

# Wprowadzenie do zrównoleglania aplikacji z wykorzystaniem standardu OpenMP

Tomasz Olas

`olas@icis.pcz.pl`

Instytut Informatyki Teoretycznej i Stosowanej  
Politechnika Częstochowska



# OpenMP

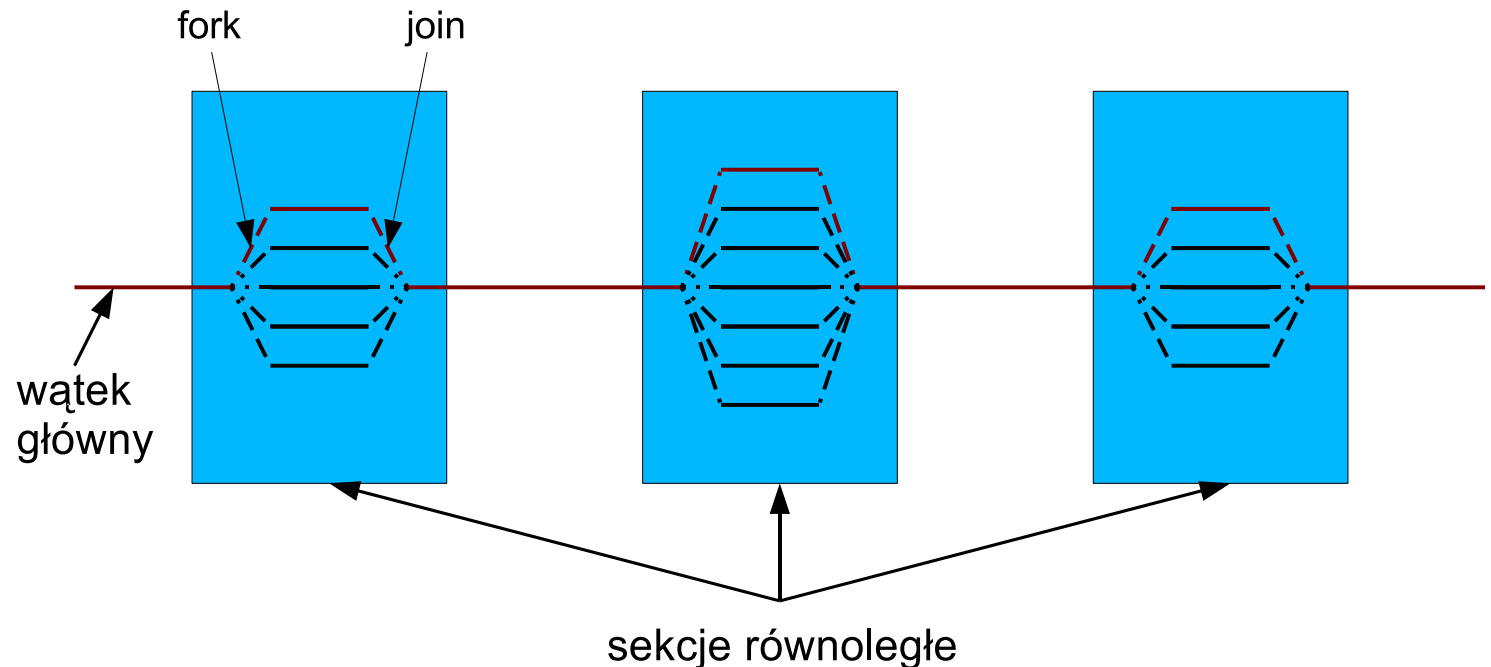
- OpenMP (*Open Multi-Processing*) jest standardem definiującym interfejs programowania aplikacji (API - Application Programming Interface) do tworzenia programów równoległych opierających się na modelu wielowątkowym dla systemów z pamięcią wspólną.
- Składa się z:
  - dyrektyw kompilatora,
  - funkcji bibliotecznych,
  - zmiennych środowiskowych.
- API OpenMP jest przeznaczony dla języków C/C++ i Fortran.
- Implementacje standardu OpenMP znajdują się na wielu platformach sprzętowych zawierających większość systemów UNIX'owych i Windows NT.

# Standard OpenMP

- W skład komitetu pracującego nad standardem OpenMP ARB (The OpenMP Architecture Review Board) wchodzi między innymi: Sun, NASA, Intel, Fujitsu, IBM, AMD, Cray, HP, SGI, The Portland Group, Inc., NEC, Microsoft.
- Standard OpenMP jest stale rozwijany - aktualną wersją standardu jest wersja 2.5 (draft 3.0).
- Strona domowa: [www.openmp.org](http://www.openmp.org)

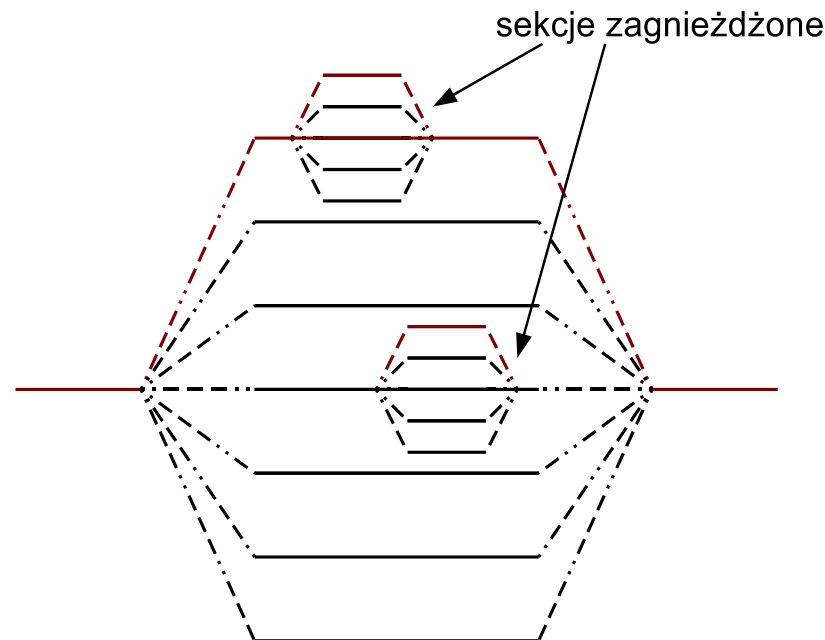
# Model programowania

- Zrównoleglanie kodu opiera się o model *fork-and-join*:
  - wątek główny tworzy dodatkowe wątki w momencie konieczności wykonywania obliczeń równoległych:



# Zrównoleglenie zagłębione

- Specyfikacja OpenMP przewiduje zrównoleglenie zagłębione (*nested parallelism*).
- Każdy wątek w grupie może utworzyć własną grupę wątków wykonujących się równolegle.
- Aby taki model przetwarzania był możliwy należy ustawić odpowiednią zmienną systemową (domyślnie nie jest ona włączona).



# Składnia OpenMP

- Większość konstrukcji w OpenMP jest dyrektywami preprocesora opartymi na dyrektywie **pragma** zdefiniowanej w C/C++.
- Ogólny format dyrektywy jest następujący:

```
#pragma omp konstrukcja [klauzula [klauzula]...]NL
```

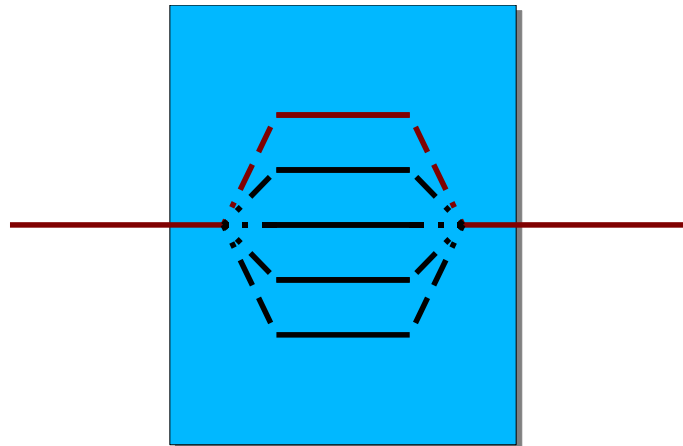
- Ponieważ podstawowym elementem składni są dyrektywy preprocesora, to program zrównoleglony przy wykorzystaniu OpenMP może być skompilowany kompilatorem nie wspierającym standardu OpenMP.

# Sekcja równoległa (I)

- Dyrektywa **omp parallel** określa obszar wykonywany równoległe przez wątki:

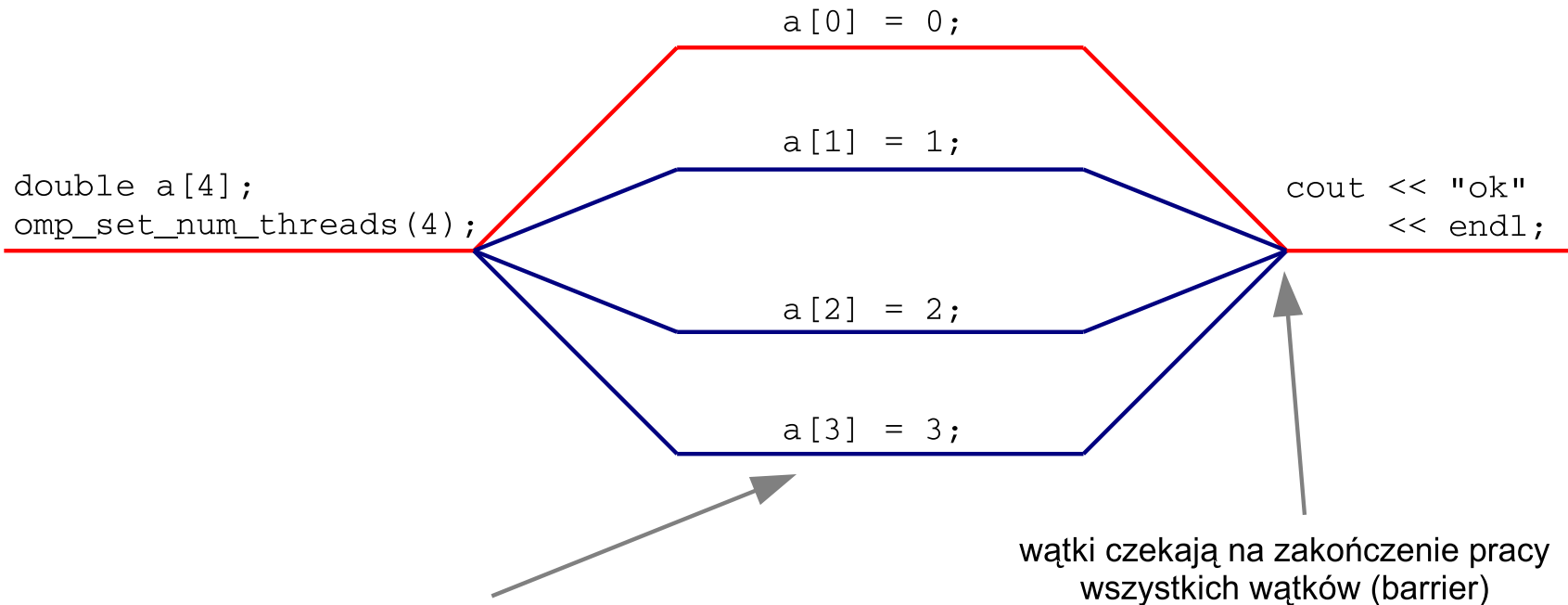
```
#pragma omp parallel [klauzula [klauzula]...]  
< C/C++ blok strukturalny >
```

- blok strukturalny jest blokiem instrukcji języka C/C++, który posiada jedno wejście i jedno wyjście.
- Liczba tworzonych wątków jest określona przez zmienną `OMP_NUM_THREADS`.



# Sekcja równoległa (II)

```
double a[4];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int id = omp_get_thread_num();  
    a[id] = id;  
}  
cout << "ok"  
      << endl;
```





# Komunikacja pomiędzy wątkami

- Każdy wątek w sekcji równoległej posiada zmienne prywatne i zmienne dzielone (wspólne dla wątków).
- OpenMP standardem programowania w oparciu o model z pamięcią wspólną - komunikacja pomiędzy wątkami odbywa się poprzez wspólne zmienne.
- Domyślnie każda zmienna widoczna podczas pracy równoległej jest traktowana jako wspólna dla wszystkich wątków.
- Zmienne lokalne wewnątrz bloku równoległego są prywatne.
- Zmienne statyczne zadeklarowane wewnątrz rozszerzenia dynamicznego są traktowane jako wspólne.
- Zaalokowana pamięć jest wspólna (choć wskaźnik do niej może być prywatny).

# Zakres widoczności zmiennych (I)

## ● Składnia:

```
#pragma omp parallel [Klauzule]  
<blok strukturalny języka C/C++>
```

## ● Klauzule:

```
private(list)  
shared(list)  
default(shared | none)  
firstprivate(list)
```

# Zakres widoczności zmiennych (II)

- **private(list)**  
Zmienne zadeklarowane wewnątrz klauzuli **private** są prywatne dla każdego wątku. Dla każdego wątku tworzona jest zmienna prywatna. Jej wartość jest nieokreślona przy wejściu do bloku równoległego, jak i po wyjściu z niego.
- **firstprivate(list)**  
Rozszerza klauzulę **private**. Zmienna, która jest tworzona jest kopią zmiennej globalnej.
- **shared(list)**  
Zmienne wymienione w klauzuli są wspólne dla wszystkich wątków.

# Zakres widoczności zmiennych (III)

## ● **default(shared | none)**

Klauzula default pozwala określić zakresy zmiennych

- **default(shared)** oznacza że każda z widocznych zmiennych będzie traktowana jako wspólna, poza zadeklarowanymi przy pomocy threadprivate i stałymi.
- **default(none)**  
Wymaga, aby każda z aktualnie widocznych w rozszerzeniu leksykalnym zmiennych była jawnie zadeklarowana w którejś z klauzul zakresu widoczności danych.

# Zakres widoczności zmiennych (IV)

## ● Przykład:

```
double a = 1;
```

```
double b = 2;
```

```
double c = 3;
```

```
#pragma omp parallel private(b) firstprivate(c)
```

```
{
```

```
    (1)
```

```
}
```

```
    (2)
```

- wewnątrz obszaru równoległego (1):
  - a jest zmienną wspólną dla wszystkich wątków (jest równa 1),
  - b i c są zmiennymi lokalnymi dla wątków (wewnątrz wątków wartość początkowa zmiennej b jest niezdefiniowana, natomiast zmiennej c jest równa 3).
- poza obszarem równoległym (2):
  - wartości zmiennych b i c są niezdefiniowane.

# Dyrektywa threadprivate

- Składnia:

```
#pragma omp threadprivate (list)
```

- Dyrektywa threadprivate oznacza, że wszystkie zmienne podane jako parametry na liście będą prywatne dla wątków w całej przestrzeni programu.

# Zrównoleglanie pętli

- Najczęściej stosowanym sposobem zrównoleglania programów przy użyciu standardu OpenMP jest zrównoleglanie pętli:
  - krok 1: określenie najbardziej czasochłonnych pętli w programie,
  - krok 2: paralelizacja pętli poprzez przetwarzanie jej przez wiele wątków.
- Wykorzystuje się do tego celu dyrektywę **omp for**:

```
#pragma omp konstrukcja [klauzula [klauzula]...]
```

< pętla, która ma być wykonywana w sposób równoległy >

## program sekwencyjny

```
void main()
{
    double res[1000];

    for (int i = 0; i < 1000; i++)
        oblicz(res[i]);
}
```

## program równoległy

```
void main()
{
    double res[1000];

    #pragma omp parallel for
    for (int i = 0; i < 1000; i++)
        oblicz(res[i]);
}
```

# Zrównoleglanie pętli - przykład

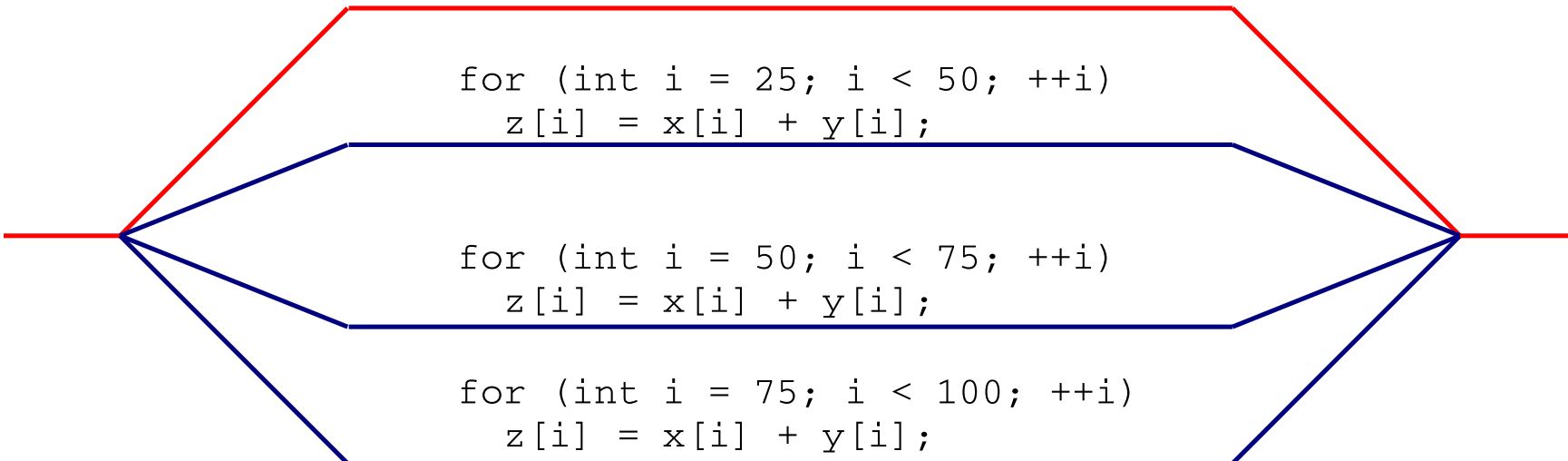
```
double x[100];  
double y[100];  
double z[100];  
  
#pragma omp parallel  
#pragma omp for  
for (int i = 0; i < 100; ++i)  
    z[i] = x[i] + y[i];
```

```
for (int i = 0; i < 25; ++i)  
    z[i] = x[i] + y[i];
```

```
for (int i = 25; i < 50; ++i)  
    z[i] = x[i] + y[i];
```

```
for (int i = 50; i < 75; ++i)  
    z[i] = x[i] + y[i];
```

```
for (int i = 75; i < 100; ++i)  
    z[i] = x[i] + y[i];
```





# omp for

## ● Składnia:

```
#pragma omp for [Klauzule]  
<pętla języka C/C++, która będzie wykonywana równolegle>
```

## ● Klauzule:

```
private(list)  
firstprivate(list)  
lastprivate(list)  
reduction(operator: list)  
schedule(kind[, chunk])  
ordered  
nowait
```

# Klauzula lastprivate

- Klauzula **lastprivate** powoduje, że zmienna globalna, która jest zmienną prywatną podczas wykonywania pętli po jej zakończeniu przyjmuje wartość dla ostatniej iteracji pętli

```
int a, v = 0;
```

```
#pragma omp parallel
#pragma omp for lastprivate(a)
for (int i = 0; i < 10; ++i)
{
    a = i * 100;
    v += a;
}
```

```
std::cout << a << std::endl;
```

# Sekcja krytyczna

- OpenMP udostępnia mechanizm sekcji krytycznej.
- Fragment kodu, który ma być wykonywany w danym czasie tylko przez jeden wątek należy umieścić w bloku po dyrektywie `OMP CRITICAL`.
- Szczególnym przypadkiem sekcji krytycznej jest operacja atomowa (dyrektywa `OMP ATOMIC`). Polega ona na modyfikacji tylko jednej komórki pamięci przez jeden wątek w danym czasie, np. inkrementacja, dekrementacja.

# Sekcja krytyczna - przykład

```
#include <iostream>

int main()
{
    double a[100];
    double sum = 0.0;

    #pragma omp parallel
    {
        double lsum = 0.0;

        #pragma omp for
        for (int i = 0; i < 100; ++i)
            lsum += a[i];

        #pragma omp critical
            sum += lsum;
    }

    std::cout << "suma: " << sum << std::endl;
```

# Klauzula reduction I

- **reduction(operator: list)**
- operator: +, \*, -, &, ^, |, &&, ||
- Na początku bloku równoległego tworzone są prywatne kopie zmiennych z listy (tak jak w przypadku klauzuli **private**). Następnie są one inicjowane wartościami zależnymi od użytego operatora. Na końcu obszaru równoległego oryginalne zmienne są aktualizowane wartością stanowiącą połączenie wartości przed wykonaniem konstrukcji z końcowym rezultatem operacji redukcji wykonanej na prywatnych kopiach.

# Klauzula reduction II

- Wartości początkowe dla poszczególnych operacji:

operator	wartość początkowa
+	0
	1
-	0
&	$\sim 0$
	0
^	0
& &	1
	0

# Klauzula reduction - przykład

```
#include <iostream>

int main()
{
    double sum = 0.0;
    double tab[100];
    ...

    #pragma omp parallel
    #pragma omp for reduction(+: sum)
    for (int i = 0; i < 100; i++)
        sum += tab[i];

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

# omp ordered

- Składnia:

```
#pragma omp ordered  
<blok strukturalny języka C/C++>
```

- Dyrektywa **omp ordered** może wystąpić jedynie w dynamicznym rozszerzeniu dyrektyw **for** lub **parallel for**.
- Blok strukturalny występujący po dyrektywie **omp ordered** jest wykonywany w danej chwili tylko przez jeden wątek w porządku takim jak w pętli sekwencyjnej.



# omp for - przydział zadań

- Pętle można podzielić na dwa rodzaje:
  - w każdej iteracji pętli wykonuje się tyle samo operacji, w związku z tym każda iteracja wykonuje się w takim samym czasie,
  - czas wykonania poszczególnych iteracji różni się w zależności od pewnych czynników od innych.
- OpenMP zawiera mechanizmy do automatycznej optymalizacji przydziału zadań do wątków w dyrektywie **omp for**.
- O sposobie przydziału zadań decyduje programista, a tzw. sheduler automatycznie przydziela zadania do wątków.
- Możliwe są cztery rodzaje przydzielania zadań do wątków - **static**, **dynamic**, **guided**, **runtime**.

# omp for - schedule static

- Gdy jest określona klauzula **schedule (static, chunk)** iteracje są przydzielane do poszczególnych wątków w blokach o rozmiarze **chunk**. Bloki iteracji przydzielane są w sposób cykliczny.
- **chunk** musi być liczbą całkowitą.
- Kiedy nie został określony rozmiar bloku (**chunk**), to jest on równy:  
$$(\text{number\_of\_iterations} + \text{omp\_num\_threads}() - 1) / \text{omp\_num\_threads}()$$

# omp for - schedule dynamic

- Jeżeli jest wykorzystywana klauzula **schedule (dynamic, chunk)**, to każdy wątek otrzymuje do przetworzenia jeden blok iteracji o rozmiarze **chunk**. Gdy wątek skończy, to otrzymuje do przetworzenia następny blok iteracji.
- **chunk** musi być liczbą całkowitą.
- Kiedy nie został określony rozmiar bloku (**chunk**), to jest on równy 1.

# omp for - schedule guided

- Iteracje są dzielone na części o wielkości malejącej wykładniczo, do wielkości równej kwant (domyślnie równej 1). Następnie części są przydzielane dynamicznie do wątków. Wielkość kawałka początkowego oraz stopień zanikania są zależne od implementacji.
- **chunk** musi być liczbą całkowitą.

# omp for - schedule runtime

- Jeżeli jest wykorzystywana klauzula **schedule (dynamic, chunk)**, to decyzja o sposobie przydziału iteracji do wątków zostaje odroczone aż do momentu wykonania pętli.
- Typ przydziału zadań oraz rozmiar bloku iteracji (**chunk**) mogą być zmieniane w trakcie wykonywania programu poprzez zmienną **OMP\_SCHEDULE**.
- Jeżeli zmienna ta nie została ustawiona, to przyjmuje się sposób przydziału - **schedule(static)**.

# Serializacja w obszarze równoległym

- W przypadku, gdy wymagane jest wykonanie pewnych fragmentów kodu programu w sekcji równoległej tylko przez jeden wątek standard OpenMP udostępnia dyrektywy:
  - **omp master**
  - **omp single**

# omp master

- Składnia:

```
#pragma omp master  
<blok strukturalny języka C/C++>
```

- Blok strukturalny występujący po dyrektywie **omp master** jest wykonywany tylko przez wątek główny. Pozostałe wątki omijają ten fragment i wykonują pozostały kod sekcji równoległej.
- Wątki nie są blokowane przed i po sekcji wykonywanej przez wątek główny.

# omp single

- Składnia:

```
#pragma omp single [Klauzule]  
<blok strukturalny języka C/C++>
```

- Klauzule:

```
private(list)  
firstprivate(list)  
nowait
```

- Kod w bloku po dyrektywie **omp single** jest wykonywany w sekcji równoległej tylko przez jeden wątek (nie jest określone przez który).
- Przy końcu bloku **omp single** wątki są blokowane.



# Serializacja w obszarze równoległym

```
#pragma omp parallel for
for (int i = 0; i < 1000; ++i)
{
    # pragma omp single
        starting_operation();

    operation(i);

    # pragma omp critical
        synchronized_operation(i);

    # pragma omp master
        master_operation(i);
}
```

# Punkty synchronizacji wątków (I)

- Ze względu na operowanie wątków na wspólnych danych do poprawnego działania programu często niezbędne są mechanizmy synchronizacji.
- Każdy wątek, który dojdzie do punktu synchronizacji czeka na pozostałe wątki.
- W OpenMP można wyróżnić dwa rodzaje synchronizacji:
  - synchronizacja domyślna,
  - synchronizacja jawna.
- W chwili, gdy wszystkie wątki dojdą do punktu synchronizacji wykonują dalszą część programu lub następuje ich zniszczenie.

# Punkty synchronizacji wątków (II)

- Synchronizacja domyślna występuje na końcu bloku kodu dyrektyw:
  - `omp parallel,`
  - `omp for,`
  - `omp sections,`
  - `omp single,`
- błąd, że została użyta klauzula `nowait`, która nakazuje brak synchronizacji wątków.
- Programista w sposób jawny może określić miejsce synchronizacji wątków poprzez wykorzystanie dyrektywy `omp barrier`

# omp sections

## ● Konstrukcja sections:

```
#pragma omp parallel sections [Klauzule]
{
    [#pragma omp section]
        blok strukturalny
    [#pragma omp section]
        blok strukturalny
    ...
}
```

## ● Przykład:

```
#pragma omp parallel
#pragma omp sections
{
    # pragma omp section
        zad1();
    # pragma omp section
        zad2();
    # pragma omp section
        zad3();
}
```

# Konstrukcje łączone I

- Konstrukcje łączone są skrótami konstrukcji **parallel** i konstrukcji pracy dzielonej zawierającymi jedynie obszar pracy dzielonej.

- Konstrukcja **parallel for**:

```
#pragma omp parallel for [Klauzule]
```

```
<pętla języka C/C++, która będzie wykonywana równolegle>
```

Dyrektywa **parallel for** jest skrótem dla konstrukcji **parallel** zawierającej pojedynczą dyrektywę **for**. Jest równoważna dyrektywie **parallel** z bezpośrednio po niej następującą dyrektywą **for**.

# Konstrukcje łączone II

## ● Konstrukcja parallel sections:

```
#pragma omp parallel sections [Klauzule]
{
    [#pragma omp section]
        blok strukturalny
    [#pragma omp section]
        blok strukturalny
    ...
}
```

Dyrektywa **parallel sections** jest skrótem dla konstrukcji **parallel** zawierającej pojedynczą dyrektywę **sections**. Jest równoważna dyrektywie **parallel** z bezpośrednio po niej następującą dyrektywą **sections**.

## ● Klauzule są takie same, za wyjątkiem klauzuli **nowait**.

# Zmienne środowiskowe

- **OMP\_SCHEDULE** - ustawia rodzaj i ewentualnie porcje dla parametru **runtime** klauzuli **schedule**
- **OMP\_NUM\_THREADS** - ustawia ilość wątków wykorzystywaną podczas wykonywania programu.
- **OMP\_DYNAMIC** - ustawia lub blokuje dynamiczne przydzielanie wątków.
- **OMP\_NESTED** - ustawia lub blokuje zagnieżdżanie równoległości.

# Funkcje OpenMP I

- `omp_set_num_threads` - Ustawia ilość wątków używanych podczas wykonywania obszaru równoległego. Działanie jest zależne od tego, czy umożliwiające jest dynamiczne przydzielanie wątków. Jeśli nie, to ustawiona wartość oznacza ilość wątków tworzoną przy wejściu w każdy obszar równoległy (także zagnieżdżony). W przeciwnym wypadku wartość oznacza maksymalną ilość wątków która może zostać użyta.
- `omp_get_num_threads` - Funkcja zwraca aktualną ilość wątków w grupie wykonujących obszar równoległy z którego wywołano funkcję.



# Funkcje OpenMP II

- `omp_get_max_threads` - Zwraca maksymalną wartość, jaka może zostać zwrócona po wywołaniu funkcji `omp_get_num_threads`.
- `omp_get_thread_num` - Zwraca numer wątku w grupie (wątek główny ma numer 0).
- `omp_get_num_procs` - Zwraca maksymalną liczbę procesorów jaka może być przeznaczona na wykonywanie programu.
- `omp_in_parallel` - Zwraca wartość niezerową jeśli została wywołana z dynamicznego rozszerzenia obszaru równoległego wykonywanego równolegle. W przeciwnym wypadku zwraca 0.

# Funkcje OpenMP III

- `omp_set_dynamic` - Umożliwia lub zabrania dynamicznego przydzielania wątków do wykonywania obszarów równoległych. Jeżeli zezwolimy na dynamiczne przydzielanie, to ilość wątków używanych do wykonywania obszarów równoległych może zmieniać się automatycznie, tak, aby jak najlepiej wykorzystane zostały zasoby systemowe. W szczególności ilość wątków podana przez użytkownika jest ilością maksymalną. Ustawienie domyślne jest zależne od implementacji.
- `omp_get_dynamic` - Zwraca wartość niezerową, gdy ustawione jest dynamiczne przydzielanie wątków, w przeciwnym przypadku 0.

# Funkcje OpenMP IV

- `omp_set_nested` - Włącza lub wyłącza zagnieżdżanie równoległości. Jeżeli nie ma zgody na zagnieżdżanie, to zagnieżdżające się obszary równoległe są wykonywane sekwencyjnie, w przeciwnym wypadku zagnieżdżające się obszary równoległe mogą tworzyć dodatkowe grupy wątków.
- `omp_get_nested` - Zwraca wartość niezerową jeśli zagnieżdżanie jest włączone, a 0 w przeciwnym wypadku.

# Funkcje OpenMP - przykład

- W celu zmiany liczby wątków jakie będą wykorzystywane w sekcji równoległej najpierw należy wyłączyć tryb dynamicznego przydziału wątków do wykonania, a następnie ustawić liczbę wątków jaka ma być tworzona:

```
#include <omp.h>
```

```
void main()
```

```
{
```

```
    omp_set_dynamic(0);
```

```
    omp_set_num_threads(10);
```

```
#pragma omp parallel
```

```
{
```

```
    ...
```

```
}
```

```
}
```

# Funkcje lock

- Do synchronizacji wątków można w OpenMP wykorzystać również niskopoziomowe funkcje zarządzające ryglami (ang. lock):
  - `omp_init_lock()`,
  - `omp_set_lock()`,
  - `omp_unset_lock()`,
  - `omp_test_lock()`.

# Funkcje lock - przykład

```
omp_lock_t lock;  
omp_init_lock(&lock);  
  
#pragma omp parallel  
{  
    omp_set_lock(&lock);  
  
    sekcja krytyczna  
  
    omp_unset_lock(&lock);  
}
```

# Kompilacja warunkowa

- Jedną z cech standardu OpenMP jest możliwość przeprowadzenia prawidłowej kompilacji kodu zrównoleglonego przy wykorzystaniu standardu OpenMP przez kompilator nie wspierający standardu OpenMP. W takim przypadku wygenerowana zostanie wersja sekwencyjna programu.
- Kompilatory OpenMP rozpoznają makrodefinicję `_OPENMP`, co pozwala na przeprowadzenie kompilacji warunkowej:

```
#ifdef _OPENMP  
...  
#endif
```

Makrodefinicja `_OPENMP` nie może być poddawana operacjom preprocesora `#define` i `#undef`.