

Marek, Kubicki

Programowanie równoległe.

Sprawozdanie z laboratorium 9,10 – OpenMP.

1. Celem tych zajęć było zapoznanie się z działaniem OpenMP(ang. *Open Multi-Processing*) i zależnościami występującymi przy zrównoleglaniu kodu.

OpenMP to API wykorzystywane do zrównoleglania kodu. Może być wykorzystywany na wielu systemach (np. Unix i Microsoft Windows).

Aby wykorzystać OpenMP należy umieścić odpowiednie dyrektywy w kodzie a następnie skompilować go z poleceniem **-fopenmp** aby linker podłączył odpowiednią bibliotekę. W kodzie fragmenty kodu przeznaczone do zrównoleglania należy oznaczyć odpowiednimi dyrektywami w poniższym formacie (dla C i C++):

#pragma omp nazwa_dyrektywy lista_klauzul znak_nowej_linii

Od dyrektywy i klauzul zależy sposób zrównoleglania. Każda z dyrektyw ma listę dopuszczalnych klauzul. Zmienne istniejące przed obszarem równoległym lub zmienne statyczne są wspólne, a te zadeklarowane za pomocą **threadprivate**, umieszczone w klauzuli **private**, zdefiniowane wewnątrz obszaru równoległego lub sterujące pętlą **for** są prywatne.

Dyrektywy (ważne klauzule opisane poniżej):

parallel - Definiuje obszar równoległy za pomocą klamerek **{}**. Polecenia w obszarze równoległym będą wykonane przez każdy wątek.

for – Dzieli zadania w pętli znajdującej się bezpośrednio pod **#pragma** dla istniejących wątków. Pętla nie może zakończyć się za pomocą **break**.

parallel for – Łączy funkcjonalność dwóch powyższych dyrektyw umożliwiając zdefiniowanie pojedynczej zrównoleglonej pętli za pomocą jednego polecenia.

critical – Tworzy sekcję krytyczną zamkniętą w klamerkach **{}**. Tylko jeden wątek może znajdować się w danej sekcji krytycznej.

atomic – Umożliwia na atomowy dostęp do danej zmiennej. Działa jako szybsza sekcja krytyczna dla danej operacji. Najlepiej jest stosować **atomic** zamiast **critical** jeżeli dana operacja jest obsługiwana.

flush – Służy do synchronizacji widoku wątku na dzieloną pamięć.

Barrier – Tworzy barierę w sekcji równoległej służącą do synchronizacji wątków. Wątki wznowiają działanie kiedy wszystkie dotrą do bariery.

Ordered – Dyrektywa ta występuje wyłącznie wewnątrz obszaru **omp for** lub **omp parallel for** z klauzulą **ordered**. Ta dyrektywa sygnalizuje że następne polecenie musi być wykonane w kolejności iteracji pętli.

Single – Deklaruje obszar kodu zamknięty w klamerkach **{}** który wykonany będzie przez jeden wątek.

Master - Deklaruje obszar kodu zamknięty w klamerkach **{}** który wykonany będzie przez **"master thread"** czyli wątek główny, który otworzył sekcję równoległą.

Sections – Służy do zadeklarowania sekcji kodu (za pomocą **section**) do wykonania przez istniejące wątki z barierą istniejącą po wykonaniu sekcji.

Parallel sections – podobnie jak **parallel for**, ta dyrektywa łączy dyrektywy **parallel** i **sections**.

Task – tworzy zadania dla wątki przeznaczone do wykonania asynchronicznie. Zadanie może być wykonane natychmiastowo, lub później w tym samym obszarze równoległym.

Taskyield – oznacza miejsce w którym zadanie może być wstrzymane w celu uruchomienia innego zadania.

Taskwait – oczekiwanie na zakończenie wykonywania utworzonych zadań. Dyrektywa **barrier** także wymusza dokończenie istniejących zadań.

Klauzule dla zmiennych:

Private (list) – zmienne przekazane do tej klauzuli będą prywatne dla każdego wątku, oznacza to że każdy wątek będzie miał swoją zmienną której wartość może różnić się od wartości innych zmiennych.

Firstprivate (list) – podobne do **private**. Różni się tym że zmienna będzie inicjalizowana wartością sprzed obszaru równoległego.

Lastprivate (list) – podobne do **private**. Zmienna będzie miała przypisaną wartość po zakończeniu obszaru równoległego za pomocą wartości z ostatniej iteracji.

Reduction (operator: list) – Zmienne są tworzone dla każdego wątku a ich wynik jest łączony w sposób zdefiniowany za pomocą operatora.

Shared (list) - przekazane zmienne będą dzielone dla wszystkich wątków.

Inne klauzule:

Schedule – definiuje typ podziału pracy dla pętli **for**.

Typy podziału pracy:

Auto – determinowane przez system.

Dynamic – podział dynamiczny, bloki są przydzielane wątkom według zasady „**first-come, first-do**”. Efektem jest to że nie wiadomo który wątek wykona które zadanie.

Guided – działa jak **dynamic** z rozmiarem bloków progresywnie zmniejszającym się aż do minimalnej wartości, domyślnie 1,

Runtime – odczytuje zasadę dzielenia pracy z zmiennej środowiskowej **OMP_SCHEDULE**

Static – każdy wątek ma przydzielone zadanie (lub zadania). Domyślnie każdy wątek ma jedno zadanie, a zadania mają jak najbardziej zbliżoną wielkość.

Wielkość bloków dla tych podziałów może być zdefiniowana poprzez oddanie dodatniej zmiennej **int** jako argument do **schedule**

If(warunek) – jeżeli warunek nie zostanie spełniony, polecenie nie zostanie wykonane.

num_threads (liczba) – jeden z sposobów na określenie liczby wątków. Liczbe wątków można określić także za pomocą zmiennej środowiskowej (**\$ export OMP_NUM_THREADS=liczba**) albo procedury (**omp_set_num_threads(liczba)**).

Nowait - zastępuje barierę niejawną.

Wykosztowanie wielu wątków wiąże się z możliwością wystąpienia zależności, które mogą wpłynąć na wartości zmiennych.

Typowe Zależności to:

Zależności wyjścia: zapis po zapisie, write-after-write:

a = ...

a= ...

Wartość zmiennej jest nadpisywana wiele razy, tylko ostatnia wartość zostaje zapisana.

Aby uniknąć takich zależności należy pilnować kolejności zapisu przy zrównolegleniu.

Anty zależności: zapis po odczycie, write-after-read:

... = a

a= ...

Kiedy operacja zapisu miała być zrealizowana po operacji odczytu, a zrealizowane jest przed nią, możliwe jest że ostateczna wartość zmiennej będzie różnić się od docelowej wartości.

Aby uniknąć takich zależności można przemianować zmienne.

Zależności żeczywiste: zapis po odczycie, read-after-write:

... = a

a= ...

Takie zależności są trudniejsze do wyeliminowania.

Zależności mogą występować jawnie(jak powyżej), lub niejawnie, np. w pętli.

laboratorium 9

Na laboratorium 9 zadaniem było zmodyfikowanie podanych programów. Trzeba było wykorzystać dyrektywę **parallel for** z różnymi opcjami dzielenia zadań, liczba wątków to 4. Dla każdego wariantu suma wyrazów tablicy była zgodna z przewidywaną liczbą 156.

Przykład pętli wraz z **pragma omp** bez **Schedule**:

```
#pragma omp parallel for default(none) shared(suma_par, a) private(suma_parallel)
num_threads(4) ordered
for(int i=0;i<WYMIAR;i++){
```

```

    suma_parallel += a[i];
#pragma omp ordered
    printf("a[%2d]->W_%1d \n",i,omp_get_thread_num());
#pragma omp critical
{
    suma_par = suma_par + a[i];
}
}

```

Ciało pętli nie zmienia się pomiędzy wariantami.

Brak	static	Static,3	dynamic	Dynamic,3
a[0] ->W_0	a[0] ->W_0	a[0] ->W_1	a[0] ->W_1	a[0] ->W_1
a[1] ->W_0	a[1] ->W_0	a[1] ->W_1	a[1] ->W_0	a[1] ->W_1
a[2] ->W_0	a[2] ->W_0	a[2] ->W_1	a[2] ->W_2	a[2] ->W_1
a[3] ->W_0	a[3] ->W_0	a[3] ->W_3	a[3] ->W_3	a[3] ->W_3
a[4] ->W_0	a[4] ->W_0	a[4] ->W_3	a[4] ->W_1	a[4] ->W_3
a[5] ->W_1	a[5] ->W_1	a[5] ->W_3	a[5] ->W_0	a[5] ->W_3
a[6] ->W_1	a[6] ->W_1	a[6] ->W_0	a[6] ->W_2	a[6] ->W_0
a[7] ->W_1	a[7] ->W_1	a[7] ->W_0	a[7] ->W_3	a[7] ->W_0
a[8] ->W_1	a[8] ->W_1	a[8] ->W_0	a[8] ->W_1	a[8] ->W_0
a[9] ->W_1	a[9] ->W_1	a[9] ->W_2	a[9] ->W_0	a[9] ->W_2
a[10] ->W_2	a[10] ->W_2	a[10] ->W_2	a[10] ->W_2	a[10] ->W_2
a[11] ->W_2	a[11] ->W_2	a[11] ->W_2	a[11] ->W_3	a[11] ->W_2
a[12] ->W_2	a[12] ->W_2	a[12] ->W_1	a[12] ->W_1	a[12] ->W_1
a[13] ->W_2	a[13] ->W_2	a[13] ->W_1	a[13] ->W_0	a[13] ->W_1
a[14] ->W_3	a[14] ->W_3	a[14] ->W_1	a[14] ->W_2	a[14] ->W_1
a[15] ->W_3	a[15] ->W_3	a[15] ->W_3	a[15] ->W_3	a[15] ->W_3
a[16] ->W_3	a[16] ->W_3	a[16] ->W_3	a[16] ->W_1	a[16] ->W_3
a[17] ->W_3	a[17] ->W_3	a[17] ->W_3	a[17] ->W_0	a[17] ->W_3

Pragmy dla powyższych wariantów:

Wariant 1: **static**

```

#pragma omp parallel for schedule(static) default(none) shared(suma_par, a)
private(suma_parallel) num_threads(4) ordered

```

Wariant 2: **static,3**

```

#pragma omp parallel for schedule(static,3) default(none) shared(suma_par, a)
private(suma_parallel) num_threads(4) ordered

```

Wariant 3: **dynamic**

```
#pragma omp parallel for schedule(dynamic) default(none) shared(suma_par, a)
private(suma_parallel) num_threads(4) ordered
```

Wariant 4: **dynamic,3**

```
#pragma omp parallel for schedule(dynamic,3) default(none) shared(suma_par, a)
private(suma_parallel) num_threads(4) ordered
```

laboratorium 10

To laboratorium polegało na zidentyfikowaniu i wyeliminowaniu błędów w działaniu programu.

Poprawne wartości zmiennych dla wątków i po zakończeniu obszaru równoległego. **c_firstprivate** nie ma stałej wartości dla wątków (celowo).

```
w obszarze równoległym: aktualna liczba watkow 4, moj ID 3
    a_shared      = 41
    b_private     = 0
    c_firstprivate = 33
    d_local_private = 4
    e_atomic      = 65
po zakonczeniu obszaru rownoleglego:
    a_shared      = 41
    b_private     = 2
    c_firstprivate = 3
    e_atomic      = 65
```

Fragment kodu z poprawionymi błędami

```
int i;//to jest linijka 28
int d_local_private;
d_local_private = a_shared + c_firstprivate;
//WAR na a_shared z linijka 31 - 37, rozwiazane przez bariere
#pragma omp barrier
#pragma omp critical
{
    for(i=0;i<10;i++){
        a_shared ++;
        //WAR a_shared z printf linijka 37 - 61, rozwiozane przez bariere linijka 55
        //WAR WAW RAW a_shared - a_shared linijka 37
    }
}

for(i=0;i<10;i++){
    c_firstprivate += omp_get_thread_num();
}

for(i=0;i<10;i++){
    #pragma omp atomic
    e_atomic+=omp_get_thread_num();
}
```

```

//WAR a_shared z printf linijka 49 - 61, rozwiazane przez barriere linijka 55
//WAR WAW RAW e_atomic - e_atomic linijka 49
}

{
    #pragma omp barrier //WAR oczekiwanie na zakonczenie obliczen przed wypisywaniem,
    rozwiazanie problemow z zmiennymi shared z lini 37 i 49
    #pragma omp critical //zabezpieczenie przed nalozeniem sie printfow
    {
        printf("\nw obszarze równoległym: aktualna liczba watkow %d, moj ID %d\n",
            omp_get_num_threads(), omp_get_thread_num());

        printf("\ta_shared \t= %d\n", a_shared);
        printf("\tb_private \t= %d\n", b_private);
        printf("\tc_firstprivate \t= %d\n", c_firstprivate);
        printf("\td_local_private = %d\n", d_local_private);
        printf("\te_atomic \t= %d\n", e_atomic);
    }
}

```

Pierwszy błąd w tym kodzie wynikał z zależności **WAR** polegającej na zwiększaniu wartości zmiennej **a_shared** przed zapisaniu jej wartości do **d_local_private**. Ten błąd rozwiązałem za pomocą utworzeniu bariery bezpośrednio po operacji na zmiennej **d_local_private**, można to było także rozwiązać za pomocą zapisaniu wartości zmiennej **a_shared** przed blokiem równoległym do innej zmiennej i przekazaniu tej zmiennej do bloku.

Dwa następne błędy wynikały z zwiększania zmiennych **a_shared** i **e_atomic** w pętlach for. Efektem były zależności **WAR**, **WAW** i **RAW**. Problemy ten można rozwiązać za pomocą utworzenia sekcji krytycznej zawierającej iterację zmiennej lub całą pętlę for, zmianę iteracji na operację atomową(szybsze), lub, przy lekkiej zmianie kodu, zmianę iterowanej zmiennej na zmienną prywatną i późniejszym zapisywaniu wartości do zmiennej wspólnej.

Ostatni błąd wynikający z złego zarządzania zmiennymi polegał na wypisywaniu wartości zmiennych przed zakończeniem obliczeń (**WAR**). Rozwiązałem ten problem za pomocą utworzenia bariery po zakończeniu obliczeń.

Sekcja krytyczna na wypisywaniu zapobiega błędom w wypisywaniu na konsole.