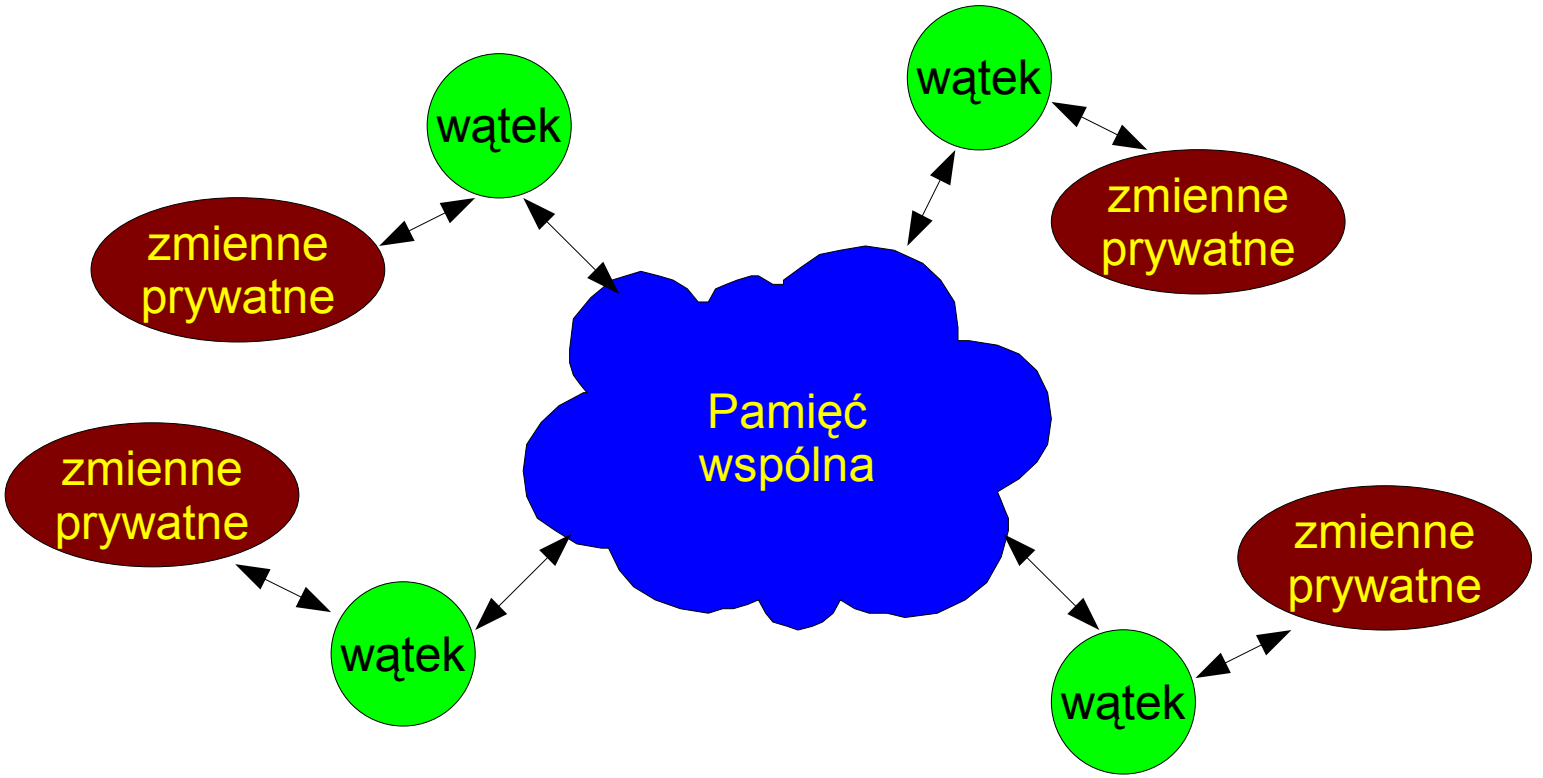
Programowanie maszyn z pamięcią wspólną w standardzie OpenMP.

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 (standardowy kompilator open source na Linuksie, ale niestety większość dystrybucji oparta jest na starszej wersji niż 4.2). 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 kompilatora.
  - 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

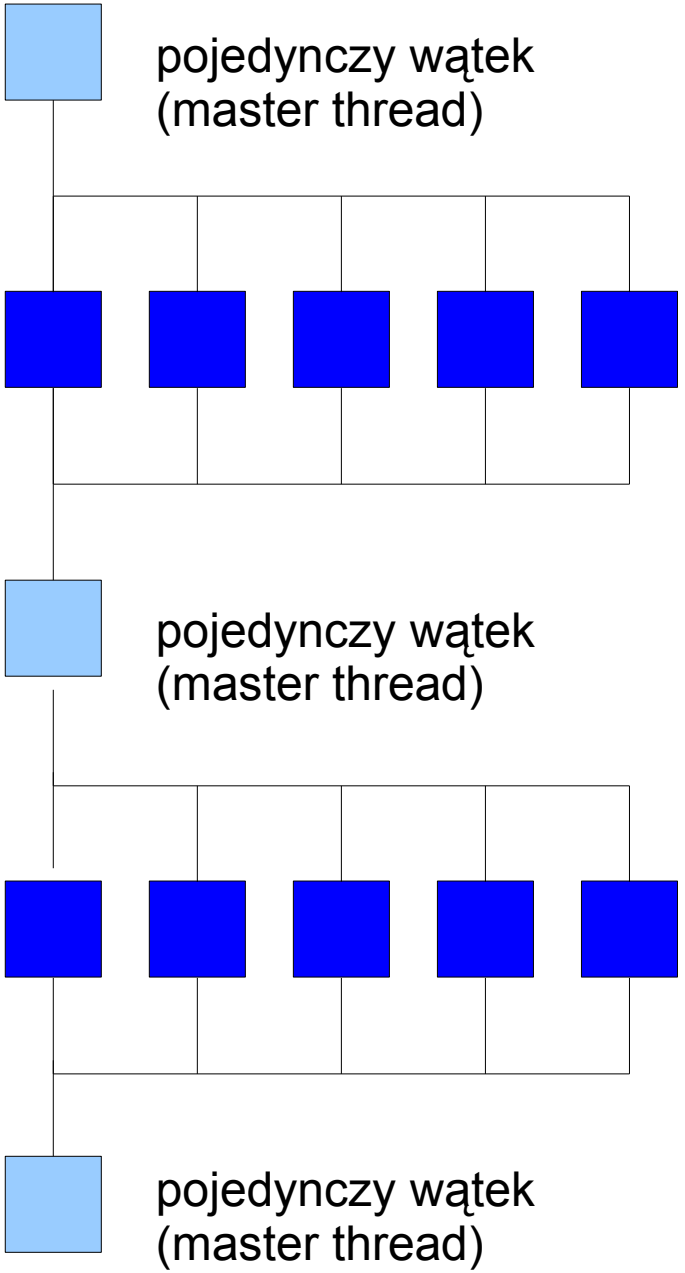
```
instrukcje szeregowe

#pragma omp parallel
{
    blok instrukcji
}

instrukcje szeregowe

#pragma omp parallel
{
    blok instrukcji
}

instrukcje szeregowe
```



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 */
```

- Dyrektywa parallel tworzy team wątków. Każdy wątek z teamu wykonuje blok instrukcji. Wątek który napotkał dyrektywę parallel staje się wątkiem głównym 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).

OpenMP - pierwszy program

```
// Plik nagłówkowy funkcji bibliotecznych
#include <omp.h>
#include <stdio.h>
int main()
{
   // Włącz dynamiczny wybór liczby wątków
   omp_set_dynamic(1);
   // 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.

  - Aby funkcja zadziałała wcześniej trzeba włączyć dynamiczne określanie liczby wątków wywołując funkcję:

  ```
  omp_set_dynamic(1)
  ```

- Po trzecie można skorzystać z dodatkowej klauzuli dyrektywy #pragma omp parallel:

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

Zachowanie się zmiennych w bloku 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. 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..

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.

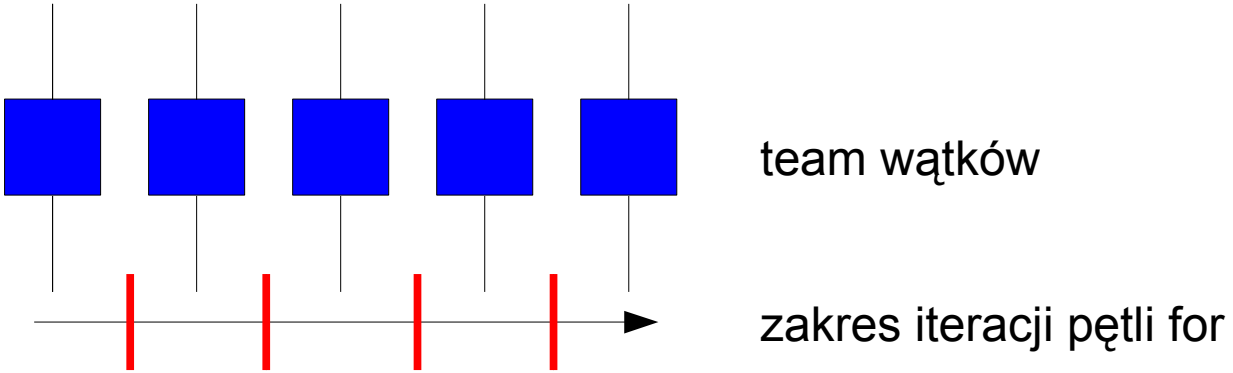
Dyrektywa for

- 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.

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
- 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];
```

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.
- 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. Umożliwia lepsze zbalansowanie obciążenia niż `dynamic`.
- `schedule (runtime)`. Określana w czasie wykonania programu na podstawie wartości zmiennej środowiskowej `OMP_SCHEDULE`.

Sieroca dyrektywa for

```
void work()
{
   #pragma omp for
   for(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 dyrektywy `parallel`. W przykładzie dyrektywa `for` jest umieszczona wewnątrz odrębnej funkcji.
- Jeżeli funkcja zostanie wywołana spoza dynamicznego kontekstu dyrektywy `parallel` dyrektywa `for` jest ignorowana.
- Jeżeli funkcja zostanie wywołana wewnątrz dynamicznego kontekstu dyrektywy `parallel` dyrektywa `for` jest wykonana (podział iteracji pętli for pomiędzy wątki teamu)

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

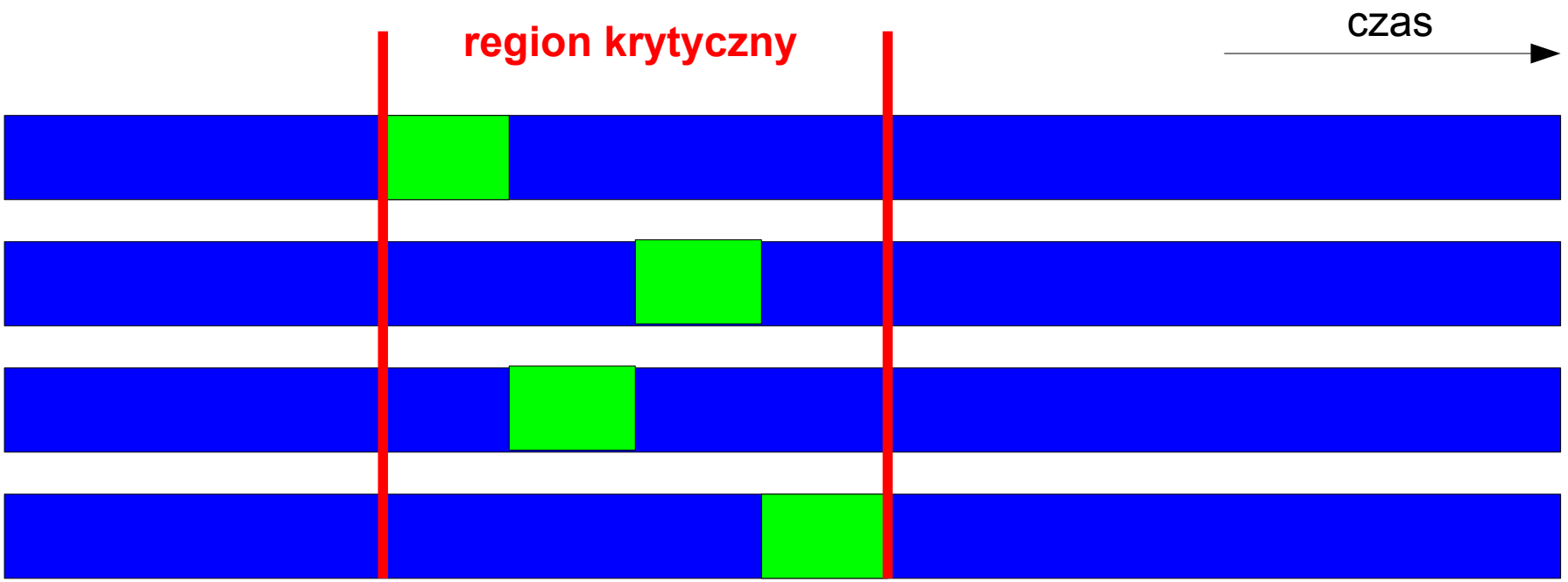
```
for(i=0;i<N;i++
{
    .....
    sum+=a[i];
    .....
}
```

- 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 wyściegu.
- 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.

Dyrektywa atomic

- Często sekcja krytyczna polega na uaktualnieniu jednej lokacji w pamięci. OpenMP dostarcza dyrektywę `atomic` dla tego typu sekcji krytycznych.
- Po dyrektywie `atomic` może wystąpić pojedyncza instrukcja uaktualniająca zmienną postaci: `x op=expr`, gdzie `expr` jest wyrażeniem, a `op` jest jednym z binarnych operatorów (dopuszczalne są również wyrażenia typu `x++` albo `--x`).
- Dyrektywa `atomic` serializuje odczyt i zapis zmiennej x w jednej sekcji krytycznej. Obliczanie wyrażenia `expr` nie jest serializowane !!!
- Każda dyrektywa `atomic` może być zastąpiona dyrektywą `critical`. Jednakże większość maszyn posiada wsparcie sprzętowe (specjalne instrukcje), sprawiające że dyrektywa `atomic` może być o wiele wydajniej zaimplementowana niż `critical`.

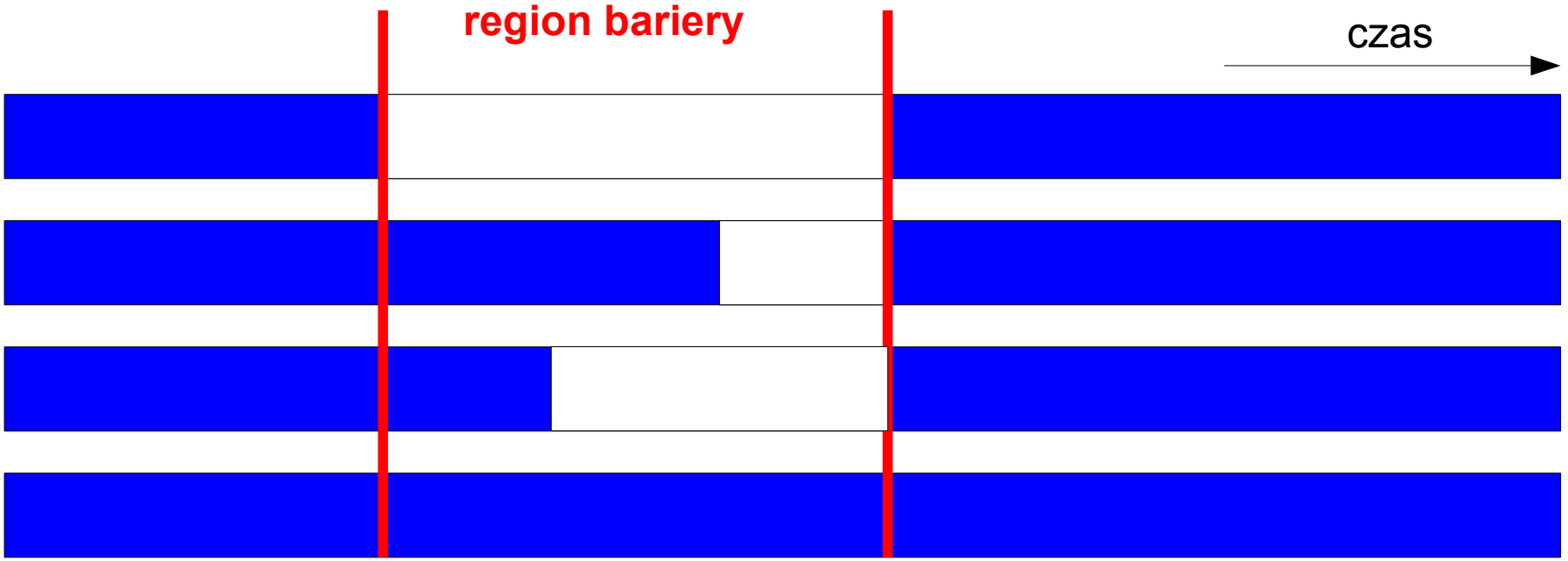
Dyrektywa barrier

- Jest najprostszą dyrektywą synchronizacyjną. Ma postać:

```
#pragma omp barrier
```

- Po napotkaniu tej dyrektywy wątek zostaje wstrzymany do momentu, gdy wszystkie inne wątki napotkają tę dyrektywę.
- Dyrektywa `barrier` jest wykonywana automatycznie po zakończeniu dyrektywy `for`, chyba że w dyrektywie `for` umieszczono klauzulę `nowait`.
  - Oznacza to że po zakończeniu zrównoleglonej pętli for wątek czeka aż pozostałe wątki zakończą pętlę for.

Dyrektywa barrier - wizualizacja



- Kolorem białym oznaczono oczekiwanie wątku.
- Wątek zostaje wypuszczony z punktu bariery dopiero wtedy, gdy wszystkie inne wątki osiągnęły ten punkt.
- Należy pamiętać, że implementacja dyrektywy barrier ma złożoność czasową O(log(p)), gdzie p jest liczbą procesorów.
  - Nadużycie dyrektywy barrier zmniejsza skalowalność.

Dyrektywy single i master

- Dyrektywa ta ma postać:

```
#pragma omp single [klauzule]
   blok_instrukcji
```

- Po napotkaniu dyrektywy single tylko jeden wątek wykona blok instrukcji. Pozostałe wątki przeskakują na koniec bloku.
- Pozostałe wątki czekają na wątek wykonujący blok instrukcji, chyba że została podana klauzula `nowait`.
- Dyrektywa `single` nie określa, który wątek wykona blok instrukcji. Specjalną formą dyrektywy `single` jest dyrektywa:

```
#pragma omp master [klauzule]
   blok_instrukcji
```

- Sprawia ona że blok instrukcji zostanie wykonany przez wątek nadrzędny (master) - o numerze 0.

Dyrektywa single - wizualizacja

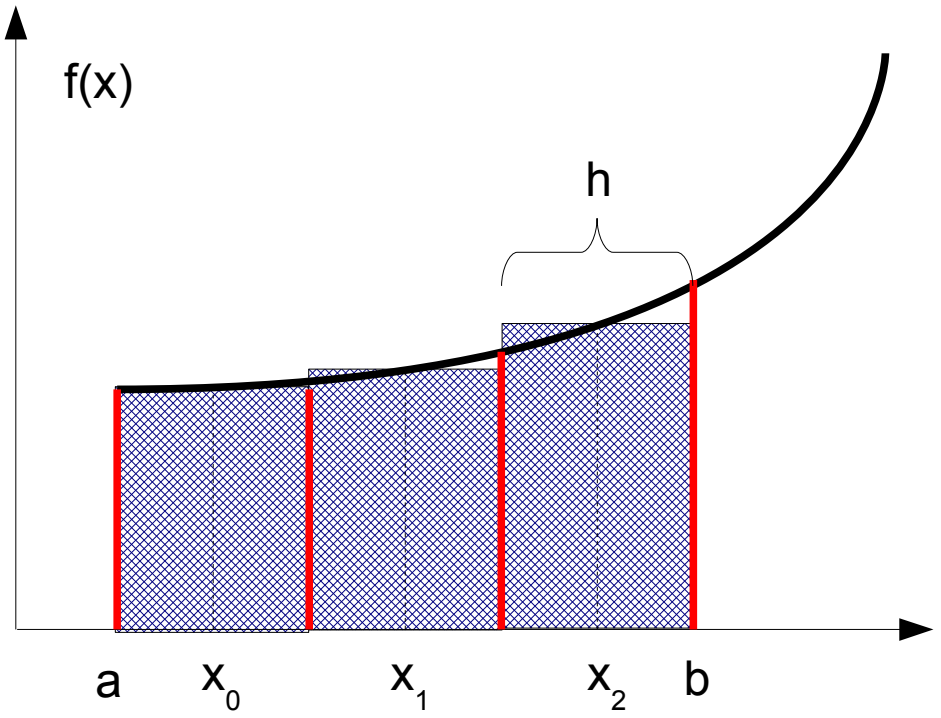
- Kolorem białym oznaczono oczekiwanie wątku.
- Przykłady zastosowania:
  - Obliczanie danych globalnych
  - Wejście-wyjście z bloku równoległego.

Funkcje operujące na zamkach

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
```

- Zamek realizuje wzajemne wykluczanie wątków.
- Przed użyciem należy go zainicjalizować poprzez `omp_init_lock`, a gdy staje się zbędny usunąć funkcją `omp_destroy_lock`.
- Funkcja `omp_set_lock` wchodzi, a `omp_unset_lock` wychodzi z sekcji krytycznej. Tylko jeden wątek może przebywać w sekcji krytycznej; pozostałe czekają w funkcji `omp_set_lock` na wejście.
- Funkcja `omp_test_lock` usiłuje wejść do sekcji krytycznej, i jeżeli w danej chwili nie jest to możliwe natychmiast powraca z wynikiem zero. Gdy udało się wejść do sekcji krytycznej natychmiast powraca z wynikiem różnym od zera.

Przykład: całkowanie metodą prostokątów



$$h = \frac{b-a}{N}$$

$$x_i = a + \frac{h}{2} + i * h$$

$$\int_a^b f(x) \approx h * \sum_{i=0}^{N-1} f(x_i)$$

- Całka oznaczona przybliżana przy pomocy sumy pól powierzchni N (w przykładzie N=3) prostokątów.
- Obliczenia można obsłużyć jedną pętlą for.

Całkowanie: wersja niewydajna

```
double Sum(int N,double a,double b)
{
   int i;
   double sum=0.0;
   double h=(b-a)/(double)N;

   #pragma omp parallel for default(none) firstprivate(N,h,a) shared(sum)
   for(i=0;i<N;i++) {
        double func=f(a+i*h+h/2);
   #pragma omp critical
                sum+=func;
   }
   return sum*h
}
```

- Ciekawostka: program skompiluje się do wersji jednowątkowej na kompilatorze nie obsługującym OpenMP
- `sum` jest zmienną współdzieloną w której gromadzona jest suma pól prostokątów.
- skoro dostęp do niej ma wiele wątków, które jednocześnie ją zwiększają, to musimy ją chronić sekcją krytyczną.
- Zastosowanie sekcji krytycznej obniża znacznie wydajność, ponieważ wątek będący w sekcji krytycznej i aktualizujący tę zmienną wstrzymuje pozostałe wątki chcące zrobić to samo.

Całkowanie: jak poprawić wydajność

- Idealnie byłoby gdyby każdy wątek posiadał prywatną kopię zmiennej sum, w której byłaby gromadzona suma pól prostokątów rozpatrywanych w iteracjach przydzielonych temu wątkowi.
  - Te lokalne sumy mogłyby być zsumowane po wyjściu z pętli for.
  - Takie rozwiązanie nie potrzebowałoby synchronizacji w każdej iteracji.
- Tak więc sum musi być zmienną prywatną.
- Pytanie: w jaki sposób zsumować prywatne zmienne ? Odpowiedź: za pomocą kluzuli `reduction(+:sum)`. Określa ona że:
  - każdy wątek w teamie ma prywatną kopię zmiennej sum.
  - po wyjściu z konstruktu wątek obliczana jest suma (znak +, możliwe są też inne operatory) prywatnych kopii i zapamiętywana w zmiennej sum wątku głównego (master).

Całkowanie: wersja z klauzulą reduction

```
double Sum(int N,double a,double b)
{
int Trials=N/size,i;
double sum=0.0;
double h=(b-a)/(double)N;

#pragma omp parallel for default(none)firstprivate(N,h,a)reduction(+:sum)
for(i=0;i<N;i++) {
   sum+=f(a+i*h+h/2);
}
return sum*h;
}
```

- Ciekawostka: program skompiluje się do wersji jednowątkowej na kompilatorze nie obsługującym OpenMP
- Brak sekcji krytycznej sprawia, że wątki w pętli wykonują się z maksymalną szybkością, a synchronizacja nastąpi dopiero gdy zakończą swoje iteracje (podczas dodawania prywatnych kopii zmiennej `sum`).

Podsumowanie

- OpenMP jest standardem programowania systemów z pamięcią wspólną.
- Określa standard programowania wielowątkowego i jest API wyższego poziomu niż POSIX threads.
- Opiera się na dyrektywach #pragma i wymaga wsparcia kompilatora.
- Jest rozpowszechniony w przemyśle software'owym (standard przemysłowy) i dlatego warto go poznać.
- Wraz z rozpowszechnianiem się procesorów wielordzeniowych jego znaczenie będzie rosnąć.