą efektywność) czyli RNSD wykorzystywał procesor efektywnie w większości badanych instancji. Używał albo wszystkich wątków (4 w przypadku 4 dostępnych i średnio 7 w przypadku 8), albo tylko jednego. Efektywnym wykorzystaniem procesora wykazały się także algorytmy Pełnego Przeglądu Listy Liczby - ROPPLL i RNPPLL. Dziło się tak dlatego, że nie istniało w nich za dużo elementów wymagających synchronizacji, oraz pracy na jednym rdzeniu, licznie obecnych w przypadku rodziny algorytmów opartych na Sitach Erastotenesa.

2. Używanie tylko jednego wątku, w przypadku algorytmów równoległych możemy tłumaczyć wykonywaniem operacji typowo sekwencyjnych. Mogły to być np. słaszczanie dwuwymiarowych struktur do jednowymiarowych wektorów, rozdział zadań między wątki.
3. Ciekawym porównaniem może być sprawdzenie jak algorytmy równoległe, mając być usprawnionymi wersjami algorytmów sekwencyjnych wypadły w porównaniu z tymi drugimi. Prezentuje to poniższa tabelka.

method	number of threads	range	Elapsed time [s]	Instructions retired	Clockticks	Retiring	Front-end bound	Back-end bound	Memory bound	Core bound	effective physical core utilization	Speed [ilość/s]
Sequential without sieve	1	2...MAX	58,092	2,25E+11	2,29E+11	50.3%	50.9%	0.0%	0.0%	0.0%	24.0%	1,20E+06
		2...MAX/2	21,86	8,38E+10	8,62E+10	49.8%	49.4%	0.4%	0.0%	0.4%	23.5%	1,60E+06
		MAX/2...MAX	36,633	1,41E+11	1,45E+11	48.1%	48.4%	3.2%	0.0%	3.2%	23.7%	9,55E+05
Parallel optimal without sieve	8	2...MAX	15,832	2,29E+11	3,24E+11	58.8%	42.7%	0.0%	4.1%	0.0%	81.3%	4421424,9
		2...MAX/2	5,992	8,68E+10	1,25E+11	55.7%	39.8%	4.1%	0.0%	4.1%	80.2%	5841121,3
		MAX/2...MAX	9,291	1,44E+11	2,14E+11	61.8%	40.1%	0.0%	0.0%	0.0%	86.7%	3767086,5

Figure 51: Porównanie SPPLL oraz ROPPLL

method	number of threads	range	Elapsed time [s]	Instructions retired	Clockticks	Retiring	Front-end bound	Back-end bound	Memory bound	Core bound	effective physical core utilization	Speed [ilość/s]
Sequential with sieve	1	2...MAX	0,579	2,31E+09	1,98E+09	33.2%	13.0%	44.1%	38.3%	5.8%	21.1%	120898098
		2...MAX/2	0,237	1,17E+09	8,59E+08	31.0%	13.0%	41.3%	0.0%	41.3%	24.2%	147679321
		MAX/2...MAX	0,453	1,95E+09	1,58E+09	26.5%	1.8%	66.4%	66.4%	0.0%	24.6%	77262695
Parallel functional naive with sieve	8	2...MAX	0,31	4,00E+09	5,29E+09	46.1%	11.1%	40.2%	17.9%	22.3%	33.8%	225806448
		2...MAX/2	0,181	1,89E+09	2,41E+09	75.4%	25.5%	0.0%	0.0%	0.0%	29.7%	193370160
		MAX/2...MAX	0,171	1,91E+09	2,37E+09	29.5%	22.4%	45.7%	26.6%	19.1%	33.4%	204678368
Parallel domain naive with sieve	8	2...MAX	0,66	6,04E+09	1,19E+10	26.4%	10.1%	62.9%	41.3%	21.5%	36.3%	106060605
		2...MAX/2	0,255	3,23E+09	5,60E+09	19.5%	6.0%	74.0%	43.0%	31.0%	28.7%	137254898
		MAX/2...MAX	0,603	4,71E+09	9,24E+09	33.3%	8.5%	58.2%	44.8%	13.4%	32.4%	58043119

Figure 52: Porównanie SSE, RNSED, RNSEF

Widzimy w niej, że algorytm równoległy jest szybszy od algorytmu sekwencyjnego, co jest oczekiwanym wynikiem.

4. W czasie przeprowadzanych testów zauważyliśmy, że głównym ograniczeniem wpływającym na efektywność naszych algorytmów sekwencyjnych i algorytmów równoległych rodziny Pełnego Przeglądu Listy Liczb, jest ograniczenie *front-end-bound* - ograniczenia wejścia - w granicach ok. 50%. W przypadku pozostałych algorytmów równoległych problematyczne okazały się ograniczenia back-end - wyjścia, oraz memory-bound - pamięci, jaki i core bound - ilość jednostek wykonawczych.
5. Algorytm RNPPLL i ROPPLL okazały się szybsze od SPPLL, jednak w prównaniu z pozostałymi algorytmami znacznie gorsze.

Glossary

PPLL Pełny Przegląd Listy Liczb. 2, 10

RNPPLL Równoległy Naiwny Pełny Przegląd Listy Liczb. 6, 18, 32, 33, 34, 37, 38

RNSED Równoległe Naiwne Sito Erastotenesa w wersji Domenowej. 9, 18, 38

RNSED Równoległe Optymalne Sito Erastotenesa w wersji Domenowej. 10, 18, 19, 29, 30, 31, 37

RNSED Równoległe Optymalne Sito Erastotenesa w wersji Funkcyjnej. 13, 18, 26, 27, 28

RNSEF Równoległe Naiwne Sito Erastotenesa w wersji Funkcyjnej. 13, 18, 23, 24, 25, 38

ROPPLL Równoległy Naiwny Pełny Przegląd Listy Liczb. 8, 18, 35, 36, 37, 38

SE Sito Erastotenesa. 2, 4, 9

SPPLL Sekwencyjny Pełny Przegląd Listy Liczb. 1, 5, 17, 20, 21, 38

SSE Sekwencyjne Sito Erastotenesa. 1, 6, 17, 21, 22, 38

TLP Tablica Liczb Pierwszych. 4

WLU Wektor Liczb Uporządkowanych. 4, 6, 9, 12, 13