Open MP wer. 2.5

Rafał Walkowiak
Instytut Informatyki Politechniki Poznańskiej
Wiosna 2020

OpenMP

standard specyfikacji przetwarzania współbieżnego

- uniwersalny i przenośny model równoległości (typu rozgałęzienie – połączeni) dla architektur z pamięcią współdzieloną
  - przetwarzanie rozpoczyna się od jednego wątku przetwarzania
  - *regiony współbieżne* (parallel regions) wymagają powstania nowych wątków, obowiązują ścisłe zasady określenia liczby wątków
  - wątki łączą się – synchronizują się przy końcu *regionu współbieżnego*
  - określona pamięć dostępna dla wszystkich wątków (współdzielona)
  - dla komputerów typu SMP (ang. Symmetrical Multi Processor) - systemy z pamięcią współdzieloną
  - możliwy do implementacji w większości platform obliczeniowych
  - dostarczany w postaci API (interfejsu dla programowania aplikacji)

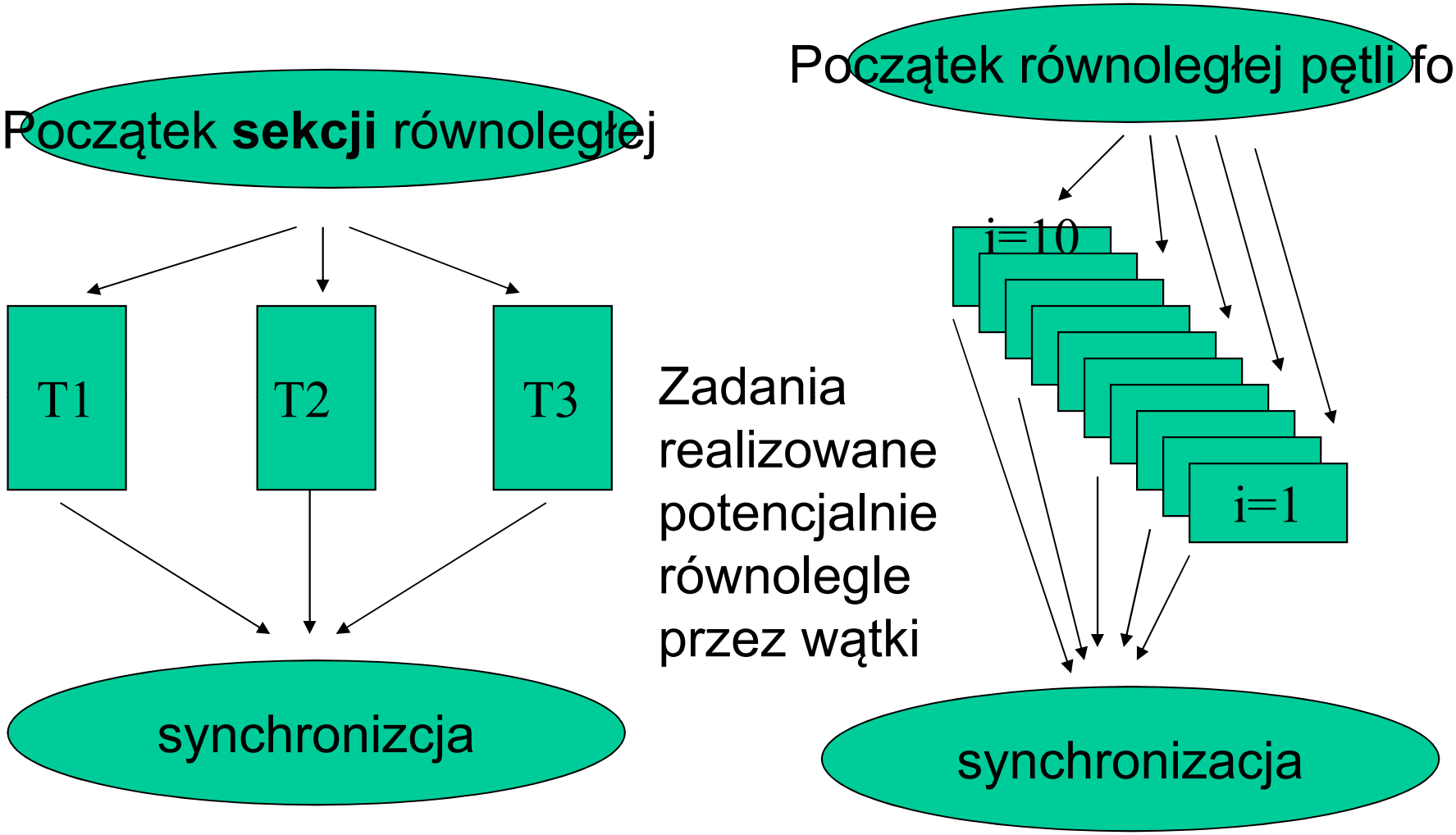
Synchronizacja jako element struktury kodu

- Synchronizacja:
  - zapewnia powiązań części programu równoległego,
  - zapobiega potencjalnie niebezpiecznym dla poprawności dostępom do pamięci np. nadpisanie wyniku, brak danej itp.
  - synchronizacja wybranych zdarzeń przetwarzania,
  - Globalna synchronizacja - bariera synchronizacyjna dla zbioru wątków,
  - specyfikowana wprost (dyrektywy i synchronizacyjne funkcje OpenMP) i specyfikowana nie wprost (wbudowana w dyrektywy o innym zadaniu podstawowym: parallel, for).

Poziomy równoległości w OpenMP

- wysoki poziom równoległości przetwarzania -
  - program podzielony na segmenty, które mogą być realizowane współbieżnie różne watki a więc różne procesory,
- niski poziom równoległości przetwarzania -
  - współbieżna realizacja przez różne wątki iteracji pętli (potencjalnie współbieżnie przez różne procesory)
  - współbieżność na poziomie wykonania puli zadań Open MP wersja >= 3.0

Poziomy równoległości OpenMP 2.0



Projektowanie przetwarzania w Open MP – program równoległy - jak napisać?

- Dyrektywy dla kompilatora:

  #pragma omp *nazwa_dyrektywy klauzule*
- Funkcje biblioteki czasu wykonania
  - określenie/poznanie liczby wątków
  - poznanie numeru wątku
  - poznanie liczby wykorzystywanych procesorów
  - funkcje blokady

Projektowanie przetwarzania w OpenMP

- Sterowanie wykonaniem
  - parallel, for, sections, single, master (dyrektywy)
- Specyfikacja danych
  - shared, private, reduction
- Synchronizacja
  - często wbudowana na początku i końcu bloków kodu podlegającym dyrektywom sterującym,
  - specyfikacja wprost – zastosowanie dyrektyw: barrier, critical, atomic, flush, ordered;

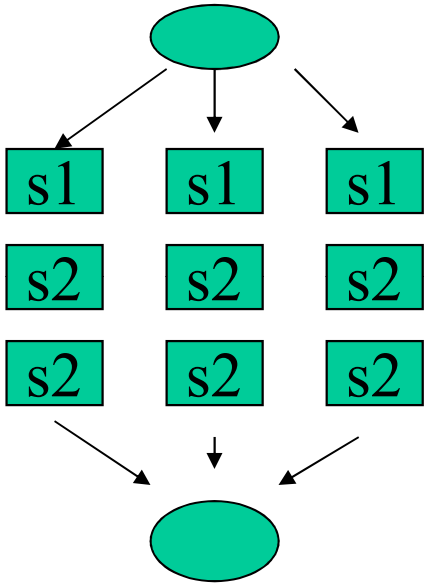
Dyrektywy sterujące przetwarzaniem (1)

- parallel – podstawowa dyrektywa kodu równoległego
  - dyrektywa **tworząca** związany z nią zespół wątków (team) realizujących współbieżnie region równoległy czyli blok strukturalny następujący bezpośrednio po dyrektywie; definiująca **współbieżnie** realizowany fragment kodu; blok strukturalny posiada jedno wejście, jedno wyjście.
- for (dyrektywa wewnątrz regionu równoległego bezpośrednio przed pętlą **for**)
  - dyrektywa określająca sposób przydziału **iteracji pętli for -** każda iteracja wykonywana jest raz - potencjalnie równolegle z innymi iteracjami – np. realizowanymi przez inne wątki na tym samym lub innych procesorach.
- sections, section (wewnątrz regionu równoległego)
  - **sections** – definiuje początek zbioru bloków kodu (każdy z nich oznaczany **section)** realizowanych każdy jednokrotnie przez jeden z wątków ze zbioru aktywnych wątków (potencjalnie współbieżnie).

Dyrektywy sterujące przetwarzaniem (2)

- single
  - dyrektywa definiująca blok kodu realizowany przez tylko jeden, dowolny wątek związany z bieżącym regionem II, przy końcu bloku kodu wbudowana bariera synchronizacyjna,
- master
  - dyrektywa definiująca blok kodu realizowany jednokrotnie – tylko przez wątek główny (id=0), bez bariery synchronizacyjnej,
- parallel for
  - fragment kodu realizowany współbieżnie będący jednym blokiem **for**,
- parallel sections
  - fragment kodu realizowany współbieżnie będący jednym blokiem **sections**

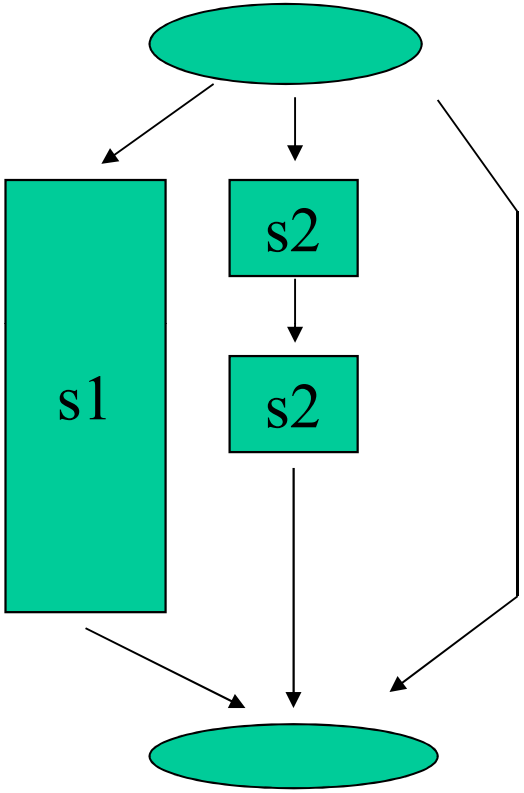
Dyrektywy sterujące przetwarzaniem (3)




```
#pragma omp parallel
        {
        s1();
         for ( int i =1; i<3, i++)
                s2();
        }
```

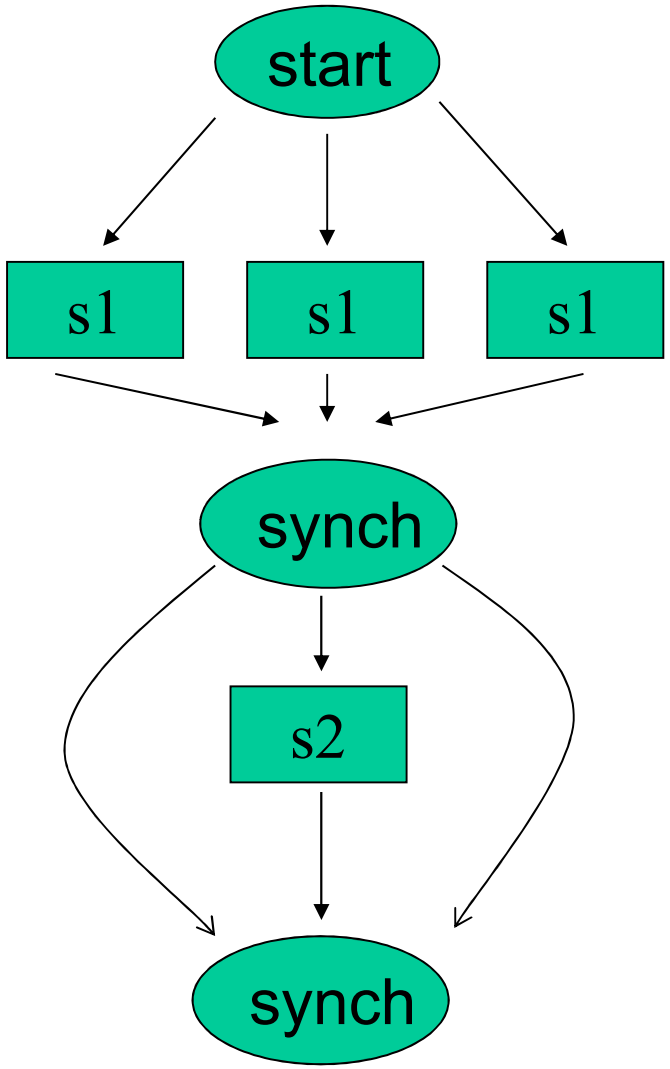
//zakładamy, że tworzone są 3 wątki

Dyrektywy sterujące przetwarzaniem (4)




```
#pragma omp parallel sections
{
#pragma omp section
        s1();
#pragma omp section
        for (i =1; i<3, i++)
                s2();
}
//Zakładamy, że tworzone są 3 wątki
```

Dyrektywy sterujące przetwarzaniem (5)




```
#pragma omp parallel
        {
        #pragma omp for
         for (i =1; i<4, i++)
                s1();
//synchronizacja od #for
        #pragma omp single
                s2();
//synchronizacja od #single
        }
```

Szeregowanie pętli **for** (1)

- Zakres iteracji pętli znany (ustalony w run-time, przy wejściu do pętli)
- Szeregowanie statyczne
  - –zakres iteracji pętli jest podzielony na podzakresy iteracji o jednakowym rozmiarze określonym jako *x* w *schedule(static, x),* liczba i wielkość zakresów przydzielanych do każdego wątku z góry ustalona, domyślnie każdy wątek dostaje jeden zakres, wątek 0 od początku zakresu iteracji;
  - –bazuje na zakresie pętli wyznaczonym w czasie przetwarzania.
- Szeregowanie dynamiczne
  - –wątki zwracają się do modułu szeregującego w celu uzyskania pierwszego i po jego wykonaniu kolejnego zakresu iteracji do realizacji, specyfikowany rozmiar *x* w *schedule(dynamic, x),* domyślnie wielkość przydziału = 1, możliwy proporcjonalnie znaczący koszt czasowy szeregowania w przypadku wielkości przydziału 1 i prostego przetwarzania w ramach ciała pętli.

Szeregowanie pętli **for** (2)

- Szeregowanie sterowane
  - wielkość zakresu wyznaczana na bieżąco maleje w zależności od pozostałej do realizacji liczby iteracji schedule(guided)
- Szeregowanie realizowane w sposób określany w czasie przetwarzania schedule(runtime)
  - decyduje aktualna wartość zmiennej środowiska OMP_SCHEDULE

Synchronizacja w OpenMP (1)

- Wewnętrzne bariery synchronizacyjne - oczekiwanie (koniec bloku kodu) aż wszystkie wątki ze zbioru zrealizują kod bloku poprzedzonego jedną z dyrektyw:
  - *for*
  - *sections*
  - *single*
- Klauzula *nowait* likwiduje barierę synchronizacyjną

Synchronizacja w OpenMP (2)

```
#pragma omp parallel
{
#pragma omp for nowait
    for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1])/2.0

#pragma omp for
    for (i=0; i<m; i++)
    y[i]=sqrt(z[i]);
}
```

klauzula *nowait* zapobiega barierze synchronizacyjnej po zakończeniu bloku poprzedzonego dyrektywą *for*

Dyrektywy synchronizacyjne (1)

barrier

– synchronizacja wszystkich wątków z bieżącego zbioru (#pragma omp parallel), wątki które dotarły w przetwarzaniu do dyrektywy oczekują na pozostałe wątki; po osiągnięciu miejsca przez wszystkie rozpoczynają one współbieżnie przetwarzanie następnego po dyrektywie wyrażenia;

Dyrektywy synchronizacyjne (3)

Flush – synchronizacja danych współdzielonych wątków

- zapewnia spójny obraz pamięci w ramach prywatnego stanu danych wątku realizującego działania dyrektywy i pamięci wspólnej wątków,
- dotyczy zmiennych wyspecyfikowanych jako parametry (lub wszystkich zmiennych współdzielonych występujących w widzianym przez wątek prywatnym stanie danych programu);
- w efekcie oznacza że, wszystkie wcześniejsze operacje wątku realizującego flush odwołujące się do (podanych) zmiennych się zakończyły, a następne jeszcze nie zaczęły,
- możliwe jest, że operacje realizacji dyrektywy wystąpią w czasie później niż miejsce pojawienia się dyrektywy w kodzie, ale przed pierwszą operacją na zmiennej objętej dyrektywą;
- wymaga, aby kompilator spowodował zapis (do pamięci wspólnej wątków) zmiennych zmodyfikowanych w tym wątku, a przechowywanych w ramach prywatnego stanu danych (np. w rejestrach),
- usuwa zmienne współdzielone wątków z prywatnego stanu danych wątku i wymaga aby wątek ponownie załadował ich wartości z pamięci przed ich wykorzystaniem w kodzie.

Dyrektywy synchronizacyjne (3)

Flush

- automatycznie realizowana w ramach:
- *barrier,*
- *critical* we/wy bloku kodu,
- *ordered* we/wy bloku kodu,
- *parallel* wy bloku kodu,
- *for* wy bloku kodu,
- *sections* wy bloku kodu,
- *single* wy bloku kodu,
- atomic we/wy (w atomic dotyczy tylko zmiennej uaktualnianej)

Dyrektywy synchronizacyjne (2)

critical etykieta

- specyfikuje blok, który może być realizowany jednocześnie tylko przez jeden wątek programu, wzajemne wykluczanie dotyczy bloków o jednakowych etykietach (lub bez etykiet), wbudowana dyrektywa flush

atomic

- Zapewnia niepodzielność operacji uaktualnienia specyfikowanej lokacji pamięci. Realizowane przez: (wykluczanie dostępu innych wątków do zmiennej i zapewnienie dostępu do pamięci (odczyt i zapis do pamięci)
- dotyczy wyrażeń:
  - x++, ++x, x--, --x
  - gdzie x jest zmienną chronioną
- lub postaci:
  - x *operator* = *wyrażenie*
    - *operator* = {+, *, -, /, &, ^, |, <<, >>}
    - wyznaczenie wartości *wyrażenie* nie jest atomowe !!
- więcej wariantów dyrektywy – możliwych wersji dostępów do zmiennej w OpenMP wersja 3.1

Dyrektywy synchronizacyjne (4)

ordered

- określa ograniczony blok kodu w ramach regionu równoległej pętli który będzie realizowany wg oryginalnej (sekwencyjnej) kolejności iteracji pętli,
- powoduje sekwencyjność przetwarzania oznaczonego bloku kodu, sprawdzanie zakończenia pracy dla wcześniejszych iteracji,
- pozwala aby przetwarzanie wątków poza blokiem biegło w pełni niezależnie - współbieżnie,
- Dyrektywa pojawia się w ramach bloków poprzedzonych dyrektywą *for* lub *parallel for* z opcją *ordered.*

Dyrektywy synchronizacyjne (5)

```
#pragma omp parallel shared(x,y) private (x_next, y_next)
{
    #pragma omp critical ( danex )
        x_next = pobierz_z_kolejki (x);

    work(x_next);

    #pragma omp critical ( daney )
        y_next = pobierz_z_kolejki (y);

    work(y_next);
}
```

dwie sekcje krytyczne do wzajemnego wykluczania dostępu do dwóch niezależnych kolejek *x* i *y,* zapobiegają pobieraniu z kolejki tego samego zadania przez wiele wątków.

Dyrektywy synchronizacyjne (6)

```
#pragma omp parallel
{
    #pragma omp single
            printf (''Początek pracy1 \n'');
    //synchronizcja single- nikt nie rozpoczął pracy
    work1( );
    #pragma omp single
            printf(''Kończenie pracy1 \n'');
   //synchronizcja single – niektóre wątki skończyły pracę
    #pragma omp single nowait
            printf(''Zakończona praca1 i poczatek pracy2 \n'');
    work2( );
}
```

Pierwszy (gotowy) z zespołu wątków generuje komunikat.
Bariera na końcu dyrektywy single.
Bariera usunięta za pomocą klauzuli *nowait.*

Dyrektywy synchronizacyjne (7)

```
#pragma omp parallel for shared (x, y, index, n)
   for (i=0; i<n; i++) {
   #pragma omp atomic
         x[index[i]] += work1(i);
         y[i] += work2(i);
   }
```

- przewaga dyrektywy *atomic* nad *critical* – więcej równoległości
- *Zakres atomic:* dotyczy *x,* nie dotyczy: *y* i wyznaczenia *work1(i), i* domyślnie prywatne

Dane w OpenMP (1)

Dane są:

- *private* - lokalne dane wątków – lokalne kopie zmiennych niewidoczne dla innych wątków
- *shared* - globalne dane

Współdzielone dane: zmienne widoczne w momencie osiągnięcia dyrektywy *parallel* lub *parallel for*

Prywatne dane:

- zadeklarowane dyrektywą *threadprivate*,
- zdefiniowane wewnątrz obszaru równoległego,
- zmienna sterująca pętli *for* z dyrektywą podziału pracy,
- umieszczone w klauzuli *private, firsprivate, lastprivate, reduction*

Dane w OpenMP (2)

threadprivate

– dyrektywa pozwalająca określić poza regionem równoległym zmienne jako prywatne dla każdego wątku, wartości z jednego regionu II są zachowane i przechodzą do kolejnego regionu II w odpowiednich wątkach

```
int counter = 0;
#pragma omp threadprivate(counter)
```

Klauzule sterujące współdzieleniem danych (1)

private (lista)

- klauzula dyrektywy zrównoleglającej lub współdzielącej pracę,
- pozwala na określenie zmiennych jako prywatnych dla każdego z wątków,
- dla każdego wątku tworzony jest nowy obiekt o typie określonym przez typ zmiennej,
- wartość początkowa obiektu każdego wątku jest zgodna z rezultatem działania konstruktora obiektu.

firstprivate (lista)

- w efekcie tej dyrektywy wartość początkowa lokalnego obiektu (określanego bloku i zadania) jest zgodna z wartością oryginalnego obiektu określoną przed początkiem bloku (w wątku nadrzędnym)

Klauzule sterujące współdzieleniem danych (2)

lastprivate (lista)

- dyrektywa ta tworzy według listy obiekty prywatne dla wątków
- wartość zmiennej z wątku przetwarzającego ostatnią iterację (nie ostatnią operację) pętli lub **section** – po ich zakończeniu - zostaje skopiowana do zmiennej oryginalnej - w wątku nadrzędnym.

shared (lista)

- klauzula powoduje współdzielenie miedzy wątkami wyspecyfikowanych zmiennych

Klauzule sterujące współdzieleniem danych (3)

- reduction(operacja:lista zmiennych)
  - klauzula powoduje realizację operacji redukcji (scalenie) w oparciu o podaną zmienną skalarną (współdzieloną
  - operacja redukcji ma postać typu :
    - x = x *op* expr np.: x = x – f( );
    - x *binop* = expr np.: x += f( )
    - expr – wartość wyznaczona z podanego wyrażenia

  Realizacja:
  - Powstaje prywatny obiekt dla każdego wątku (lub zadania każdego wątku OpenMP 3.1) każdej ze zmiennych z listy **reduction**. Każdy obiekt jest inicjowany w sposób zależny i adekwatny od operatora (np. dla min - max(type))
  - Na końcu regionu ze zdefiniowaną klauzulą **reduction,** wartość oryginalnego obiektu (współdzielony) jest aktualizowany do wyniku będącego połączeniem za pomocą podanego operatora jego wartości początkowej i ostatecznych wartości każdego z prywatnych obiektów
  - Wartość obiektu oryginalnego podlegającego redukcji jest nie zdeterminowana do momentu zakończenia bloku posiadającego klauzulę **reduction** (w przypadku klauzuli **nowait** konieczna bariera do zapewnienia poprawności wartości obiektu).

Klauzule sterujące współdzieleniem danych (4)

- reduction – przykład

```
#pragma omp parallel for reduction (+:a,y)
for (i=0; i<n; i++) {
    a += b[i];
    y = sum(y, c[i]);
        }
```

- operacje dodawania realizowane na prywatnych obiektach a, y wątków
- operator redukcji (+) ukryty w wywoływanej funkcji sum()

copyin(lista zmiennych)

- dotyczy zmiennych typu **threadprivate**
- dyrektywa powoduje zsynchronizowanie wartości zmiennej każdego z wątków do wartości oryginalnej zmiennej - w ramach wątku *master*
- realizacja na początku regionu równoległego

copyprivate(lista zmiennych)

- zsynchronizowanie wartości zmiennych prywatnych wątków do wartości obiektu jednego z wątków – realizującego blok **single**
- klauzula dostępna tylko w ramach dyrektywy **single**
- realizacja przy wyjściu z bloku single

Klauzule sterujące współdzieleniem danych (6)

- default(private) - sterowanie współdzieleniem zmiennych, których atrybuty współdzielenia są zdeterminowane nie wprost. Traktowane będą jako prywatne.
- default(none) – wymaga, aby wszystkie zmienne, do których następuje odwołanie w regionie współbieżnym, a które nie posiadają predefiniowanych atrybutów współdzielenia, uzyskały takie poprzez wylistowanie w ramach klauzul atrybutów współdzielenia danych.

Funkcje biblioteki czasu wykonania (1)
funkcje środowiska przetwarzania

- omp_set_num_threads (int liczba_wątków)
  - określa liczbę wątków powoływanych przy wejściu do regionu równoległego, posiada wyższy priorytet nad **OMP_NUM_THREADS (zm. środowiskowa)**
  - realizowana w obszarze kodu w którym funkcja:

    omp_in_parallel() zwraca 0
  - Inne sposoby określania liczby wątków:
    - **num_threads** – klauzula dyrektywy **parallel**
    - **OMP_NUM_THREADS –** zmienna środowiska
    - **omp_set_dynamic()** i **OMP_DYNAMIC** zezwalają na dynamiczną modyfikację liczby wątków
- Złożony algorytm określenia liczby wątków w regione II w 3.1. bazuje na ICV (internal control variable)

Funkcje biblioteki czasu wykonania (2)

- omp_get_num_threads
  - pozwala uzyskać liczbę wątków w zbiorze realizującym bieżący region współbieżny;
  - **omp_get_max_threads** zwraca (także poza regionem równoległym) maksymalną wartość zwracaną przez **omp_get_num_threads;**
- omp_get_thread_num
  - zwraca numer wątku (<0,omp_get_num_threads()-1>) w ramach zbioru realizującego region równoległy, wątkowi **master** zwraca 0;
- omp_get_num_procs
  - zwraca maksymalną liczbę procesorów, która może zostać przydzielona do programu;

Funkcje biblioteki czasu wykonania (3)

- omp_in_parallel()
  - zwraca wartość <> 0 jeśli wywołana w ramach regionu realizowanego współbieżnie;
- omp_set_nested (true/false) - włączenie/ wyłączenie zagnieżdżonej równoległości dla bieżącego wątku (zadania)
- omp_get_nested() – informacja o statusie zagnieżdżonej równoległości dla wątku(zadania)

  Parametr OpenMP służy do określenia poziomu zagnieżdżania równoległości w 3.1.

Funkcje biblioteki czasu wykonania (4) - funkcje zamków

Zamki powodują zawieszenie przetwarzania w przypadku odwołania się procesu do założonego zamka i pozwalają na wznowienie przetwarzania po otwarciu (usunięciu)zamka;

Typy:

- omp_lock_t – typ standardowy zamek
- omp_nest_lock_t – typ zagnieżdżony zamek

Realizacja:

1. system zapewnia dostęp do najbardziej aktualnego stanu zamka
2. inicjalizacja zamka / usuwanie zamka
   - omp_init_lock/omp_destroy_lock
   - omp_init_nest_lock/omp_destroy_nest_lock – tworzenie - ustawiana wartość parametru zagnieżdżenia zamka równa 0
3. omp_set_(nest_)lock - zamykanie
   - wątek jest zawieszany do momentu, gdy wskazany zamek zostanie otwarty przez proces, który go zamknął (typ standardowy),
   - gdy zamek został zamknięty wcześniej przez ten sam wątek (typ zagnieżdżony zamka) – następuje zwiększenie licznika zagnieżdżenia;

Funkcje biblioteki czasu wykonania (5) - funkcje zamków

4. Otwarcie zamka: *omp_unset_(nest_)lock*
   - parametrem jest zamek zamknięty przez ten sam wątek,
   - jest on otwierany (typ standardowy zamka) lub
   - licznik zagnieżdżenia jest zmniejszony i zamek jest zwolniony pod warunkiem, że licznik jest równy 0 (typ zagnieżdżony zamka)
5. test i zamknięcie zamka: *omp_test_(nest_)lock*
   - funkcja działa jak *omp_set_(nest_)lock* lecz nie wstrzymuje przetwarzania wątków w przypadku braku możliwości zamknięcie zamka (zwraca zero)
   - zwraca nową wartość licznika zagnieżdżenia

Zmienne środowiskowe dla OpenMP

- OMP_SCHEDULE
- OMP_NUM_THREADS
- OMP_DYNAMIC
- OMP_NESTED
- Ustawianie:
  - csh: setenv OMP_SCHEDULE ”dynamic”
  - ksh: export OMP_SCHEDULE=”guided,4”

Przykładowe błędy

```
//ZAMIAR: każdy wątek niech wykona raz pracę z parametrem = id
np = omp_get_num_threads();      /* błędny kod  */
#pragma omp parallel for private(i) schedule(static)
    for (i=0; i<np; i++)
          work(i);
-------------------------------------------------------------
#pragma omp parallel private(i)    /* kod poprawny */
{
    i = omp_get_thread_num( );
    work(i);
}
```

- barrier wewnątrz bloku critical lub single

Lastprivate, firsprivate - przkład

```
void example(float*a, float*b)
{
#pragma omp parallel for lastprivate(i)
for(i=0; i<k; i++ )a[i] = b[i];
-- i master’a równe k
#pragma omp parallel for firstprivate(i)
for(j=i; j<n; j++ )a[j] = 1.0;
}
```

Dalsze informacje :

OPENMP.ORG

Przykładowe zadanie zaliczeniowe (OMP, pamięć podręczna)

- Za pomocą odpowiedniej funkcji przydzielono na stałe 4 wątki na 4 procesory systemu równoległego (z prywatnymi pamięciami podręcznymi o długości linii pp 64 bajty). Obliczenia 4 wątków utworzonych dla kodu podanego poniżej korzystają ze zmiennych współdzielonych i prywatnych typu float (4 bajty).
-
  - Z ilu różnych obiektów zmiennych korzystają wszystkie wątki - proszę wymienić wszystkie obiekty? Które obiekty zmiennych są prywatne, a które są współdzielone?
  - Proszę wyjaśnić - Ile wynosi stosunek trafień do pp dla dostępów (odczyty i zapisy) do zmiennych współdzielonych, a ile do zmiennych prywatnych?
- int i,n;
- float a[n],Suma;
- ……
- #pragma omp parallel for reduction(Suma:+) schedule(static,1)
- for (i=0; i<n; i++)
- Suma+=a[i];