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

Celem niniejszego projketu jest stworzenie efektywnego programu wyszukującego liczby pierwsze w zadanym przedziale. W tym celu wykorzystamy porogramowanie 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 sobię 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 zapisywały 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. Kolejość 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 przypadu 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 pozbędziemy się problemu wielodostępu i nigdy nie dopuścimy do spełnienia warunku koniecznego. Narzędziem, które może okazać się przydatne mogą być monitory, lub semafory. W naszym projekcie zastosujemy odpowiednie klauzule, które zapobiegną opisywanemu zjawisku. Będą to klauzule synchronizujące i dzielącę 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 pmięci podręcznej procesora. Dzieje się tak, ponieważ leżą one na tej samej lini 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 lini pamięci. W takim wypadku, zmienna ta zostanie uznana za *"brudną"* i będzie wymagała ponownego pobrania z pamięci.

Zjawisko False Sharingu prowadzi do znaczengo 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 liń pamięci. Co sprawia, że wymagany jest dodatkowy narzut czasowy związnay 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 lini pamięci.

W wyniku eksperymentu przeprowadzonego w ramach zadania pierwszego wyznaczyliśmy wielkość pojedyńczej lini pamięci na 64B

5 Zastosowane Algorytmy

5.1 Opis teoretyczny

5.1.1 Sito Erastotenesa

W swojej najprostszej formie SE jest alogrytmem, który przyjmuje na swoje wejście wektor liczb naturalnych, uporządkowanych rosnąco z krokiem jeden - w dalszych częściach teog 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 złozone 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 przejrzeniu 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 optymalizację jaką można dokonać jest ograniczenie wielkości tablicy liczb peirwszych. 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 jden z najprostszych algorytmów wyszukiwania liczb pierwszych. W swojej sekwencyjnej wersji polega na pełnym przejrzeniu 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 jesst liczbą złożoną. Po zakończeniu iterowania i nie znalezieniu dzielnika, możemy stwierdzić, że sprawdzana liczba jest pierwsza. Optymalizacja algorytmu polega na sprawdzaniu liczby 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 wątli

`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 Przegląd Listy Liczb

Tak jak opisano w części teoretycznej polega na pełnym przeglądzie wszystkich liczb z podanego zakresu. Aby zpoptymalizować 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.

```
std::vector<int> sequential::withoutSieve(int start, int end, bool printOutput){
    int size = end - start + 1;
    std::vector<int> result;
    double timeStart, timeStop;

    result.reserve(size);
    timeStart = omp_get_wtime();
    for (size_t i = start; i <= end; ++i){
        if (isPrime(i)) {
            result.push_back(i);
        }
    }
    timeStop = omp_get_wtime();
    if(printOutput) period(timeStart, timeStop, "sequential::withoutSieve");

    return result;
}
```

Figure 2: Sekwecyjny Przegląd Listy Liczb

5.2.4 Sekwencyjne Sito Erastotenesa

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.

5.2.5 Równoległe Określanie Pierwszości Liczb

Podejście Naiwne

Podejście Optymalne

5.3 Równoległy Pelny 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

```
std::vector<int> sequential::withSieve(int start, int end, bool printOutput){
    int sqrtEnd = (int)sqrt(end);
    double timeStart, timeStop;
    timeStart = omp_get_wtime();
    std::vector<int> primes = sequential::withoutSieve(2, sqrtEnd, false);
    std::vector<bool> isPrime(end + 1, false);
    int primesSize = primes.size();
    for (auto& p: primes){
        int multiple = p * 2;
        while (multiple <= end) {
            isPrime[multiple] = true;
            multiple += p;
        }
    }
    int size = end - start + 1;
    std::vector<int> result;
    result.reserve(size);
    for (size_t i = start; i <= end; i++){
        if (!isPrime[i]) result.push_back(i);
    }
    timeStop = omp_get_wtime();
    if (printOutput) period(timeStart, timeStop, "sequential::withSieve");
        return result;
    return std::vector<int>();
}
```

Figure 3: Sekwencyjne Sito Erastotenesa

sprawdzenia, czy jest pierwsza. Jeżeli taka jest to należy ją zapisać do wektora wyjściowego.

Podejście Naiwne W wersji naiwnej algorytm zapisuje wyniki do jednowymiarowego wektora. Takie rozwiązanie nie jest poprawne. W skrajnych 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 wekotra wyjściowego są ułożone w pamięci jeden obok drugiego, co łatwo prowadzi do uniważniania sąsiednich liń pamięci.

```
std::vector<int> primes(int start, int end, bool printOutput){
    int size = end - start + 1;
    std::vector<int> result;
    double timeStart, timeStop;
    result.reserve(size);
    timeStart = omp_get_wtime();
#pragma omp parallel
    {
#pragma omp for
        for (int i = start; i <= end; i++){
            if (isPrime(i)) result.push_back(i);
        }
    }
    timeStop = omp_get_wtime();
    if (printOutput) period(timeStart, timeStop, "domain::primes");
    return result;
}
```

Podejście Optymalne 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 - conjmniej 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.

```cpp
std::vector<int> primes(int start, int end, bool printOutput){
    int size = end - start + 1;
    std::vector<int> result;
    double timeStart, timeStop;
    int threadsCount = omp_get_max_threads();
    result.reserve(size);
    timeStart = omp_get_wtime();
    std::vector<std::vector<int>> privateResults(threadsCount, std::vector<int>());
    for (int i = 0; i < threadsCount; i++) privateResults[i].reserve(end - start + 1);
    #pragma omp parallel
    {
    #pragma omp for
        for (int i = start; i <= end; i++){
            if (isPrime(i))
                privateResults[omp_get_thread_num()].push_back(i);
        }
    }
    for (size_t i = 0; i < threadsCount; i++)
        result.insert(
            result.end(), privateResults[i].begin(), privateResults[i].end()
        );
    timeStop = omp_get_wtime();
    if (printOutput) period(timeStart, timeStop, "funcional::primes");
    return result;
}
```

Figure 4: Optymalny Równoległy Przegląd Listy Liczb

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 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 wielokrotości 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 W przypadku podejścia domenowego znaczącej zmianie ulega sposób przetrzymywania rozwiązań. Od teraz przechowujemy je w dwuwymiarowym wektorze. Na końcu spłaszczamy tą 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.

```cpp
std::vector<int> domain::basicSieve(int start, int end, bool printOutput){
    double timeStart, timeStop;
    int sqrtEnd = (int)sqrt(end);
    std::vector<int> primes =
         domain::primes(2, sqrtEnd, false);
    std::vector<bool> isPrime(end + 1, false);
    timeStart = omp_get_wtime();
#pragma omp parallel
    {
#pragma omp for schedule(static)
        for (int i = 0; i < primes.size(); i++){
            int multiple = primes[i] * 2;
            while (multiple <= end) {
                isPrime[multiple] = true;
                multiple += primes[i];
            }
        }
    }
    int size = end - start + 1;
    std::vector<int> result;
    result.reserve(size);
    for (int i = start; i <= end; i++){
        if (!isPrime[i]) {
            result.push_back(i);
        }
    }
    timeStop = omp_get_wtime();
    if (printOutput) period(timeStart, timeStop, "domain::basicSieve");
    return result;
}
```

Figure 5: Równoległe Naiwne Sito Erastotenesa w Wersji Domenowej

```cpp
std::vector<int> domain::optimizedSive(int start, int end, bool printOutput){
    double timeStart, timeStop;
    int threadsCount = omp_get_max_threads();
    int sqrtEnd = (int)sqrt(end);
    std::vector<int> primes = domain::primes(2, sqrtEnd, false);
    std::vector<std::vector<bool>> isPrime(threadsCount, std::vector<bool>(end + 1, false));
    timeStart = omp_get_wtime();
#pragma omp parallel
    {
#pragma omp for schedule(dynamic)
        for (int i = 0; i < primes.size(); i++){
            int multiple = primes[i] * 2;
            while (multiple <= end) {
                isPrime[omp_get_thread_num()][multiple] = true;
                multiple += primes[i];
            }
        }
    }
    timeStop = omp_get_wtime();
    int size = end - start + 1;
    std::vector<int> result;
    result.reserve(size);
    for (int i = start; i <= end; i++){
        int sum = false;
        for (int j = 0; j < threadsCount; j++){
            sum |= isPrime[j][i];
        }
        if (!sum) {
            result.push_back(i);
        }

    }
    period(timeStart, timeStop, "domain::optimizedSive");
    return result;
}
```

Figure 6: Równoległe Optymalne Sito Erastotenesa w Wersji Domenowej

5.3.2 Równoległe Sito Erastotenesa - wersja funkcyjna

Zrównoeglenie 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 znaleź w części teoretycznej niniejszego

sprawozdania.

Podejście Naiwne 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 lińi danych, a to doprowadza do strat w efektywności.

Podejście Optymalne Uoptymalnienie

```cpp
std::vector<int> funcional::basicSieve(int start, int end, bool printOutput){
    double timeStart, timeStop;
    int sqrtEnd = (int)sqrt(end);
    std::vector<int> primes =
        funcional::primes(2, sqrtEnd, false);
    std::vector<bool> isPrime(end + 1, false);
    int threadsCount = omp_get_max_threads();
    timeStart = omp_get_wtime();
#pragma omp parallel
    {
        int privateA =
            funcional::getStart(start, end, omp_get_thread_num(), threadsCount);
        int privateB =
            funcional::getEnd(start, end, omp_get_thread_num(), threadsCount);
        for (int i = 0; i < primes.size(); i++){
            int multiple = primes[i] * 2;
            int expr = ((privateA - primes[i] * 2) / primes[i]) * primes[i];
            if (expr > 0) multiple += expr;
            while (multiple <= privateB) {
                isPrime[multiple] = true;
                multiple += primes[i];
            }
        }
    }
    int size = end - start + 1;
    std::vector<int> result;
    result.reserve(size);
    for (int i = start; i <= end; i++) if (!isPrime[i]) result.push_back(i);
    timeStop = omp_get_wtime();
    if (printOutput) period(timeStart, timeStop, "funcional::basicSieve")
    return result;
}
```

Figure 7: Równoegłe Naiwne Sito Erastotenesa w Wersji Funkcyjnej

```cpp
std::vector<int> funcional::optimizedSive(int start, int end, bool printOutput){
    double timeStart, timeStop;
    int threadsCount = omp_get_max_threads();
    int sqrtEnd = (int)sqrt(end);
    std::vector<int> primes =
        funcional::primes(2, sqrtEnd, false);
    std::vector<std::vector<bool>> isPrime(threadsCount, std::vector<bool>(end + 1, false));
    timeStart = omp_get_wtime();
#pragma omp parallel
    {
        int privateA =
            funcional::getStart(start, end, omp_get_thread_num(), threadsCount);
        int privateB =
            funcional::getEnd(start, end, omp_get_thread_num(), threadsCount);
        for (int i = 0; i < primes.size(); i++)
        {
            int multiple = primes[i] * 2;
            int expr = ((privateA - primes[i] * 2) / primes[i]) * primes[i];
            if (expr > 0) {
                multiple += expr;
            }
            while (multiple <= privateB) {
                isPrime[omp_get_thread_num()][multiple] = true;
                multiple += primes[i];
            }
        }
    }
    timeStop = omp_get_wtime();
    int size = end - start + 1;
    std::vector<int> result;
    result.reserve(size);
    for (int i = start; i <= end; i++){
        int sum = false;
        for (int j = 0; j < threadsCount; j++){
            sum |= isPrime[j][i];
        }
        if (!sum){
            result.push_back(i);
        }
    }
    period(timeStart, timeStop, "funcional::optimizedSive");
    return result;
}
```

Figure 8: Równoległe Optymalne Sito Erastotenesa w Wersji Funkcyjnej

6 Prezentacja i Omówienie eksperymentu obliczeniowo-pomiarowego

6.1 Metodyka wykonania eksperymetów

Wszystkie prezentowane eksperymenty zostały uruchomione na maszynie opisanej na początku tego sprawozdania. Do testów zostały użyte wszystkie algorytmy opisane w powyższej części sprawozdania.
Rozmiary instanji 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 porcesorów 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
- secji 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 określenie zmian wydajnościowych w dalszych analizach dla rozwiązań równoległyh.

6.4 Wyniki Uzyskane Dla Algorytmów Równoległych

Glossary

PPLL Pełny Przegląd Listy Liczb. 2, 10

SE Sito Erastotenesa. 2, 4, 9

TLP Tablica Liczb Pierwszch. 4

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