Programowanie Równoległe	Kierunek: Informatyka techniczna	Grupa: 9
Imię i nazwisko: Jakub Świerczyński	Tematy: Lab8 - Zmienne warunku Lab9 - OpenMP - pętle Lab10 - OpenMP zmienne	Termin oddania: 16.12.2024

Laboratorium 8

Część 1: Implementacja bariery

Stworzenie algorytmu realizującego funkcję bariery, w której wątek może zakończyć realizację funkcji dopiero po jej wywołaniu przez wszystkie inne wątki.

Kod bariera.c:

```c
static int threads_to_wait;
static int threads_waiting;
static int generation = 0;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void bariera_init(int n) {
    threads_to_wait = n;
    threads_waiting = 0;
    generation = 0;
}

void bariera(void) {
    pthread_mutex_lock(&mutex);
    int my_generation = generation;

    threads_waiting++;

    if (threads_waiting == threads_to_wait) {
        threads_waiting = 0;
        generation++;
        pthread_cond_broadcast(&cond);
    } else {
        while (my_generation == generation) {
            pthread_cond_wait(&cond, &mutex);
        }
    }

    pthread_mutex_unlock(&mutex);
}
```

Mechanizm:

- Liczba wątków oczekujących i obecna generacja bariery są zarządzane za pomocą zmiennych statycznych.

- Wątki synchronizują się z użyciem muteksa i zmiennej warunku:

    - Wątek zwiększa licznik oczekujących.

    - Jeśli liczba wątków osiągnie oczekiwany próg, zmienna generation jest zwiększana i wątki są wybudzane.

- Pozostałe wątki oczekują na zmianę generacji.

Wynik kodu z barierą oraz bez niej:

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab8/lab_8_bariera# ./bariera 1
przed bariera - watek 0
przed bariera - watek 1
przed bariera - watek 2
przed bariera - watek 3
po barierze - watek 3
po barierze - watek 0
po barierze - watek 1
po barierze - watek 2
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab8/lab_8_bariera# ./bariera 0
przed bariera - watek 0
po barierze - watek 0
przed bariera - watek 1
po barierze - watek 1
przed bariera - watek 2
po barierze - watek 2
przed bariera - watek 3
po barierze - watek 3
```

Część 2: Problem Czytelników i Pisarzy w C

Problem czytelników i pisarzy polega na synchronizacji współbieżnych operacji czytania i pisania na wspólnym zasobie, przy jednoczesnym zapewnieniu wzajemnego wykluczania. Czytelnicy mogą jednocześnie czytać, ale pisarze wymagają wyłącznego dostępu. Ponadto pisarze mają priorytet, aby uniknąć ich głodzenia.

Fragment kodu czytelnia.c:

```c
int my_read_lock_lock(cz_t* cz_p) {
    pthread_mutex_lock(&cz_p->mutex);
    printf("czytelnik próbuje wejść: l_c=%d, l_p=%d\n", cz_p->l_c, cz_p->l_p);
    while (cz_p->l_p > 0 || cz_p->waiting_writers) {
        pthread_cond_wait(&cz_p->czytelnicy_cond, &cz_p->mutex);
    }
    cz_p->l_c++;
    printf("czytelnik wszedł: l_c=%d, l_p=%d\n", cz_p->l_c, cz_p->l_p);
    pthread_mutex_unlock(&cz_p->mutex);
    return 0;
}

int my_read_lock_unlock(cz_t* cz_p) {
    pthread_mutex_lock(&cz_p->mutex);
    cz_p->l_c--;
    printf("czytelnik wychodzi: l_c=%d, l_p=%d\n", cz_p->l_c, cz_p->l_p);
    if (cz_p->l_c == 0) {
        pthread_cond_signal(&cz_p->pisarze_cond);
    }
    pthread_mutex_unlock(&cz_p->mutex);
    return 0;
}
```

Mechanizm:

- Zarządzanie dostępem czytelników i pisarzy:
    - Czytelnicy mogą wchodzić do sekcji krytycznej tylko wtedy, gdy nie ma pisarzy ani oczekujących pisarzy.
    - Pisarze mają priorytet i mogą wchodzić do sekcji krytycznej tylko wtedy, gdy żaden czytelnik nie czyta.

Zmienne synchronizujące:

- mutex: Zapewnia wzajemne wykluczanie dostępu do sekcji krytycznej.
- czytelnicy_cond: Wybudza czytelników, gdy sekcja krytyczna jest dostępna.
- pisarze_cond: Wybudza pisarzy, gdy sekcja krytyczna jest dostępna.

Priorytet pisarzy:

- Użycie zmiennej waiting_writers, aby zasygnalizować obecność oczekujących pisarzy.
- W sekcji krytycznej najpierw obsługiwani są pisarze, a następnie czytelnicy.

Wnioski

Mechanizmy synchronizacji w Pthreads, takie jak mutexy i zmienne warunkowe, są skuteczne w implementacji złożonych problemów współbieżności.

Priorytet dla pisarzy zapobiega głodzeniu, ale wymaga starannego zarządzania kolejkami wątków i budzeniem odpowiednich grup.

Debugowanie współbieżnego kodu jest trudne. Dodanie wydruków kontrolnych i użycie flagi warunkowej #ifdef MY_DEBUG znacznie ułatwia wykrywanie błędów.

Problem czytelników i pisarzy ilustruje kluczowe zasady współdzielenia zasobów, które mają zastosowanie w bazach danych, systemach plików i aplikacjach sieciowych.

Implementacja złożonych protokołów wymaga dokładnego śledzenia stanów zmiennych synchronizacyjnych oraz ich odpowiedniej inicjalizacji.

Laboratorium 9

Temat: Programowanie równoległe z wykorzystaniem OpenMP – zrównoleglenie pętli, klauzule schedule, dekopmozja danych.

Celem ćwiczenia jest nabycie umiejętności tworzenia i implementacji programów równoległych z wykorzystaniem OpenMP, w szczególności zrównoleglanie pętli, używanie klauzul sterujących podziałem pracy (schedule) oraz redukcji (reduction), a także obserwacja wpływu różnych wariantów podziału na rozkład iteracji pomiędzy wątki.

Część 1: Prosty przykład z równoległą pętlą (openmp_petle_simple.c)

W pliku openmp_petle_simple.c zrealizowano inicjalizację tablicy jednowymiarowej i obliczenie sumy jej elementów zarówno sekwencyjnie, jak i równolegle.

Fragment kodu:

```
    // Równoległa pętla z klauzulą reduction i ordered
    double suma_parallel = 0.0;
#pragma omp parallel for default(none) shared(a) reduction(+:suma_parallel) ordered
schedule(static)
    for (int i = 0; i < WYMIAR; i++) {
        int id_w = omp_get_thread_num();
        suma_parallel += a[i];
#pragma omp ordered
        printf("a[%2d] -> W_%1d\n", i, id_w);
    }
```

Wynik działania:

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab9/petle# gcc -fopenmp openmp_petle_simple.c -o openmp_petle_simple
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab9/petle# export OMP_NUM_THREADS=4
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab9/petle# ./openmp_petle_simple
Suma wyrazów tablicy (sekwencyjnie): 156.060000
a[ 0] -> W_0
a[ 1] -> W_0
a[ 2] -> W_0
a[ 3] -> W_0
a[ 4] -> W_0
a[ 5] -> W_1
a[ 6] -> W_1
a[ 7] -> W_1
a[ 8] -> W_1
a[ 9] -> W_1
a[10] -> W_2
a[11] -> W_2
a[12] -> W_2
a[13] -> W_2
a[14] -> W_3
a[15] -> W_3
a[16] -> W_3
a[17] -> W_3

Suma wyrazów tablicy równolegle: 156.060000
```

Mechanizm i obserwacje:

Klauzula default(none) wymusza jawne określenie charakteru zmiennych wewnątrz regionu równoległego.

Klauzula reduction(+:suma_parallel) zapewnia automatyczne zredukowanie lokalnych sum do jednej wartości globalnej po zakończeniu pętli równoległej.

Klauzula ordered w połączeniu z dyrektywą #pragma omp ordered wymusza serializację wybranych sekcji kodu, np. wypisywania wyników w kolejności. Jest to tylko zabieg ilustracyjny.

Część 2: Analiza klauzul schedule z plikiem openmp_petle_simple.c

Zmieniamy warianty klauzuli schedule i obserwujemy podział iteracji.

1. schedule(static, 3)

```
#pragma omp parallel for default(none) shared(a) reduction(+:suma_parallel) ordered
schedule(static,3)
```

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab9/petle# ./openmp_petle_simple
Suma wyrazów tablicy (sekwencyjnie): 156.060000
a[ 0] -> W_0
a[ 1] -> W_0
a[ 2] -> W_0
a[ 3] -> W_1
a[ 4] -> W_1
a[ 5] -> W_1
a[ 6] -> W_2
a[ 7] -> W_2
a[ 8] -> W_2
a[ 9] -> W_3
a[10] -> W_3
a[11] -> W_3
a[12] -> W_0
a[13] -> W_0
a[14] -> W_0
a[15] -> W_1
a[16] -> W_1
a[17] -> W_1

Suma wyrazów tablicy równolegle: 156.060000
```

Domyślna porcja statyczna (bez podania rozmiaru) to równy podział iteracji pomiędzy wątki. Tutaj wymusiliśmy porcje wielkości 3. Podział jest deterministyczny i przy każdym uruchomieniu taki sam.

2. schedule(dynamic, 2)

```
#pragma omp parallel for default(none) shared(a) reduction(+:suma_parallel) ordered
schedule(dynamic,2)
```

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab9/petle# ./openmp_petle_simple
Suma wyrazów tablicy (sekwencyjnie): 156.060000
a[ 0] -> W_2
a[ 1] -> W_2
a[ 2] -> W_3
a[ 3] -> W_3
a[ 4] -> W_0
a[ 5] -> W_0
a[ 6] -> W_1
a[ 7] -> W_1
a[ 8] -> W_2
a[ 9] -> W_2
a[10] -> W_3
a[11] -> W_3
a[12] -> W_0
a[13] -> W_0
a[14] -> W_1
a[15] -> W_1
a[16] -> W_2
a[17] -> W_2

Suma wyrazów tablicy równolegle: 156.060000
```

Iteracje przydzielane są dynamicznie w porcjach po 2. Pierwszy dostępny wątek pobiera kolejne dwie iteracje, po ich wykonaniu pobiera następną porcję i tak dalej.

Rozkład zależy od szybkości poszczególnych wątków i może się różnić między uruchomieniami.

3. schedule(dynamic)

```
#pragma omp parallel for default(none) shared(a) reduction(+:suma_parallel) ordered
schedule(dynamic)
```

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab9/petle# ./openmp_petle_simple
Suma wyrazów tablicy (sekwencyjnie): 156.060000
a[ 0] -> W_1
a[ 1] -> W_3
a[ 2] -> W_0
a[ 3] -> W_2
a[ 4] -> W_1
a[ 5] -> W_3
a[ 6] -> W_0
a[ 7] -> W_2
a[ 8] -> W_1
a[ 9] -> W_3
a[10] -> W_0
a[11] -> W_2
a[12] -> W_1
a[13] -> W_3
a[14] -> W_0
a[15] -> W_2
a[16] -> W_1
a[17] -> W_3

Suma wyrazów tablicy równolegle: 156.060000
```

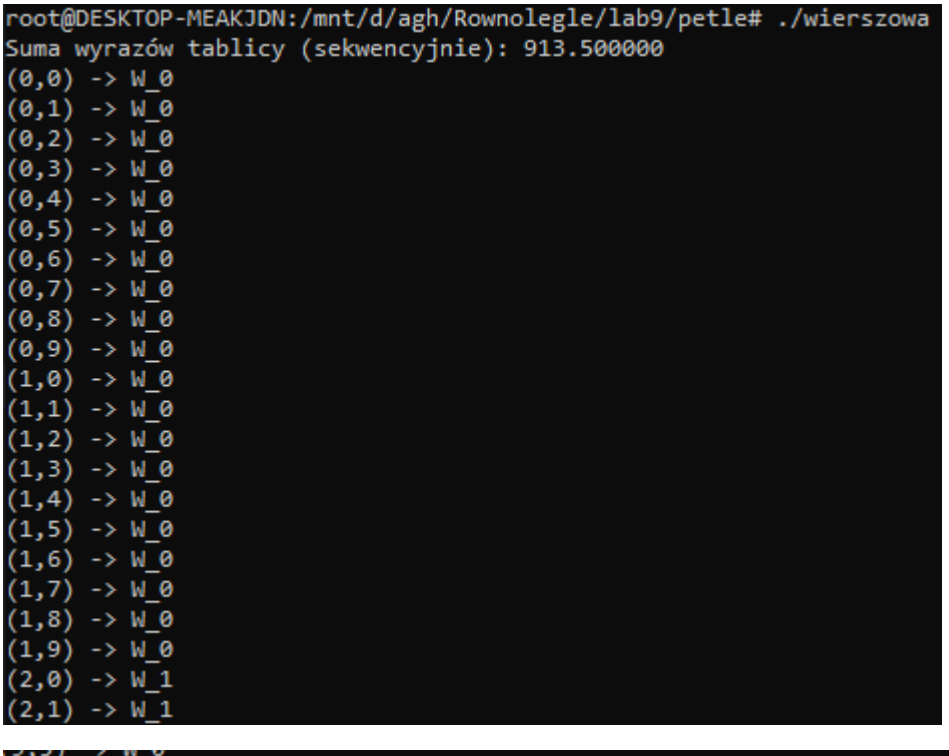
Iteracje przydzielane pojedynczo wątkom w miarę ich dostępności.

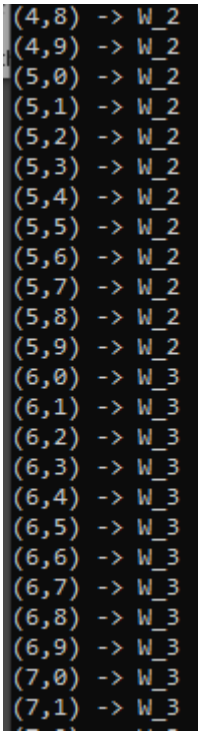
Część 3: Przykład dekompozycji dla tablicy 2D

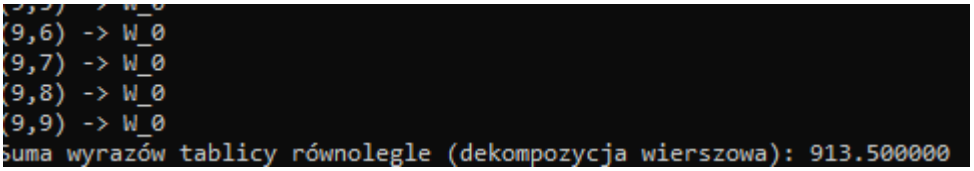
Przykład dekompozycji wierszowej:

W tym przykładzie zrównoleglamy pętlę zewnętrzną (po wierszach). Każdy wątek dostaje określone wiersze zgodnie z klauzulą schedule(static, 2).

```
#pragma omp parallel for default(none) shared(a) reduction(+:suma_parallel)
schedule(static, 2) ordered
    for (int i = 0; i < WYMIAR; i++) {
        int id_w = omp_get_thread_num();
        for (int j = 0; j < WYMIAR; j++) {
            suma_parallel += a[i][j];
#pragma omp ordered
            printf("(%1d,%1d) -> W_%1d\n", i, j, id_w);
        }
    }
```







Widać, że wątki otrzymują zdefiniowane porcje wierszy (np. wątek 0: i=0,1; wątek 1: i=2,1; wątek 2: i=4,8; itd. przy schedule(static,2)).

Zastosowany wariant schedule(static,2) rozdziela wiersze w paczkach po 2. Jest to analogia do przypadku 1D: każdy wątek otrzymuje stały, blokowy fragment wierszy.

Przykład dekompozycji kolumnowej:

Tutaj pętla po kolumnach jest pętlą wewnętrzną i jest zrównoleglana. Używamy klauzuli schedule(dynamic,2) (w kodzie przykładowym jest podane schedule(dynamic, 2)).

```
    for (int i = 0; i < WYMIAR; i++) {
#pragma omp parallel for default(none) shared(a, i) reduction(+:suma_parallel)
schedule(dynamic, 2) ordered
        for (int j = 0; j < WYMIAR; j++) {
            int id_w = omp_get_thread_num();
            suma_parallel += a[i][j];
#pragma omp ordered
            printf("(%1d,%1d) -> W_%1d\n", i, j, id_w);
        }
    }
```

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab9/petle# ./kolumnowa
Suma wyrazów tablicy (sekwencyjnie): 913.500000
(0,0) -> W_0
(0,1) -> W_0
(0,2) -> W_1
(0,3) -> W_1
(0,4) -> W_3
(0,5) -> W_3
(0,6) -> W_2
(0,7) -> W_2
(0,8) -> W_0
(0,9) -> W_0
(1,0) -> W_0
(1,1) -> W_0
(1,2) -> W_1
(1,3) -> W_1
(1,4) -> W_3
(1,5) -> W_3
(1,6) -> W_2
(1,7) -> W_2
(1,8) -> W_0
(1,9) -> W_0
(2,0) -> W_0
(2,1) -> W_0
(2,2) -> W_1
(2,3) -> W_1
(2,4) -> W_2
(2,5) -> W_2
(2,6) -> W_3
```

Każdy wiersz powoduje utworzenie nowego obszaru równoległego. Dla każdej pętli zewnętrznej następuje zakończenie poprzedniego obszaru i synchronizacja.

Podział kolumn między wątki jest dynamiczny po 2 kolumny na raz. Oznacza to, że kolejność przydziału może się zmieniać.

Przykład dekompozycji kolumnowej przez zamianę kolejności pętli i ręczną sumę:

Tutaj dokonujemy zmiany kolejności pętli, aby pętla po kolumnach stała się pętlą zewnętrzną. Użyto klauzuli schedule(static) i tym razem sumowanie wykonujemy ręcznie – każdy wątek ma prywatną sumę, a do sumy globalnej dodajemy wyniki w sekcji krytycznej (lub atomic).

Przykład dekompozycji kolumnowej ze zmianą kolejności pętli:

```
    double suma_parallel = 0.0;
#pragma omp parallel for default(none) shared(a) reduction(+:suma_parallel)
schedule(static) ordered
    for (int j = 0; j < WYMIAR; j++) {
        for (int i = 0; i < WYMIAR; i++) {
            int id_w = omp_get_thread_num();
            suma_parallel += a[i][j];
#pragma omp ordered
            printf("(%1d,%1d) -> W_%1d\n", i, j, id_w);
        }
    }
```

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab9/petle# ./kolumnowa2
Suma wyrazów tablicy (sekwencyjnie): 913.500000
(0,0) -> W_0
(1,0) -> W_0
(2,0) -> W_0
(3,0) -> W_0
(4,0) -> W_0
(5,0) -> W_0
(6,0) -> W_0
(7,0) -> W_0
(8,0) -> W_0
(9,0) -> W_0
(0,1) -> W_0
(1,1) -> W_0
(2,1) -> W_0
(3,1) -> W_0
(4,1) -> W_0
(5,1) -> W_0
(6,1) -> W_0
(7,1) -> W_0
(8,1) -> W_0
(9,1) -> W_0
(0,2) -> W_0
```

```
(3,7) -> W_2
(4,7) -> W_2
(5,7) -> W_2
(6,7) -> W_2
(7,7) -> W_2
(8,7) -> W_2
(9,7) -> W_2
(0,8) -> W_3
(1,8) -> W_3
(2,8) -> W_3
(3,8) -> W_3
(4,8) -> W_3
(5,8) -> W_3
(6,8) -> W_3
(7,8) -> W_3
(8,8) -> W_3
(9,8) -> W_3
(0,9) -> W_3
(1,9) -> W_3
(2,9) -> W_3
(3,9) -> W_3
(4,9) -> W_3
(5,9) -> W_3
(6,9) -> W_3
(7,9) -> W_3
(8,9) -> W_3
(9,9) -> W_3
Suma wyrazów tablicy równolegle (dekompozycja kolumnowa, pętla zewnętrzna): 913.500000
```

Ten przykład pokazuje, jak zmiana kolejności pętli (dekompozycja kolumnowa) oraz równoleglenie pętli zewnętrznej wpływa na podział pracy pomiędzy wątki. Przy statycznym podziale iteracji możemy w przewidywalny sposób przypisać całe kolumny do konkretnych wątków.

Wnioski

- Klauzule schedule w OpenMP umożliwiają elastyczny podział iteracji pętli między wątki.

- Schedule(static) przydziela iteracje w blokach stałych, deterministycznie, co ułatwia przewidywanie i debugowanie.

- Schedule(dynamic) przydziela iteracje w trakcie wykonania, zapewniając bardziej elastyczne równoważenie obciążenia, ale trudniejszą przewidywalność.

- Klauzula reduction ułatwia zbieranie wyników sumowania (lub innych operacji łączenia) bez ręcznej sekcji krytycznej.

- Użycie default(none) wymusza jawne określenie zmiennych i sprzyja czytelności i poprawności kodu.

- Wymuszenie kolejności (ordered) jest przydatne jako narzędzie do debugowania i ilustracji, ale w obliczeniach równoległych zazwyczaj spowalnia wykonanie i nie jest zalecane, jeśli nie jest konieczne.

- Dekompozycja danych (wierszowa i kolumnowa) oraz zmiana kolejności pętli pozwalają dopasować sposób rozdziału pracy do charakterystyki problemu i zasobów sprzętowych.

Laboratorium 10

Temat: Zaawansowane aspekty programowania równoległego z wykorzystaniem OpenMP – zmienne, dyrektywy, deterministyczność wykonania, zmienne threadprivate, zależności danych.

Celem ćwiczenia jest dalsze pogłębienie umiejętności tworzenia programów równoległych w OpenMP, analiza deterministyczności wyników, zastosowanie klauzul takich jak threadprivate, operacji atomowych, sekcji krytycznych, a także rozwiązanie problemów wynikających z zależności danych w pętlach.

Część 1: Analiza openmp_zaleznosci.c

Przed wejściem do obszaru równoległego wypisywane są wartości zmiennych:

- a_shared – zmienna współdzielona (wartość początkowa: 1)
- b_private – zmienna prywatna (wartość początkowa: 2)
- c_firstprivate – zmienna z wartością początkową kopiowaną do każdego wątku (3)
- e_atomic – zmienna współdzielona (5), modyfikowana atomowo

Dla domyślnej liczby wątków:

a_shared po zakończeniu wynosi 41. Każdy wątek w pętli zwiększa a_shared o 10 (w sumie 4 wątki * 10 = 40) oraz była wartość początkowa 1, stąd 1 + 40 = 41.

c_firstprivate w wątkach jest modyfikowane o wielokrotność ID wątku w każdej z 10 iteracji, co skutkuje różnymi wartościami (np. wątek 1: start 3, po 10 dodaniach ID=1 mamy 3+10=13, wątek 2: 3+20=23, itd.).

e_atomic jest zwiększane atomowo o ID wątku w 10 iteracjach na wątek, co daje deterministyczny wynik.

d_local_private ma wartość 4 we wszystkich wątkach. Jest obliczane jako a_shared + c_firstprivate przed modyfikacją a_shared i przy założeniu, że przed startem obliczeń była bariera, zapewniająca spójny stan zmiennych.

Wyniki są deterministyczne przy 4 wątkach.

```
Kompilator rozpoznaje dyrektywy OpenMP
przed wejsciem do obszaru rownoleglego -  nr_threads 1, thread ID 0
        a_shared        = 1
        b_private       = 2
        c_firstprivate  = 3
        e_atomic        = 5

w obszarze równoległym: aktualna liczba watkow 4, moj ID 2
        a_shared        = 41
        b_private       = 0
        c_firstprivate  = 23
        d_local_private = 4
        e_atomic        = 65

w obszarze równoległym: aktualna liczba watkow 4, moj ID 0
        a_shared        = 41
        b_private       = 0
        c_firstprivate  = 3
        d_local_private = 4
        e_atomic        = 65

w obszarze równoległym: aktualna liczba watkow 4, moj ID 3
        a_shared        = 41
        b_private       = 0
        c_firstprivate  = 33
        d_local_private = 23
        e_atomic        = 65

w obszarze równoległym: aktualna liczba watkow 4, moj ID 1
        a_shared        = 41
        b_private       = 0
        c_firstprivate  = 13
        d_local_private = 4
        e_atomic        = 65
po zakonczeniu obszaru rownoleglego:
        a_shared        = 41
        b_private       = 2
        c_firstprivate  = 3
        e_atomic        = 65
```

Część 2: Testowanie z różną liczbą wątków i pojawienie się problemów

Po zwiększeniu liczby wątków (np. OMP_NUM_THREADS=10, OMP_NUM_THREADS=20, OMP_NUM_THREADS=100) zauważono pewien wzorzec. Wywołania programu:

Dla 10 wątków:

```
w obszarze równoległym: aktualna liczba watkow 10, moj ID 9
        a_shared        = 101
        b_private       = 0
        c_firstprivate  = 93
        d_local_private = 4
        e_atomic        = 455
```

- a_shared po obszarze równoległym: 101 (1 + 10 wątków * 10 przyrostów = 1+100)
- e_atomic: 455, co wynika z sumy identyfikatorów wątków pomnożonej przez 10 (każdy wątek 10 razy dodaje swoje ID). Wątek 0 nic nie dodaje (0*10=0), wątek 1 dodaje 10, wątek 2 dodaje 20, ... wątek 9 dodaje 90. Suma 0+10+20+...+90= (10/2)*(0+90)=450, plus wartość początkowa 5 daje 455. Wynik ten jest deterministyczny.
- d_local_private pozostaje 4 we wszystkich wątkach.
- c_firstprivate zależy od ID wątku, np. wątek 9: start 3, +9 w każdej iteracji * 10 iteracji = 3 + 90 = 93.

Dla 20 wątków:

```
w obszarze równoległym: aktualna liczba watkow 20, moj ID 14
        a_shared        = 201
        b_private       = 0
        c_firstprivate  = 143
        d_local_private = 4
        e_atomic        = 1905

w obszarze równoległym: aktualna liczba watkow 20, moj ID 19
        a_shared        = 201
        b_private       = 0
        c_firstprivate  = 193
        d_local_private = 4
        e_atomic        = 1905
```

- a_shared = 201 (1 + 20*10 = 201)
- e_atomic = 1905 – suma ciągu (0+1+2+...+19)*10 + 5 = (19*20/2)*10 +5 = (190)*10+5=1905.
- c_firstprivate odpowiednio rośnie w zależności od numeru wątku.

Dla 100 wątków:

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab10/zmienne/openmp_watki_zmienne# export OMP_NUM_THREADS=100
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab10/zmienne/openmp_watki_zmienne# ./openmp_watki_zmienne

Kompilator rozpoznaje dyrektywy OpenMP
przed wejsciem do obszaru rownoleglego -  nr_threads 1, thread ID 0
        a_shared        = 1
        b_private       = 2
        c_firstprivate  = 3
        e_atomic        = 5

w obszarze równoległym: aktualna liczba watkow 100, moj ID 6
        a_shared        = 1001
        b_private       = 0
        c_firstprivate  = 63
        d_local_private = 4
        e_atomic        = 49505

w obszarze równoległym: aktualna liczba watkow 100, moj ID 28
        a_shared        = 1001
        b_private       = 0
        c_firstprivate  = 283
        d_local_private = 4
        e_atomic        = 49505

w obszarze równoległym: aktualna liczba watkow 100, moj ID 36
        a_shared        = 1001
        b_private       = 0
        c_firstprivate  = 363
        d_local_private = 4
        e_atomic        = 49505
```

- a_shared = 1001 (1 + 100*10)
- e_atomic = 49505 – (0+1+...+99)*10 +5 = (99*100/2)*10+5 = (4950)*10+5=49505.

Analiza wyników dla rosnącej liczby wątków:

- a_shared jest zawsze deterministyczne (1 plus 10 mnożone przez liczbę wątków).
- e_atomic jest również deterministyczne i da się łatwo obliczyć sumując ID wątków.
- c_firstprivate różni się dla poszczególnych wątków zgodnie z oczekiwaniami.
- d_local_private pozostaje stałe (4), co sugeruje, że bariera została umieszczona w odpowiednim miejscu zapewniając spójność wartości a_shared i c_firstprivate przed wyliczeniem d_local_private.

Wnioski z analizy:

- Przy dużej liczbie wątków wartości pozostają deterministyczne, co oznacza, że użyto sekcji krytycznej, bariery i atomów. Dzięki temu brak jest wyścigów danych i wartości zmiennych współdzielonych są zawsze takie same dla różnych uruchomień, bez względu na liczbę wątków.
- Gdyby tych zabezpieczeń nie było (np. brak sekcji krytycznej dla a_shared, brak atomowych operacji dla e_atomic, brak bariery przed obliczeniem d_local_private), moglibyśmy zaobserwować różnice w wynikach w kolejnych uruchomieniach, zwłaszcza przy większej liczbie wątków. Takie sytuacje byłyby przejawem niedeterministycznego działania programu wynikającego z wyścigów danych.

Część 3: Dyrektywa threadprivate

```c
int f_threadprivate;
#pragma omp threadprivate(f_threadprivate)

int main() {
    omp_set_num_threads(5);

    #pragma omp parallel default(none)
    {
        f_threadprivate = omp_get_thread_num();
        #pragma omp critical
        printf("Pierwszy obszar: watek %d, f_threadprivate = %d\n", omp_get_thread_num(),
f_threadprivate);
    }

    #pragma omp parallel default(none)
    {
        #pragma omp critical
        printf("Drugi obszar: watek %d, f_threadprivate = %d\n", omp_get_thread_num(),
f_threadprivate);
    }

    return 0;
}
```

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab10/zmienne/openmp_watki_zmienne# ./threadprivate
Pierwszy obszar: watek 0, f_threadprivate = 0
Pierwszy obszar: watek 2, f_threadprivate = 2
Pierwszy obszar: watek 4, f_threadprivate = 4
Pierwszy obszar: watek 1, f_threadprivate = 1
Pierwszy obszar: watek 3, f_threadprivate = 3
Drugi obszar: watek 2, f_threadprivate = 2
Drugi obszar: watek 0, f_threadprivate = 0
Drugi obszar: watek 1, f_threadprivate = 1
Drugi obszar: watek 3, f_threadprivate = 3
Drugi obszar: watek 4, f_threadprivate = 4
```

W pierwszym obszarze każdy wątek ustawia zmienną f_threadprivate na swój numer wątku.

W drugim obszarze równoległym, mimo że jest to nowy obszar, wartość f_threadprivate pozostała taka sama jak w pierwszym obszarze, ponieważ threadprivate zapewnia, że zmienna jest prywatna dla wątku przez cały czas życia programu, a nie tylko w jednym obszarze.

Wynik: w drugim obszarze każdy wątek drukuje tę samą wartość f_threadprivate, którą nadał sobie w pierwszym obszarze.

Część 4: Usuwanie zależności danych (openmp_zaleznosci.c)

Oryginalna pętla:

```
for(i=0; i<N; i++){
    A[i] += A[i+2] + sin(B[i]);
  }
```

Ma zależność typu RAW: A[i] zależy od A[i+2]. Bezpośrednie zrównoleglenie prowadziłoby do niepoprawnych wyników.

Rozwiązanie:
Używamy tablicy pośredniej temp:

```
  #pragma omp parallel for default(none) shared(A, B, temp) private(i)
  for(i=0; i<N; i++){
      temp[i] = A[i] + A[i+2] + sin(B[i]);
  }
  #pragma omp parallel for default(none) shared(A, temp) private(i)
  for(i=0; i<N; i++){
      A[i] = temp[i];
  }
```

Teraz nie ma zależności między iteracjami pętli, można bezpiecznie zrównoleglić pętle. Suma elementów A po operacji jest taka sama jak w wersji sekwencyjnej.

Wynik:

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab10/pde# export OMP_NUM_THREADS=2
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab10/pde# ./openmp_zaleznosci
suma 1459701.114868, czas obliczen 0.009672
suma 1459701.114868, czas obliczen rownoleglych 0.006962
```

Wnioski

Poprawne użycie sekcji krytycznych, operacji atomowych i barier zapewnia deterministyczność i powtarzalność wyników, niezależnie od liczby wątków.

Klauzule private, firstprivate, shared oraz threadprivate pozwalają na pełną kontrolę nad sposobem inicjalizacji i zarządzania zmiennymi w obszarach równoległych.

Threadprivate zachowuje wartości zmiennych prywatnych między kolejnymi obszarami równoległymi, co jest przydatne w bardziej złożonych scenariuszach.

Eliminacja zależności danych (np. za pomocą tablicy pomocniczej) jest kluczem do uzyskania efektywnej równoległości w przypadku pętli, które sekwencyjnie nie dają się zrównoleglić.

Analiza wyników potwierdza poprawność i deterministyczność działania oraz wskazuje na zwiększoną skalowalność i przewidywalność programu po odpowiednich modyfikacjach.