|  |  |  |
| --- | --- | --- |
| 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:

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:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

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:

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:  
Obraz zawierający tekst, zrzut ekranu

Opis wygenerowany automatycznie

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.

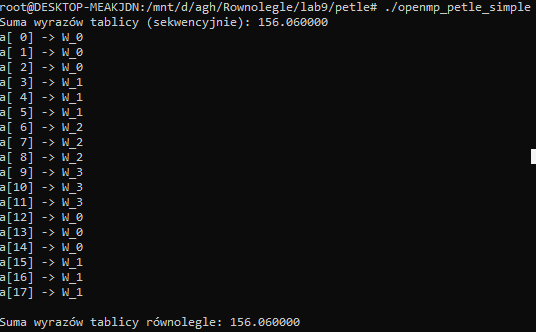
Klauzula ordered w połączeniu z dyrektywą #pragma omp ordered wymusza serializację wybranych sekcji kodu, np. wypisywania wyników w kolejności. Jest to tylko zabieg ilustracyjny.

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)



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.

1. schedule(dynamic, 2)

#pragma omp parallel for default(none) shared(a) reduction(+:suma\_parallel) ordered schedule(dynamic,2)

Obraz zawierający tekst, zrzut ekranu, wyświetlacz, oprogramowanie

Opis wygenerowany automatycznie

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.

1. schedule(dynamic)

#pragma omp parallel for default(none) shared(a) reduction(+:suma\_parallel) ordered schedule(dynamic)

Obraz zawierający tekst, zrzut ekranu, wyświetlacz

Opis wygenerowany automatycznie

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);

        }

    }

Obraz zawierający wiosna, sprężyna

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu

Opis wygenerowany automatycznie

Obraz zawierający zrzut ekranu, tekst

Opis wygenerowany automatycznie

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);

        }

    }

Obraz zawierający tekst, zrzut ekranu

Opis wygenerowany automatycznie

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);

        }

    }

Obraz zawierający tekst, zrzut ekranu

Opis wygenerowany automatycznie Obraz zawierający zrzut ekranu, tekst

Opis wygenerowany automatycznie

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.

Obraz zawierający tekst, zrzut ekranu, Czcionka, menu

Opis wygenerowany automatycznie

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:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

* 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:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

* 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:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Opis wygenerowany automatycznie

* 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

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;

}

Obraz zawierający tekst, zrzut ekranu

Opis wygenerowany automatycznie

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:

Obraz zawierający tekst, zrzut ekranu, Czcionka, linia

Opis wygenerowany automatycznie

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.