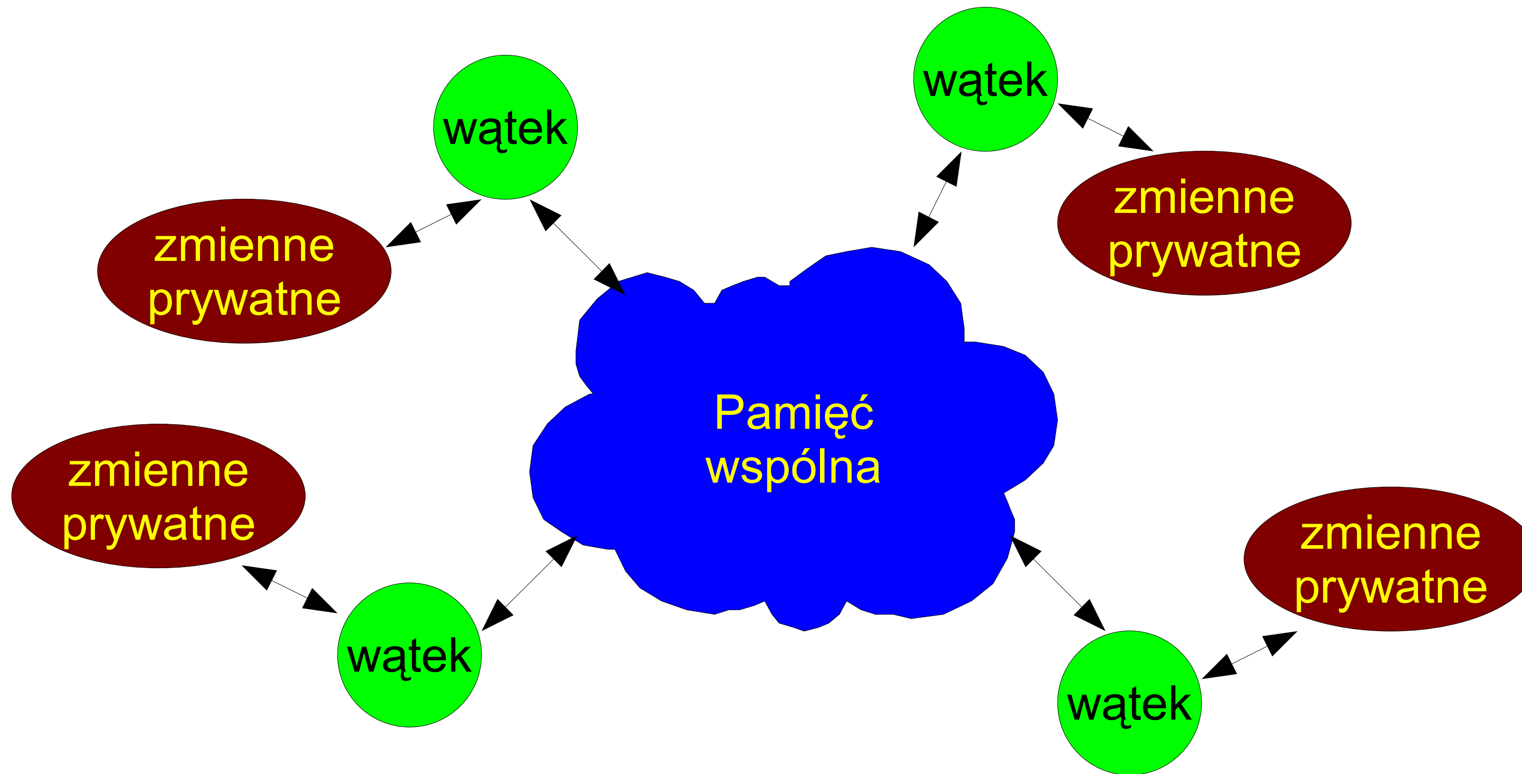
Programowanie maszyn z pamięcią wspólną w standardzie OpenMP.

OpenMP

- Standard rozwinięty i zdefiniowany w latach 90 przez grupę specjalistów z przemysłu.
- Strona www: **www.openmp.org**
- Składa się ze zbioru dyrektyw dla kompilatora oraz z niewielkiego zbioru funkcji bibliotecznych. (+ kilka zmiennych środowiskowych).
- Dyrektywy i funkcje biblioteczne zdefiniowane są dla języków C/C++ oraz Fortran (bardzo ważny język w obliczeniach naukowo inżynierskich).
- Wymaga wsparcia kompilatora akceptującego standard. Z kompilatorów obsługujących OpenMP wspominamy:
  - gcc w wersji 4.2 oraz 4.3 (standardowy kompilator open source na Linuksie). Do pobrania i skompilowania ze strony **gcc.gnu.org**.
  - icc (Intel C Compiller) firmy Intel. Dla osób nie używających go do pracy zarobkowej dostępny za darmo (w wersji Linuksowej). Generuje bardzo dobry kod na procesorach Intela, znacznie szybszy od kodu generowanego na przez gcc.
  - Sun Studio 12. Wersja Linuksowa dostępna za darmo. Generuje bardzo dobry kod dla procesorów 64-bitowych firmy AMD (Athlon 64, Opteron).

Model programowania OpenMP



- Aplikacja składa się z wielu wątków mających dostęp do pamięci współdzielonej.
- Dane mogą być ***współdzielone*** (ang. shared) lub ***prywatne*** (ang. private).
- Dostęp do danych prywatnych ma dostęp jedynie ten wątek, który je posiada.
- Dostęp do danych współdzielonych mają wszystkie wątki.

Postać dyrektyw OpenMP

- Dla języków C/C++ dyrektywy OpenMP mają postać dyrektyw #pragma preprocesora.
  - Dyrektywa #pragma preprocesora służy do przekazywania poleceń zależnych od implementacji. Dyrektywa #pragma nierozpoznawana przez kompilator powinna być ignorowana (być może wraz z ostrzeżeniem).
- Dyrektywy OpenMP mają zawsze postać:

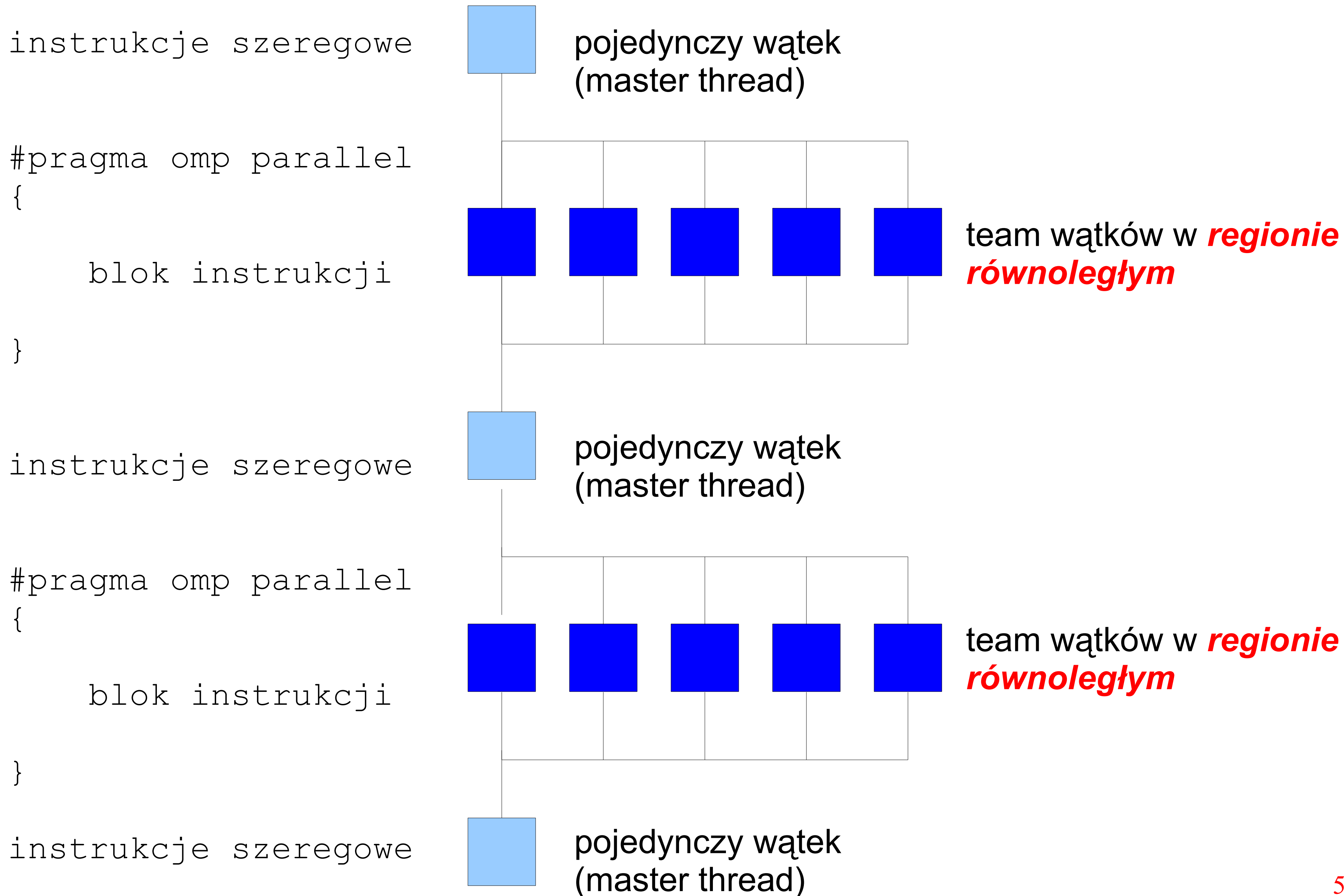
  #pragma omp nazwa_dyrektywy parametry

  gdzie omp jest słowem kluczowym OpenMP.
- Niektóre dyrektywy wymagają, aby występował po nich blok instrukcji (lub pojedyncza instrukcja).
  - Blok instrukcji w języku C/C++ ma postać

```
{
  instr1;
  instr2;
  instr3;
}
```

- Dyrektywa wraz z blokiem nazywana jest ***konstruktem***.

Model wykonania OpenMP



Dyrektywa parallel

- Program OpenMP wykonuje się sekwencyjnie w jednym wątku, do momentu napotkania dyrektywy parallel. Ma ona postać:

```
#pragma omp parallel [dodatkowe klauzule]
/* blok instrukcji będący regionem równoległym */
```

- Dyrektywa parallel tworzy team wątków. Każdy wątek z teamu wykonuje blok instrukcji. Wątek który napotkał dyrektywę parallel staje się wątkiem głównym (ang. master) teamu i otrzymuje identyfikator 0.
- Jedna z możliwych dodatkowych klauzul ma postać `if(wyrażenie)`. Jej użycie oznacza zrównoleglenie warunkowe: team wątków jest tworzony, wtedy i tylko wtedy gdy wyrażenie ma wartość różną od zera.
  - W przeciwnym wypadku blok instrukcji wykonywany jest przez jeden wątek (wątek główny).
- Na zakończenie kodu w dyrektywie parallel jest wykonywana operacja barriery. Każdy wątek czeka na zakończenie pracy w regionie równoległym przez pozostałe wątki.
  - Operacja ta zabezpiecza przed np. rozpoczęciem kolejnego regionu równoległego gdy aktualny nie został zakończony.
- Po wyjściu z regionu równoległego pracę kontynuuje wyłącznie wątek główny.

OpenMP - pierwszy program

```
// Plik nagłówkowy funkcji bibliotecznych
#include <omp.h>
#include <stdio.h>
int main()
{
   // Ustaw liczbę wątków na 15
   omp_set_num_threads(15);
   // Wypisze się jeden raz
   printf("Hello World! Kod szeregowy\n");
   // Poniższy blok wykona się w 15 wątkach
   #pragma omp parallel
   {
       printf("Hello, jestem wątkiem nr %d\n",omp_get_thread_num());
   }
}
```

- Komunikat wypisze się 15 razy, kolejność wątków nie jest zdefiniowana.

Jak określić liczbę wątków w teamie

- Można to zrobić na trzy sposoby.
- Po pierwsze można nadać wartość zmiennej środowiskowej OMP_NUM_THREADS. Z linii poleceń shella wpisujemy:

```
setenv OMP_NUM_THREADS 15 [dla shelli csh, tcsh]

export OMP_NUM_THREADS=15 [dla shelli sh, bash,ksh]
```

- Po drugie można skorzystać z funkcji bibliotecznej OpenMP:

  ```
  void omp_set_num_threads(int numthreads).
  ```

  Funkcję tą można wywoływać wyłącznie na zewnątrz bloku #pragma omp parallel.
- Po trzecie można skorzystać z dodatkowej klauzuli dyrektywy #pragma omp parallel:

```
#pragma omp parallel num_threads(15)
```

Inne funkcje biblioteczne OpenMP

- `int omp_get_num_threads()` zwraca liczbę wątku w teamie.
- `int omp_get_thread_num()` zwraca identyfikator wątku jest to liczba z zakresu 0,..,liczba_wątków-1.
- `int omp_get_max_threads()` zwraca maksymalną możliwą liczbę wątków w teamie.
- `int omp_in_parallel()` - zwraca jeden jeżeli jesteśmy wewnątrz regionu równoległego; przydatne w tzw. dyrektywach sierocych.
- `int omp_get_num_procs()` - zwraca liczbę dostępnych procesorów.
- `double omp_get_wtime()` - zwraca liczbę sekund jakie minęły od pewnego punktu w przeszłości.
- `double omp_get_wtick()` - zwraca liczbę sekund pomiędzy dwoma "tyknięciami" zegara.

Dynamiczne określanie liczby wątków

- Generalnie liczba wątków powinna być taka, że obciążenie systemu wątkami będzie nie większe niż liczba procesorów.
  - W przeciwnym wypadku może dojść do spadku wydajności spowodowanego niezrównoważonym obciążeniem, przełączaniem kontekstu, gorszym wykorzystaniem pamięci cache ...
- Jako pierwsze przybliżenie można przyjąć liczba wątków==liczba procesorów (wynik funkcji `omp_get_num_procs()`)
- Problem powstaje gdy system jest obciążony przez inne procesy (np. demony systemowe, inne aplikacje).
- Wyjście: dynamiczne dopasowywanie liczby wątków, włączane/wyłączane funkcją `void omp_set_dynamic(int val)`.
- Dynamiczne dopasowanie działa w ten sposób, że liczba wątków w regionie równoległym jest dostosowana do obciążenia systemu, tak aby zapewnić najlepsze wykorzystanie zasobów (procesorów).
- Ale uwaga dynamiczne dopasowanie liczby wątków może być niezaimplementowane.
- Można je włączyć również ustawiając zmienną środowiskową OMP_DYNAMIC na 1

Zachowanie się zmiennych w bloku parallel

- Dotyczy zmiennych zadeklarowanych ***przed blokiem parallel***.
- Zmienne mogą być albo współdzielone (ang. shared) albo prywatne (ang. private). Klauzula `shared(lista_zmiennych)` dyrektywy `parallel` deklaruje zmienne wymienione na liście jako współdzielone. Klauzula `private(lista_zmiennych)` dyrektywy `parallel` deklaruje zmienne wymienione na liście jako prywatne.
- Istnieje jedna kopia zmiennej współdzielonej do której dostęp mają wszystkie wątki.
- W przypadku zmiennej prywatnej każdy wątek ma swoją własną kopię zmiennej niedostępną innym wątkom. Zmiany wartości zmiennej prywatnej nie mają wpływu na wartości „widziane" przez inne wątki.
- Zmienne prywatne po wejściu do dyrektywy `parallel` mają wartości nieokreślone. Podobnie po wyjściu z dyrektywy. Jeżeli chcemy je zainicjalizować wartością oryginalnej zmiennej przed wejściem w dyrektywę `parallel`, to należy użyć klauzuli `firstprivate(lista_zmiennych)`.
- Domyślny typ zmiennych jest określony klauzulą `default(typ)`, gdzie typ może przyjąć jedną z wartości: `shared`, `private`, `none`. Wartość `none` oznacza, że musimy podać typ każdej ze zmiennych występującej w bloku `parallel`. C/C++ nie wspiera `default(private)`.

Typy zmiennych - przykład

```
double x=1.0;
int i=2;
float z=3.0f;
#pragma omp parallel default(none) shared(x) private(i) firstprivate(z)
{
   // default (none) oznacza, że w bloku możemy się odwołać wyłącznie
   // do zmiennych wymienionych w trzech klauzulach tzn. x,i,z.

   // każdy wątek ma prywatną kopię zmiennej i, a jej wartość jest
   // nieokreślona.

   // każdy wątek ma prywatną kopię zmiennej z, o wartości
   // początkowej 3.

   // wszystkie wątki operują na jednej kopii zmiennej x.
}
```

- Gdyby wewnątrz bloku parallel miało miejsce wywołanie funkcji to wszystkie zmienne tej lokalne funkcji będą zmiennymi prywatnymi a zmienne statyczne współdzielonymi..

Przypomnienie: POSIX threads utworzenie i dołączenie wątku

```
void *thread(void *param) {
// tu kod wątku


// możemy przekazać wynik
   return NULL
}

int main() {
    pthread_t id;
    // Parametr przekazywany wątkowi
    void *param=NULL;
    pthread_create(&id,NULL,&thread,param);
    // Funkcja thread w odrębnym wątku
    // współbieżnie z main
    // id przechowuje identyfikator wątku

    void *result;
    // Czekaj na zakończenie wątku
    pthread_join(id,&result);
    // Wynik w result,zamiast &result można
    // przekazać NULL
}
```

- Funkcja pthread_create tworzy nowy wątek. Rozpoczyna on pracę od funkcji, której adres przekazano jako trzeci argument.
- Funkcja pthread_join usypia wywołujący ją wątek do momentu, kiedy wątek o identyfikatorze przekazanym jako pierwszy argument zakończy pracę.
- Zakończenie pracy wątku – powrót z funkcji która go rozpoczyna.

Translacja wykonana przez kompilator (Grama i wsp.)

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
    // serial segment

    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);

    // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;
    // parallel segment
}
```

Code inserted by the OpenMP compiler

Zmienna prywatna

Corresponding Pthreads translation

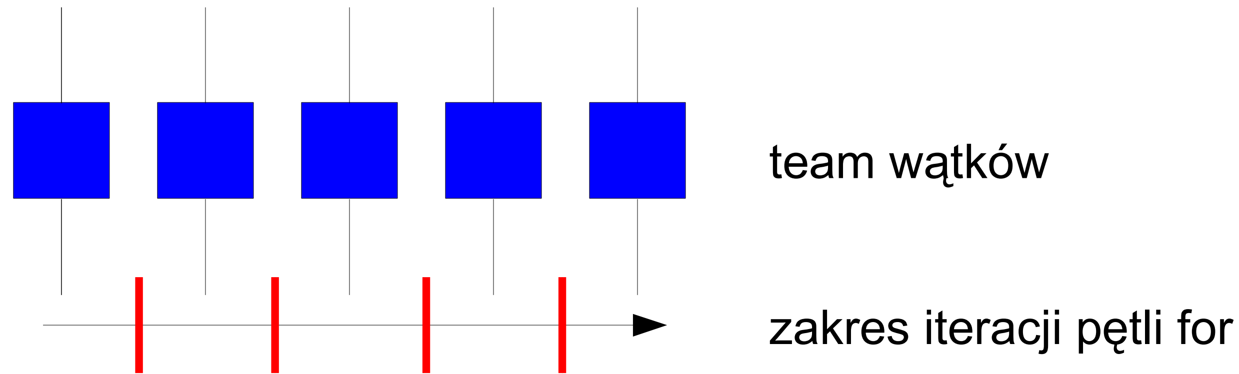
Dyrektywa for

- Dyrektywa for występuje wewnątrz konstruktu parallel (obie dyrektywy można połączyć w jedną).
- Musi poprzedzać bezpośrednio pętlę for.
- Ma ona postać:

```
#pragma omp for [dodatkowe klauzule]
instrukcja for
    instrukcje wykonujące się w pętli
```

- Bez dyrektywy for każdy wątek wykonałby oddzielnie całą pętlę.
- Z dyrektywą for nastąpi wspólne (ang. work sharing) wykonanie całej pętli. Każdy wątek otrzyma część zakresu iteracji pętli. Cała pętla zostanie wykonana ***równolegle*** przez wątki teamu.
- Zmienna sterująca pętli for jest zawsze zmienną prywatną.

Dyrektywa for



- Możliwy przykład (przy statycznej alokacji iteracji pętli)

```
#pragma omp for
   for(i=0;i<100;i++)
```

- Jeżeli mamy 10 wątków to możliwe jest, że wątek 0 wykona iteracje od 0 do 10, wątek 1 od 10 do 19, wątek 2 do 20 do 29, ..., wątek 9 od 90 do 99.

Warunki które musi spełniać pętla for w dyrektywie for

- W pętli for nie może wystąpić instrukcja break
- Zmienna sterująca pętli musi być typu całkowitego ze znakiem.
- Wyrażenie inicjujące tą zmienną musi być typu całkowitego.
- Wyrażenie logiczne na kontynuacje pętli musi być jednym z <,>,≤,≥.
- Wyrażenie inkrementujące musi powodować inkrementacje zmiennej sterującej o stałe składniki.
- Kompilator powie nam (komunikat o błędzie), gdy nie spełniamy warunków.

Dyrektywa for - przykład

```
#pragma omp parallel if (n>limit) default(none)\
shared(n,x,y) private(i)
{
   #pragma omp for
   for(i=0;i<n;i++)
      x[i]+=y[i];
}
```

- Podział na wątki wykonywany jest tylko wtedy, gdy liczba iteracji n jest większa od wartości limit. W przeciwnym wypadku pętla for wykonywana jest przez jeden wątek.
- Iteracje pętli for dzielone są pomiędzy wątki w teamie. Każdy wątek wykonuje część iteracji.
- Kompilator nie rozpoznający OpenMP też skompiluje ten program (w wersji szeregowej)
- Dyrektywę parallel można połączyć z for

```
#pragma omp parallel for if (n>limit) default(none) \
shared(n,x,y) private(i)
   for(i=0;i<n;i++)
      x[i]+=y[i];
```

Jakich typów zmiennych użyć

- Jeżeli wątek inicjalizuje i używa zmiennej (np. indeks pętli) których inne wątki nie używają, powinna ona być zadeklarowana jako prywatna.
- Jeżeli wątek często odczytuje zmienną, która została zainicjalizowana wcześniej w programie, korzystne jest utworzenie lokalnej kopii tej zmiennej i odziedziczenie wartości istniejącej w chwili utworzenia kopii. Należy użyć klauzuli `firstprivate`. W ten sposób każdy procesor będzie posiadał lokalną kopię zmiennej co prowadzi do lepszego wykorzystania pamięci cache.
- Jeżeli wiele wątków manipuluje pojedynczą zmienną, należy zbadać możliwość rozbicia tych manipulacji na lokalne operacje, po których nastąpi pojedyncza operacja globalna. Na przykład jeżeli wiele wątków zlicza jakieś zdarzenia, warto rozważyć utrzymywania lokalnych (dla każdego wątku) liczników i ich sumowanie po opuszczeniu regionu równoległego. Umożliwia to klauzula reduction.
- Dopiero na samym końcu należy deklarować zmienne jako współdzielone.
- Warto używać klauzli `default(none)` aby nie być zaskoczony domyślnym typem zmiennych.

Alokacja iteracji pętli for wątkom (1)

- Sposób alokacji iteracji pętli for dla wątków możemy zmieniać przy pomocy klauzuli `schedule` dyrektywy `pragma omp for`. Ma ona następującą składnię

```
schedule(static|dynamic|guided,size)
schedule(runtime)
```

- `schedule (static, size)`. Alokacja statyczna. Iteracje dystrybuowane są algorytmem round-robin wątkom w porcjach o rozmiarze size. Jeżeli size jest pominięte, wątek otrzymuje liczbę iteracji równą całkowitej liczbie iteracji pętli podzielonej przez liczbę wątków. ***Podział iteracji następuje przed rozpoczęciem wykonywania pętli***.

- Przykład. Pętla for z 16 iteracjami (0-15), 4 wątki

id wątku	0	1	2	3
brak size	0-3	4-7	8-11	12-15
size=2	0-1 8-9	2-3 10-11	4-5 12-13	6-7 14-15

Alokacja iteracji pętli for wątkom (2)

- `schedule (dynamic, size)`. Alokacja dynamiczna. Wykonywana jest w trakcie pracy pętli. Każdy wątek pobiera size iteracji z kolejki.
  - Jeżeli wątek skończy swoje pobrane iteracje, proces powtarza się wątek pobiera znowu size iteracji z kolejki.
  - Umożliwia dobre zrównoważenie obciążenia w sytuacji, gdy czas iteracji nie jest stały (przykład: generowanie zbioru Mandelbrota).
- `schedule (guided, size)`. Podobnie jak dynamic, ale rozmiar „porcji" iteracji jest zmniejszany wykładniczo. W niektórych przypadkach umożliwia osiągnięcie lepszej wydajności niż `dynamic`.
- Uwaga: szeregowanie metodą dynamic/guided obarczone jest znacznie większym narzutem związanym z organizacją równoległej pętli for niż static.
  - Konieczność utrzymania puli niewykorzystanych iteracji i konieczność synchronizacji dostępu do tej puli.
- `schedule (runtime)`. Określana w czasie wykonania programu na podstawie wartości zmiennej środowiskowej `OMP_SCHEDULE`.

Dyrektywy Sieroce

```
void work()
{
   #pragma omp for
   for(int i=0;i<n;i++)
   {
        :
   }
}
work() // sekwencyjna pętla for w jednym wątku
#pragma omp parallel
{
   work() // równoległa pętla for w teamie wątków
}
```

- Standard OpenMP pozwala na umieszczenie dyrektyw (np. `omp for`) poza kontekstem leksykalnym konstruktu `parallel`. W przykładzie dyrektywa `for` jest umieszczona wewnątrz odrębnej funkcji.
- Jeżeli funkcja zostanie wywołana spoza dynamicznego kontekstu regionu równoległego dyrektywa `for` jest ignorowana.
- Jeżeli funkcja zostanie wywołana wewnątrz dynamicznego kontekstu regionu równoległego dyrektywa `for` jest wykonana (podział iteracji pętli for pomiędzy wątki teamu)

Domyślne typy zmiennych

- Dane zaalokowane ze sterty (ang. heap) np. poprzez **malloc/new** są współdzielone. Podobnie zmienne globalne.
- Zmienne zadeklarowane przed konstruktem równoległym są współdzielone.
- Wyjątkiem jest indeks pętli for dyrektywy **#pragma omp for**, który jest zawsze zmienną prywatną.
- Zmienne automatyczne zadeklarowane w konstrukcie równoległym oraz w funkcjach wołanych z tego konstruktu są prywatne.
- Parametry funkcji wołanych z konstruktu równoległego są prywatne.
- Zmienne statyczne zadeklarowane w konstrukcie równoległym oraz w funkcjach wołanych z tego konstruktu są prywatne.
- Zachowania wewnątrz funkcji wołanych z konstruktu równoleglego nie można zmienić. (wyjątkiem jest dyrektywa threadprivate)

Domyślne typy - przykład

```
void f (int a[],int n) {
   int i,j,m=3;
#pragma omp parallel for
   for(i=0;i<n;i++) {
       int k=m;
       for(j=1;j<=5;j++)
           g(&a[i],&k,j);
   }
}
extern int c;
void g(int *x,int *y,int z) {
   int ii;
   static int cnt;
   cnt++;
   for(ii=0;ii<z;ii++)
      x=*y+c;
}
```

- Dane prywatne są deklarowane niebiesko a współdzielone czerwono.
- Użycie których zmiennych jest niebezpieczne ? Odp. cnt oraz j.

Klauzula lastprivate

- Jest używana w przypadku gdy wartość zmiennej prywatnej jest niezbędna po wyjściu z pętli for wykonywanej równolegle. Zapewnia ona, że ostatnia wartość zmiennej jest dostępna po zakończeniu regionu równoległego.

```
#pragma omp parallel
{
   #pragma omp for lastprivate(i)
      for (i=0; i<n-1; i++)
         a[i] = b[i] + b[i+1];
}
a[i]=b[i];
```

- Pytanie: co to jest ostatnia wartość zmiennej, w sytuacji gdy każdy wątek ma jej prywatną kopię ?
- Odpowiedź jest to wartość, którą zmienna przyjęłaby po szeregowym wykonaniu pętli for. W naszym przykładzie: n-1.

Dyrektywa threadprivate

- Zmienne globalne i statyczne są współdzielone.
- W niektórych sytuacjach chcemy aby każdy wątek posiadał prywatne dane, które istnieją przez cały czas obliczeń.
- Korzystamy z dyrektywy threadprivate. Powoduje ona, że zmienna globalna lub statyczna jest replikowana, tak że każdy wątek otrzymuje prywatną kopię.

```
int x; // x jest zmienną globalną.
#pragma omp threadprivate(x)
```

- Dyrektywa threadprivate musi wystąpić po definicji zmiennej.
- Problem powstanie jeżeli do zmiennej globalnej chcemy się odwoływać z kilku niezależnych regionów równoległych. Możliwe jest że w różnych regionach wystąpi różna liczba wątków.
  - OpenMP wymaga spełnienia kilku warunków aby wątki o tych samych identyfikatorach w dwóch regionach odwołały się do tej samej zmiennej. (np. liczba wątków w obydwu regionach musi być taka sama)

Dyrektywa sections

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      {
         taskA();
      }
      #pragma omp section
      {
         taskB();
      }
      #pragma omp section
      {
         taskC();
      }
   }
}
```

- Jeżeli w teamie są trzy wątki, to każda z funkcji `taskA()`, taskB(), taskC() wykona się w odrębnym wątku. Bo zakończeniu dyrektywy `sections` nastąpi barierowa synchronizacja wątków, chyba że użyliśmy klauzuli `nowait` w dyrektywie `sections`.

Synchronizacja w OpenMP

- Synchronizacja barrierowa (dyrektywa `barrier`).
- Sekcje krytyczne (dyrektywy `critical` i `atomic`)
- Wykonanie bloku kodu przez jeden wątek (dyrektywy `single` i `master`)
- Opróżnienie rejestrów do pamięci (dyrektywa `flush`)
- Funkcje operujące na zamkach.

Sekcje krytyczne

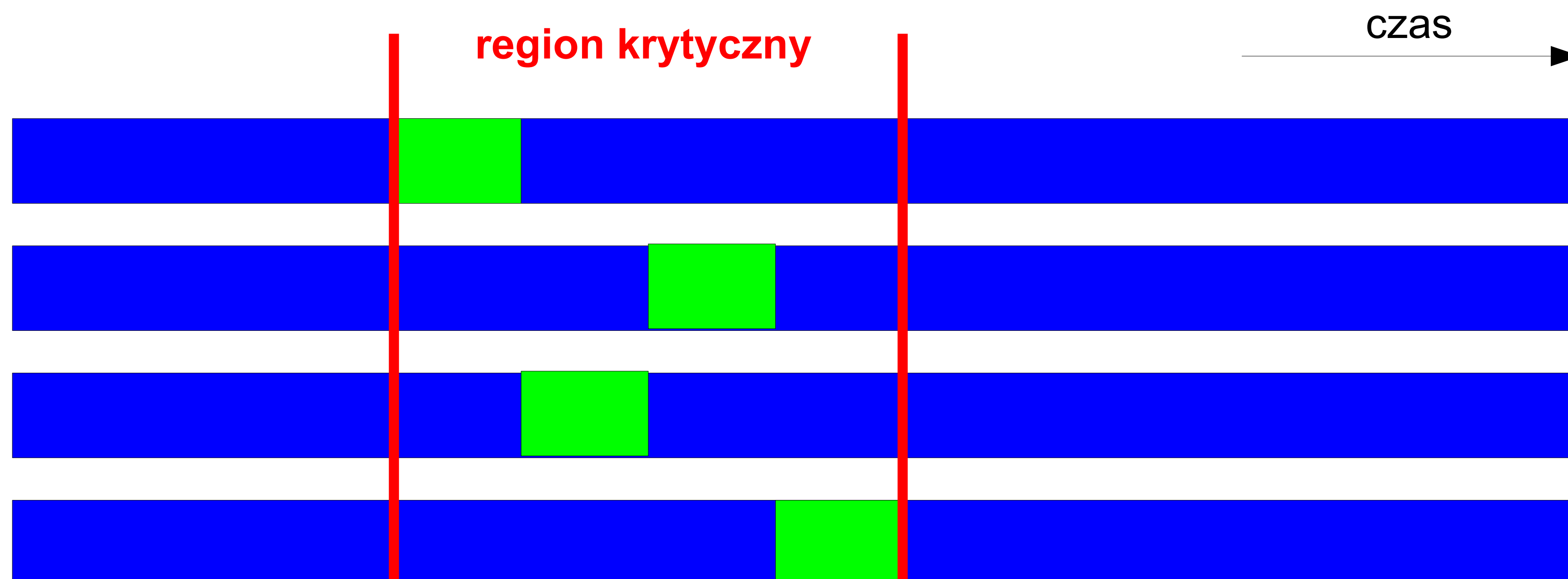
```
for(i=0;i<N;i++
{
    .....
    sum+=a[i];
    .....
}
```

- Jeżeli sum jest zmienną współdzieloną tylko jeden wątek może mieć do niej dostęp w danej chwili. (próba operacji na sum przez kilka wątków jednocześnie prowadzi do wyścigu). Należy zrównoleglić pętlę:

```
#pragma omp parallel for
for(i=0;i<N;i++)
{
    .....
    #pragma omp critical
    {
        sum+=a[i];
    }
    .....
}
```

- Instrukcja lub blok po dyrektywie `critical` wykonują się ze wzajemnym wyłączaniem. Tylko jeden wątek przebywa w sekcji krytycznej w danej chwili.

Sekcja krytyczna - wizualizacja



- Sekcja krytyczna jest użyteczna dla uniknięcia wyścigu.
- Nie można określić kolejności wejścia wątków do sekcji, jest ona niedeterministyczna.
- Kod zawarty w sekcji krytycznej jest kodem szeregowym i w żaden sposób nie jest przyspieszany przez obliczenia równoległe. ***Nadużywanie sekcji krytycznych obniża skalowalność !!!***
  - Należy starać się zminimalizować rozmiar (czas spędzany w) sekcji krytycznych.