# Open MP wer. 2.5

Wykład PR część 3

Rafał Walkowiak
Instytut Informatyki Politechniki Poznańskiej
Jesień 2016

# OpenMP

*standard specyfikacji przetwarzania współbieżnego*

- uniwersalny (przenośny) model równoległości typu fork-join (rozgałęzienie - połączenie) dla architektur z pamięcią współdzieloną
  - przetwarzanie rozpoczyna się od jednego wątku przetwarzania
  - *regiony współbieżne* (parallel regions) wymagają powstania nowych wątków
  - wątki łączą się (bariera synchronizacyjna) przy końcu *regionu współbieżnego*
  - określona pamięć dostępna dla wszystkich wątków (współdzielona)
  - dla komputerów typu SMP (ang. Symmetrical Multi Processor ) - systemy z pamięcią współdzieloną
  - implementowany w większości platform obliczeniowych
  - dostarczany w postaci API (interfejsu dla programowania aplikacji)

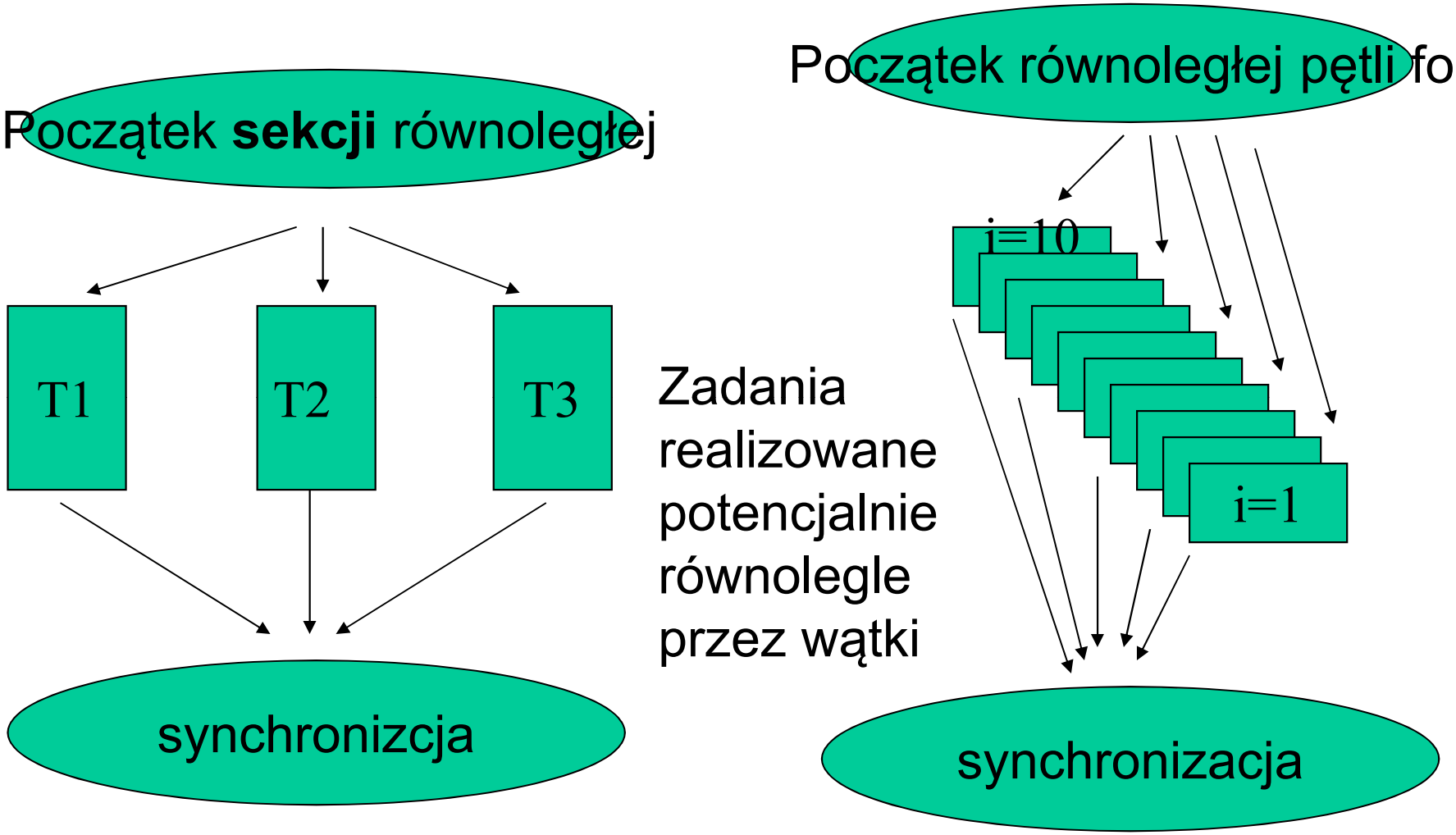
# Synchronizacja

- Synchronizacja niezbędna do:
  - Zapewnienia powiązań części programu równoległego,
  - zapobieżenia potencjalnie niebezpiecznym dostępom do pamięci np. nadpisanie wyniku, brak danej itp.
  - synchronizacji wybranych zdarzeń przetwarzania,
  - globalnej synchronizacji - bariera synchronizacyjna dla zbioru wątków.

# Poziomy równoległości w OpenMP

- wysoki poziom równoległości przetwarzania -
  - program podzielony na segmenty, które mogą być realizowane współbieżnie,
- 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



# 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 obszarów kodu podlegającym dyrektywom sterującym,
  - specyfikacja wprost – zastosowanie dyrektyw: barrier, critical, atomic, flush, ordered;

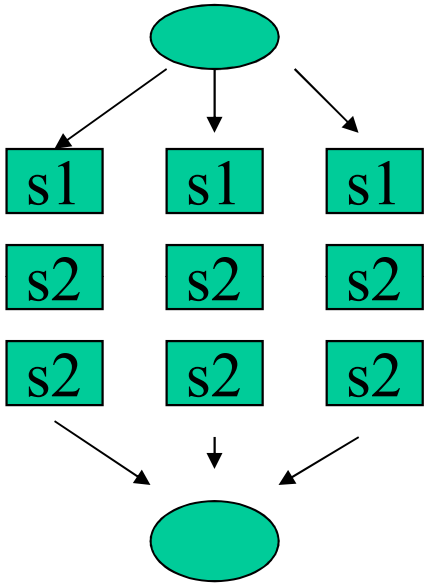
# Dyrektywy sterujące przetwarzaniem (1)

- parallel – podstawowa dyrektywa kodu równoległego
  - dyrektywa **tworząca** związany z nią zbiór 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 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 (oznaczanych **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 zakończeniu realizacji bloku 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**

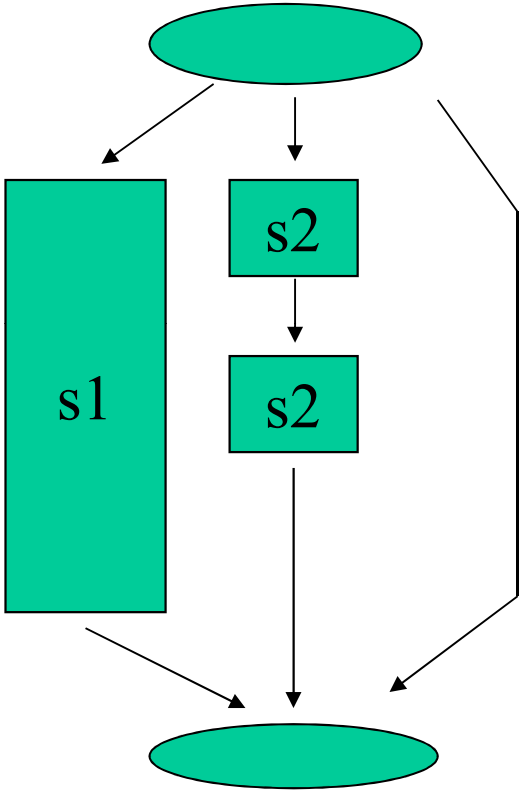
# Dyrektywy sterujące przetwarzaniem (3)




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

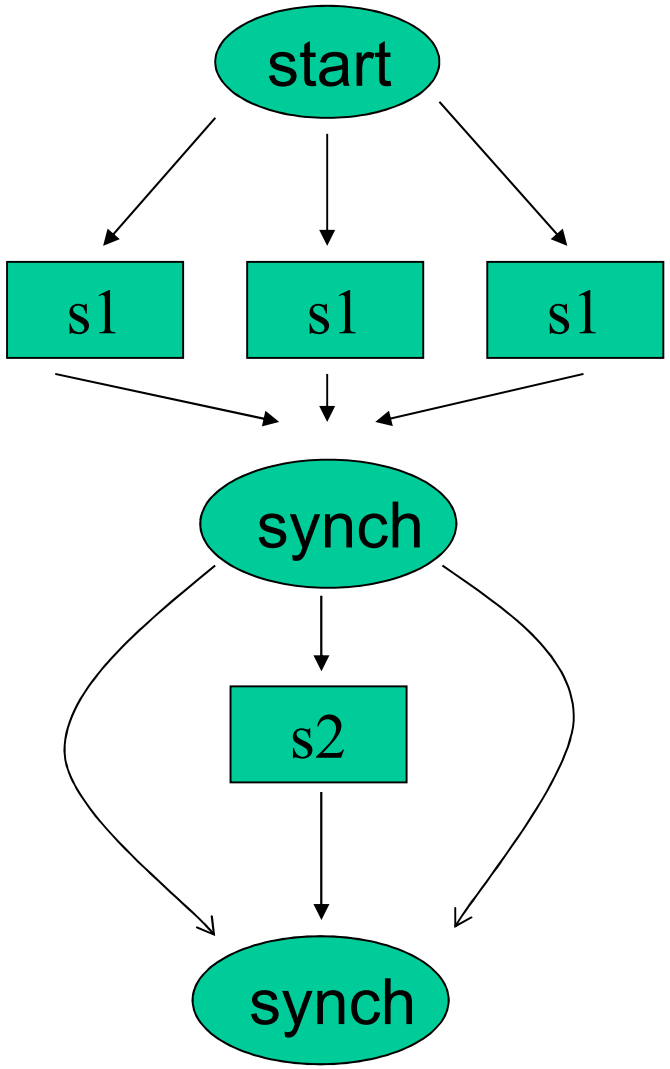
//zakładamy, że tworzone są 3 wątki

# 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
        #pragma omp single
                s2();
//synchronizacja
        }
```

# 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 pierwszej 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 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 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 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 (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), wbudowany flush

*atomic*

- Zapewnia niepodzielność operacji uaktualnienia specyfikowanej lokacji pamięci. Realizowane przez: (wykluczanie dostępu do zmiennej i zapewnienie dostępu do pamięci (nie korzystamy z wartości w rejestrze))
- dotyczy wyrażeń:
  - x++, ++x, x--, --x
  - 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 (3)

*flush*

- zapewnia spójny obraz pamięci w ramach wszystkich wątków: dotyczy zmiennych wyspecyfikowanych jako parametry (lub zmiennych współdzielonych widocznych w bieżącym kontekście);
- w efekcie: wszystkie wcześniejsze operacje tego wątku (realizującego flush) odwołujące się do podanych zmiennych się zakończyły, a następne jeszcze nie zaczęły;
- wymaga, aby kompilator spowodował zapis zmiennych przechowywanych w rejestrach do pamięci oraz wątek ponownie załadował wartości z pamięci do rejestrów (niekoniecznie wartości te będą równe  zapisanym przez siebie).
- bez parametrów - dotyczy wszystkich zmiennych współdzielonych;
- automatycznie realizowana w ramach *barrier, critical* we/wy, *ordered* we/wy, *parallel* wy, *for* wy, *sections* wy, *single* wy, atomic (w atomic dotyczy tylko zmiennej uaktualnianej)

# Dyrektywy synchronizacyjne (4)

*ordered*

- określa 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 niezależnie - współbieżnie,
- 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 ( xaxis )
        x_next = pobierz_z_kolejki (x);

    work(x_next);

    #pragma omp critical ( yaxis )
        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'');
    work1( );
    #pragma omp single
        printf(''Kończenie pracy1 \n'');
    #pragma omp single nowait
        printf(''Zakończona praca1 i poczatek pracy2 \n'');
    work2( );
}
```

Pierwszy z wątków docierający do dyrektywy *single* 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*
- *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: zmienne widoczne w momencie osiągnięcia dyrektywy *parallel* lub *parallel for*

Prywatne:

- zadeklarowane dyrektywą *threadprivate*,
- zdefiniowane wewnątrz obszaru równoległego,
- 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 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 (w wątku nadrzędnym)

# Klauzule sterujące współdzieleniem danych (2)

lastprivate (lista)

- dyrektywa ta tworzy według listy obiekty prywatne dla wątków i dodatkowo
- wartość zmiennej z wątku przetwarzającego ostatnią iterację pętli lub **section** – po ich zakończeniu- zostaje skopiowana do zmiennej oryginalnej - w wątku master.

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 (globalną)
  - operacja redukcji ma postać typu :
    - x = x *op* expr  np.: x = x – f( );
    - x *binop* = expr)  np.: x += f( )

  Realizacja:
  - Powstaje prywatna kopia dla każdego wątku (zadania każdego wątku 3.1) każdej ze zmiennych z listy **reduction**. Każda kopia jest inicjowana w sposób zależny i adekwatny od operatora (np. dla min - max(type))
  - Na końcu regionu ze zdefiniowaną klauzulą **reduction,** wartość oryginalnego obiektu jest aktualizowana do wyniku będącego połączeniem za pomocą podanego operatora jego wartości początkowej i ostatecznych wartości każdej z prywatnych kopii.
  - 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).

- reduction – przykład

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

- operator redukcji ukryty w wywoływanej funkcji sum()

## copyin(lista zmiennych)

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

## copyprivate(lista zmiennych)

- zsynchronizowanie wartości zmiennej prywatnej 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

## 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 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 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 otwarany (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

```
//każdy wątek niech wykona raz
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