Sprawozdanie z Projektu i Eksperymentu Obliczeniowego

Laboratorium z Przetwarzania Równoległego

Termin oddania: 30.12.2023

Pierwszy termin oddania: 30.12.2023

Wersja I

Autorzy:

1. Wstęp

Celem niniejszego projektu jest analiza i ocena efektywności przetwarzania równoległego w systemie komputerowym z procesorem wielordzeniowym. Projekt koncentruje się na badaniu wydajności obliczeniowej przy wykorzystaniu pamięci współdzielonej w kontekście zastosowania algorytmów równoległych do rozwiązywania problemu znajdowania liczb pierwszych w określonym przedziale. Kluczowym aspektem jest eksploracja różnych strategii zrównoleglenia, w tym podziału domenowego i funkcyjnego, oraz równoważenie obciążeń procesorów przy użyciu odpowiednio skonfigurowanych wątków OpenMP.

1.2 Opis Wykorzystanego Systemu Obliczeniowego: MacBook Air M2

1. **Procesor**:
    - **Oznaczenie**: Apple M2
    - **Liczba procesorów fizycznych**: 1 - integrowany chip Apple M2
    - **Liczba procesorów logicznych**: 8 rdzeni CPU (4 rdzenie wydajnościowe, 4 rdzenie efektywności)
    - **Liczba rdzeni GPU**: 10 rdzeni GPU
2. **Pamięć podręczna procesora**:
    - **Pamięć L1 i L2**: Zintegrowana, specyfikacja nieopisana przez Apple
    - **Pamięć L3**: Zintegrowana, współdzielona, specyfikacja nieopisana przez Apple
3. **System operacyjny**:
    - **Wersja**: MacOs Ventura 13.3
4. **Oprogramowanie do kodowania i testów**:
    - **Nazwa**: Visual Studio Code
5. **Dodatkowe specyfikacje**:
    - **Pamięć RAM**: 8Gb

1.2 Znaczenie i cel ekspreymentu

Projekt ma na celu nie tylko praktyczne zastosowanie teoretycznej wiedzy zdobytej na zajęciach, ale również stanowi okazję do eksploracji nowych technologii i platform sprzętowych. Analiza przeprowadzona na Macbooku Air z procesorem M2 pozwoli na zgłębienie wiedzy na temat możliwości i ograniczeń nowoczesnych technologii Apple w kontekście przetwarzania równoległego.

2. Prezentacja Wariantów Kodów

Wariant 1 - Podejście klasyczne do wyznacznia liczb pierwszych [SP]

```
bool isPrime(int num) {
 if (num <= 1) return false;
 if (num == 2) return true;
 if (num % 2 == 0) return false;

 int limit = std::sqrt(num);
 for (int i = 3; i <= limit; i += 2) {
     if (num % i == 0) return false;
 }
   return true;
}
 int countPrimes(int start, int end) {
     std::vector<int> primes;
     for (int i = start; i <= end; i++) {
         if (isPrime(i)) {
             primes.push_back(i);
         }
     }
     return primes.size();
 }
```

Ten fragment kodu nie wykonywał przetwarzania równoległego. Wykonywał się sekwencyjnie i służył do weryfikacji wyników i obliczania przyspieszenia.

Wariant 2 - Sito Erastotelesa [SS]

```
  std::vector<int> sieveOfEratosthenes(int lower, int upper) {
  std::vector<int> primes;
  std::vector<bool> prime(upper + 1, true);
  prime[0] = prime[1] = false;
  int counter = 0;
  for (int p = 2; p * p <= upper; p++) {
      if (prime[p]) {
        // Pętla zaczyna się od kwadratu liczby pierwszej p lub od
najbliższej większej liczby podzielnej przez p, która jest większa lub
równa M.
        // To zapewnia, że wielokrotności mniejszych liczb pierwszych nie
są usuwane wielokrotnie.
          for (int i = std::max(p * p, (lower + p - 1) / p * p); i <=
upper; i += p)
              prime[i] = false;
      }
  }
  for (int p = lower; p <= upper; p++) {
      if (prime[p])
```

```
            primes.push_back(p);
    }
    return primes;
}
```

Ten fragment kodu również nie wykorzystuje elementów przetwarzania równoległego ,i służy nam za punkt odniesienia w wyznaczaniu przyspieszenia.

Wariant 3 - Rozproszona wersja podejścia klasycznego [RP]

```
bool isPrime(int num) {
    if (num <= 1) return false;
    if (num == 2) return true;
    if (num % 2 == 0) return false;

    int limit = std::sqrt(num);
    for (int i = 3; i <= limit; i += 2) {
        if (num % i == 0) return false;

    }
    return true;
}

void printPrimes(int start, int end) {
    std::vector<int> primes;

    #pragma omp parallel for schedule(dynamic) shared(primes)
    for (int i = start; i <= end; i++) {
        if (isPrime(i)) {
            #pragma omp critical
            {
                primes.push_back(i);
            }
        }
    }

    for (size_t i = 0; i < primes.size(); ++i) {
        std::cout << primes[i] << " ";
        if ((i + 1) % 10 == 0) {
            std::cout << std::endl;
        }
    }
}
```

- **Opis**:
    - Jest to zrównoleglona wersja algorytmu z Wariantu 1. Wszystkie wątki zapisują wyniki swoich obliczeń w jednym wektorze, co wymaga synchronizacji przy użyciu #pragma omp critical.Funkcja ta wykorzystuje podejście domenowe - dane dzielone są między wątki, z których każdy wykonuje swoją część.

- **Podział Pracy**
  - Praca polega na sprawdzaniu pierwszości liczb w zakresie od `start` do `end`.
  - Każda liczba w zakresie jest oddzielnym zadaniem przetwarzania.
- **Sposób Przydziału Zadań do Procesów**
  - Zadania są dynamicznie przydzielane do wątków za pomocą dyrektywy `schedule(dynamic)` w OpenMP.
  - Wątki są przydzielane do nowych zadań, gdy tylko ukończą swoje bieżące zadanie.
- **Uzasadnienie Wybranego Sposobu Podziału Przetwarzania**
  - Dynamiczny przydział zadań zapewnia lepsze zrównoważenie obciążenia między wątkami.
  - Umożliwia efektywniejsze zarządzanie różnicami w czasie potrzebnym do sprawdzenia pierwszości różnych liczb.
- **Dyrektywy i Klauzule OpenMP**
  - `#pragma omp parallel for schedule(dynamic)`: Inicjuje równoległe przetwarzanie pętli for.
  - `#pragma omp critical`: Zapewnia wyłączny dostęp do współdzielonego wektora `primes`, eliminując problem wyścigów danych.
- **Potencjalne Problemy Poprawnościowe**
  - Wyścigi danych są rozwiązane poprzez użycie sekcji krytycznej, która zapobiega jednoczesnym zapisom do wektora `primes`.
- **Potencjalne Problemy Efektywnościowe**
  - **False Sharing**: Ryzyko jest minimalne, ale mogłoby wystąpić przy częstym dostępie do sąsiednich lokalizacji pamięci.
  - **Synchronizacja**: Sekcja krytyczna może wprowadzać narzut synchronizacji, zwłaszcza przy częstym dostępie wielu wątków.

Wariant 4 - Ulepszona wersja RP [URP]:

```
  bool isPrime(int num) {
    if (num <= 1) return false;
    if (num == 2) return true;
    if (num % 2 == 0) return false;

    int limit = std::sqrt(num);
    for (int i = 3; i <= limit; i += 2) {
        if (num % i == 0) return false;

    }
    return true;
  }

void printPrimes(int start, int end) {
```

```
    std::vector<int> primes;

    #pragma omp parallel
    {
        std::vector<int> local_primes;
        #pragma omp for schedule(static)
        for (int i = start; i <= end; i++) {
            if (isPrime(i)) {
                local_primes.push_back(i);
            }
        }

        #pragma omp critical
        {
            primes.insert(primes.end(), local_primes.begin(),
local_primes.end());
        }
    }

    for (size_t i = 0; i < primes.size(); ++i) {
        std::cout << primes[i] << " ";
        if ((i + 1) % 10 == 0) {
            std::cout << std::endl;
        }
    }
}
```

- **Opis**:
    - Jest to ulepszona wersja Wraiantu 3. Każdy wątek operuje na lokalnej wersji tablicy, a pozniej wyniki przetwarzania każdego wątku połączone są w sekcji omp critical.Funkcja ta wykorzystuje podejście domenowe - dane dzielone są między wątki, z których każdy wykonuje swoją część.
- **Podział Pracy**:
    - Zakres liczb dzielony jest na indywidualne zadania, gdzie każde zadanie polega na sprawdzeniu pierwszości konkretnej liczby.
- **Przydział Zadań**:
    - Zadania są statycznie przydzielane do wątków (`schedule(static)`), równomiernie dzieląc zakres między wątki.
- **Uzasadnienie Podziału Przetwarzania**:
    - Statyczny harmonogram wybrany ze względu na prostotę i równomierny podział zadań.
- **Dyrektywy OpenMP**:
    - `#pragma omp parallel`: Inicjuje blok równoległy.
    - `#pragma omp for schedule(static)`: Równomiernie rozdziela iteracje pętli między wątki.

- `#pragma omp critical`: Zapewnia bezpieczeństwo wątków przy scalaniu lokalnych wyników do globalnego wektora `primes`.

- **Kwestie Poprawności**:
  - Unika ryzyka warunków wyścigu dzięki użyciu lokalnych wektorów i sekcji krytycznej.

- **Rozważania Efektywnościowe**:
  - **False Sharing**: Zminimalizowany dzięki lokalnym wektorom dla każdego wątku.
  - **Narzut Synchronizacji**: Występuje podczas scalania lokalnych wektorów, ale jest zredukowany w porównaniu do wcześniejszej wersji.

Wariant 5 - Domenowa wersja rozproszonego sita Erastotelesa [RSD]:

```
void printPrimes(const std::vector<bool>& isPrime, int M, int N) {
    int count = 0, lineCount = 0;
    for (int i = M; i <= N; ++i) {
        if (isPrime[i]) {
            std::cout << i << " ";
            if (++lineCount % 10 == 0) std::cout << std::endl;
            ++count;
        }
    }
    std::cout << "\nZnaleziono " << count << " liczb pierwszych w zakresie
od " << M << " do " << N << "." << std::endl;
}

void sieveOfEratosthenes(int M, int N) {
    std::vector<bool> isPrime(N + 1, true);
    isPrime[0] = isPrime[1] = false;
    int limit = std::sqrt(N);

    #pragma omp parallel for schedule(dynamic)
    for (int p = 2; p <= limit; ++p) {
        if (isPrime[p]) {
          // Pętla zaczyna się od kwadratu liczby pierwszej p lub od
najbliższej większej liczby podzielnej przez p, która jest większa lub
równa M.
          // To zapewnia, że wielokrotności mniejszych liczb pierwszych
nie są usuwane wielokrotnie.
            for (int i = p * p; i <= N; i += p) {
                isPrime[i] = false;
            }
        }
    }

    printPrimes(isPrime, M, N);
}
```

- **Opis**:

- Jest to pierwsza próba zrównoleglenia sita eratostenesa (Wariant 2).Funkcja ta wykorzystuje podejście domenowe - dane dzielone są między wątki, z których każ**dy wykonuje swoją część.

- **Podział Pracy**
  - **Wielkość Zbioru Zadań**: Zbiór zadań obejmuje sprawdzanie każdej liczby od 2 do `N` w celu oznaczenia jej jako pierwszą lub nie.
  - **Sposób Przydziału Zadań do Procesów**: Zadania są przydzielane dynamicznie do dostępnych wątków. Każdy wątek bierze na siebie część zakresu do sprawdzenia, co pozwala na lepsze wykorzystanie zasobów obliczeniowych.

- **Dyrektywy OpenMP**
  - **Wykorzystane Dyrektywy**: `#pragma omp parallel for schedule(dynamic)` jest użyta do równoległego iterowania przez liczby od `M` do pierwiastka z `N`.
  - **Znaczenie Dyrektyw**: Ta dyrektywa pozwala na równoległe przetwarzanie pętli, co znacznie przyspiesza proces identyfikacji liczb pierwszych.

- **Problemy Poprawnościowe**
  - **Wyścigi**: Istnieje potencjalny problem wyścigu związany z aktualizacją wektora `isPrime` w wewnętrznej pętli. Różne wątki mogą próbować jednocześnie zaktualizować ten sam indeks wektora, co może prowadzić do niespójności danych. Rozwiązaniem tego problemu może być użycie atomowych operacji lub struktur synchronizacji.

- **Problemy Efektywnościowe**
  - **False Sharing**: Może występować, gdy wątki modyfikują elementy wektora `isPrime`, które znajdują się w tej samej linii pamięci podręcznej. Ten problem może wpływać na wydajność przez nieefektywne wykorzystanie pamięci podręcznej i zwiększenie opóźnień.
  - **Synchronizacja**: Narzut związany z synchronizacją jest ograniczony dzięki zastosowaniu `schedule(dynamic)` w OpenMP, ale false sharing może nadal wpływać na ogólną wydajność.

Wariant 6 - Ulepszona wersja RSD [RDSC]

```
void printPrimes(const std::vector<bool>& isPrime, int M, int N) {
    int count = 0, lineCount = 0;
    for (int i = M; i <= N; ++i) {
        if (isPrime[i]) {
            std::cout << i << " ";
            if (++lineCount % 10 == 0) std::cout << std::endl;
            ++count;
        }
    }
    std::cout << "\nZnaleziono " << count << " liczb pierwszych w zakresie
od " << M << " do " << N << "." << std::endl;
}

void sieveOfEratosthenes(int M, int N) {
    std::vector<bool> isPrime(N + 1, true);
```

```
    isPrime[0] = isPrime[1] = false;
    int limit = std::sqrt(N);

    #pragma omp parallel for schedule(dynamic)
    for (int p = 2; p <= limit; ++p) {
        if (isPrime[p]) {
          // Pętla zaczyna się od kwadratu liczby pierwszej p lub od
najbliższej większej liczby podzielnej przez p, która jest większa lub
równa M.
          // To zapewnia, że wielokrotności mniejszych liczb pierwszych
nie są usuwane wielokrotnie.
            for (int i = std::max(p * p, (M + p - 1) / p * p); i <= N; i
+= p) {
                #pragma omp critical
                isPrime[i] = false;
            }
        }
    }

    printPrimes(isPrime, M, N);
}
```

- **Opis**
  - Jest to ulepszona werjsa Wariantu 5. Dodana zostala dyrektywa synchronizujaca `#pragma omp critical`. Co przeciwdziała występowaniu wyściugu.
- **Podział Pracy**
  - **Wielkość Zbioru Zadań**: Zadania obejmują identyfikację liczb pierwszych oraz eliminację ich wielokrotności w zakresie od `2` do `N`.
  - **Sposób Przydziału Zadań do Procesów**: Zadania są dynamicznie przydzielane do wątków za pomocą `schedule(dynamic)` w OpenMP, co pozwala na efektywne wykorzystanie zasobów obliczeniowych.
- **Dyrektywy OpenMP**
  - **Wykorzystane Dyrektywy**: `#pragma omp parallel for schedule(dynamic)` jest użyta do równoległego iterowania przez liczby od `M` do pierwiastka z `N`.
  - **Znaczenie Dyrektyw**: Umożliwia równoległe przetwarzanie pętli, przyspieszając proces identyfikacji liczb pierwszych.
- **Problemy Poprawnościowe**
  - **Wyścigi**: Potencjalne wyścigi mogą wystąpić w momencie oznaczania liczby jako niepierwsze (`isPrime[i] = false`). Użycie dyrektywy `#pragma omp critical` likwiduje ryzyko wystąpienia wyścigu, gwarantując bezpieczny dostęp do współdzielonego zasobu.
- **Problemy Efektywnościowe**
  - **False Sharing**: Ryzyko false sharing może być obecne, zwłaszcza gdy wiele wątków próbuje zaktualizować bliskie sobie elementy wektora `isPrime`.

- **Synchronizacja**: Użycie `#pragma omp critical` wewnątrz wewnętrznej pętli wprowadza narzut związany z synchronizacją, co może negatywnie wpłynąć na wydajność przy równoległym przetwarzaniu.

Wariant 7 - Ulepszona Wersja Wariantu 6 [RDSCU]

```
  void sieveOfEratosthenes(int M, int N, std::vector<int>& primes) {
      std::vector<bool> isPrime(N + 1, true);
      isPrime[0] = isPrime[1] = false;
      int limit = std::sqrt(N);
      // Synchroniczne wyznaczenie liczb pierwszych do sqrt(N)
      for (int p = M; p <= limit; ++p) {
          if (isPrime[p]) {
              for (int i = p * p; i <= N; i += p) {
                  isPrime[i] = false;
              }
          }
      }

      #pragma omp parallel
      {
          std::vector<int> localPrimes;
          #pragma omp for nowait
          for (int i = 2; i <= N; i++) {
              if (isPrime[i]) {
                  localPrimes.push_back(i);
              }
          }
          #pragma omp critical
          {
              primes.insert(primes.end(), localPrimes.begin(),
localPrimes.end());
          }
      }

  }
```

- **Opis**
  - Przedstawiony fragment kodu jest ulepszoną wersja Wariantu 6. Każdy wątek działa na lokalnej wersji tablicy isPrime, a wyniki łączone są po wykonaniu pracy przez wątki. Dzięki temu zmniejszony jest nakład związany z synchronizacją wątków przez co algorytm działa efektywniej.
- **Podział Pracy**
  - **Wielkość Zbioru Zadań**:
    - Pierwsza część: oznaczanie niepierwszych liczb w zakresie od `2` do `limit`.
    - Druga część: zbieranie liczb pierwszych z zakresu od `M` do `N`.
  - **Sposób Przydziału Zadań do Procesów**:

        - W pierwszej części zadania nie są równoległe.
        - W drugiej części, zadania są przydzielane równolegle do wątków (`#pragma omp for nowait`).

- **Dyrektywy OpenMP**

    - **Wykorzystane Dyrektywy**:
        - `#pragma omp parallel`: Tworzy równoległy blok dla zbierania liczb pierwszych.
        - `#pragma omp for nowait`: Rozdziela zadanie zbierania liczb pierwszych między wątki, bez czekania na zakończenie wszystkich wątków po zakończeniu pętli.
        - `#pragma omp critical`: Zapewnia bezpieczne scalanie lokalnych wyników do globalnego wektora `primes`.

- **Problemy Poprawnościowe**

    - **Wyścigi**:
        - Nie występują w pierwszej części, ponieważ jest ona sekwencyjna.
        - W drugiej części, sekcja krytyczna (`critical`) zapobiega wyścigom przy scalaniu wyników.

- **Problemy Efektywnościowe**

    - **False Sharing**:
        - Możliwe w drugiej części, gdy różne wątki aktualizują różne części współdzielonego wektora `primes`.
    - **Synchronizacja**:
        - `#pragma omp critical` w drugiej części może wprowadzać narzut związany z synchronizacją podczas scalania wyników.

Wariant 8 - Funkcujna wersja rozproszonego sita Erastotelesa[RSD]:

```
void markMultiples(std::vector<bool>& isPrime, int prime, int N) {
    for (int i = prime * prime; i <= N; i += prime) {
        isPrime[i] = false;
    }
}

void sieve_of_eratosthenes(int M, int N, std::vector<int>& primes) {
    std::vector<bool> isPrime(N + 1, true);
    isPrime[0] = isPrime[1] = false;
    int limit = static_cast<int>(sqrt(N));

    #pragma omp parallel
    {
        // Znajdowanie liczb pierwszych i wykreślanie ich wielokrotności
        #pragma omp for schedule(dynamic)
        for (int p = 2; p <= limit; p++) {
            if (isPrime[p]) {
                markMultiples(isPrime, p, N);
            }
        }
```

```
        // Zbieranie liczb pierwszych
        #pragma omp for schedule(static)
        for (int p = M; p <= N; p++) {
            if (isPrime[p]) {
                #pragma omp critical
                primes.push_back(p);
            }
        }
    }
}
```

- **Opis**
  - Przedstawiony powyżej kod realizuje przetwarzanie równoległe w podejściu funkcyjnym. Polega to na tym że różne wątki mają różne funkcje do wykonania. W jednej części kodu wątki identyfikują liczby pierwsze, a w drugiej zbierają te liczby. Takie rozdzielenie zadań na różne funkcje, wykonywane przez różne wątki, jest charakterystyczne dla podejścia funkcyjnego w przetwarzaniu równoległym.
- **Podział Pracy**
  - **Wielkość Zbioru Zadań**:
    - Pierwsza faza: Identifikacja liczb pierwszych i wykreślanie ich wielokrotności w zakresie od `M` do `limit`.
    - Druga faza: Zbieranie liczb pierwszych w zakresie od `M` do `N`.
  - **Sposób Przydziału Zadań do Procesów**:
    - Przydział dynamiczny dla identyfikacji liczb pierwszych (dyrektywa `schedule(dynamic)`).
    - Przydział statyczny dla zbierania liczb pierwszych (dyrektywa `schedule(static)`).
- **Dyrektywy OpenMP**
  - **Wykorzystane Dyrektywy**:
    - `#pragma omp parallel`: Tworzy blok równoległy obejmujący obie fazy przetwarzania.
    - `#pragma omp for schedule(dynamic)` i `#pragma omp for schedule(static)`: Służą do równoległego przetwarzania dwóch faz algorytmu.
- **Problemy Poprawnościowe**
  - **Wyścigi**:
    - Możliwość wystąpienia wyścigów przy aktualizacji wektora `isPrime`.
    - W drugiej fazie, sekcja krytyczna (`critical`) jest używana do bezpiecznego dodawania liczb pierwszych do wektora `primes`, zapobiegając wyścigom.
- **Problemy Efektywnościowe**
  - **False Sharing**:
    - Potencjalne ryzyko false sharing może wystąpić, jeśli różne wątki próbują zaktualizować sąsiadujące elementy wektora `isPrime`.

- **Synchronizacja**:
  - Użycie `#pragma omp critical` przy dodawaniu liczb pierwszych do wektora `primes` może powodować narzut związany z synchronizacją, wpływając na wydajność.

Wariant 9 - Ulepszona wersja Wariantu 8 [URSD]

```
#include <omp.h>
#include <vector>
#include <iostream>
#include <cmath>

// Funkcja oznaczająca wielokrotności liczby pierwszej jako niepierwsze
void markMultiples(std::vector<bool>& isPrime, int N) {
    int sqrtN = static_cast<int>(sqrt(N));
    for (int p = 2; p <= sqrtN; p++) {
        if (isPrime[p]) {
            for (int i = p * p; i <= N; i += p) {
                isPrime[i] = false;
            }
        }
    }
}

// Funkcja zbierająca liczby pierwsze
void collectPrimes(const std::vector<bool>& isPrime, int M, int N,
std::vector<int>& primes) {
    for (int p = M; p <= N; p++) {
        if (isPrime[p]) {
            primes.push_back(p);
        }
    }
}

void sieve_of_eratosthenes(int M, int N, std::vector<int>& primes) {
    std::vector<bool> isPrime(N + 1, true);
    isPrime[0] = isPrime[1] = false;

    #pragma omp parallel sections
    {
        #pragma omp section
        {
            markMultiples(isPrime, N);
        }

        #pragma omp section
        {
            std::vector<int> localPrimes;
            collectPrimes(isPrime, M, N, localPrimes);
            #pragma omp critical
            {
                primes.insert(primes.end(), localPrimes.begin(),
localPrimes.end());
```

```
            }
        }
    }
}
```

Podział Pracy

- **Opis**
    - Kod efektywnie implementuje sito Eratostenesa z podejściem funkcyjnym, rozdzielając algorytm na dwie odrębne funkcje, które są przetwarzane równolegle. Takie podejście umożliwia lepsze wykorzystanie równoległości i potencjalnie zwiększa wydajność algorytmu, zwłaszcza na systemach wielowątkowych.
- **Wielkość Zbioru Zadań**:
    - Pierwsza faza (`markMultiples`): Oznaczanie liczb niepierwszych w zakresie od 2 do pierwiastka kwadratowego z `N`.
    - Druga faza (`collectPrimes`): Zbieranie liczb pierwszych w zakresie od `M` do `N`.
- **Sposób Przydziału Zadań do Procesów**:
    - Rozdzielenie zadań między dwie sekcje w bloku `#pragma omp parallel sections`, z jednym zadaniem w każdej sekcji.

Dyrektywy OpenMP

- **Wykorzystane Dyrektywy**:
    - `#pragma omp parallel sections`: Umożliwia równoległe wykonanie różnych zadań w różnych sekcjach.
    - `#pragma omp section`: Określa indywidualne sekcje dla zadań `markMultiples` i `collectPrimes`.

Problemy Poprawnościowe

- **Wyścigi**:
    - Potencjalne ryzyko wyścigów jest zminimalizowane dzięki odpowiedniemu podziałowi zadań i użyciu sekcji krytycznej przy scalaniu wyników.

Problemy Efektywnościowe

- **False Sharing**:
    - Ryzyko false sharing jest ograniczone, ponieważ wątki pracują na różnych zadaniach.
- **Synchronizacja**:
    - Użycie sekcji krytycznej w `collectPrimes` do scalania wyników może powodować pewien narzut, ale jest to ograniczone tylko do końcowego etapu algorytmu.

Punkt 3: Prezentacja Wyników i Omówienie Eksperymentu

a) **Testowane Wersje Kodów:** [Wariant 1, Wariant 2, ...].
b) **Tabela Wyników:**
c) **Omówienie Wyników:**

Porównanie Jakości Rozwiązań:[Prędkość przetwarzania vs. czas obliczeń].
Analiza Efektywności Zrównoleglenia: [Przyspieszenie, Efektywność, Prędkość].

Punkt 4: Wnioski

Porównanie Podejść: [Wariant 1 vs. Wariant 2, ...].
Podsumowanie Zrównoważenia Przetwarzania: [Analiza zrównoważenia pracy procesorów].
Ocena Efektywności Skalowania: [Efektywność w zależności od liczby procesorów].
Ograniczenia Efektywnościowe: [Dominujące ograniczenia w kodzie].
(Używanie miar względnych: np. "czas przetwarzania 2 razy krócej" zamiast "o 2 sekundy krócej").