Dyrektywy zrównoleglające kompilatorów dla maszyn SM

Standard OpenMP

(PRiR, wykład 4)

Standard OpenMP

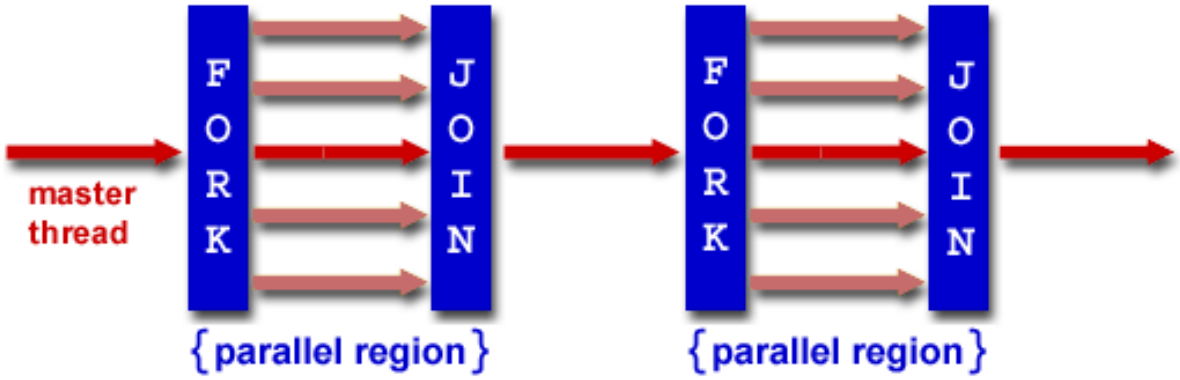
- Porozumienie producentów (IBM, SGI, Sun, HP, Compaq, Intel) – jednolite środowisko dyrektyw zrównoleglających dla maszyn z pamięcią wspólną:
  - wsparcie dla języków Fortran 90, C/C++,
  - biblioteka procedur,
  - zmienne środowiskowe.
- Oznaczone sekwencją:

  !$OMP – Fortran,

  #pragma omp – C/C++.

Model programowy OpenMP

- Pamięć współdzielona
- Dyrektywy kompilatora
- Przetwarzanie wielowątkowe
- Model rozwidlenie – połączenie



Sekcja równoległa

#pragma omp parallel [clause ...] newline

if (scalar_expression)

private (list)

shared (list)

default (shared | none)

firstprivate (list)

reduction (operator: list)

copyin (list)

num_threads (integer-expression)

structured_block

Sekcja równoległa, cd

- Kod sekcji równoległej jest duplikowany i wykonywany w równoległych wątkach
- Na końcu bloku zrównoleglonego znajduje się domyślna bariera. Po jej przekroczeniu działa tylko wątek główny
- Jeśli jeden z wątków się ,,wywali”, inne też staną. Wynik jest nieokreślony.

Ustalenie liczby wątków

Wg następującego porządku:

1. sprawdzenie IF (jeśli zdefiniowane)
2. parametr NUM_THREADS
3. funkcja omp_set_num_threads()
4. zmienna środowiskowa OMP_NUM_THREADS
5. domyślna wartość (liczba CPU)

Zmienna liczba wątków

Funkcja

omp_get_dynamic()

pozwala ustalić, czy można.

Jeśli można to do wyboru:

1. omp_set_dynamic()
2. env OMP_DYNAMIC=TRUE

Zagnieżdżanie

* Funkcja

  omp_get_nested ()

  pozwala ustalić, czy można.

  Jeśli można to do wyboru:

  1. omp_set_nested()
  2. env OMP_NESTED=TRUE

* Jeśli nie można, to kompilator ignoruje

Ograniczenia

- Zagnieżdżony blok nie może zawierać wielu funkcji, a tym bardziej plików z kodem
- Nie można do/z niego w/wyskakiwać
- Dopuszczalny jest tylko jeden parametr IF
- Dopuszczalny jest tylko jeden parametr NUM_THREADS

Przykład

```
#include <omp.h>

main ()  {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and terminate */

}
```

Przykład

```
jstar@gaus:~/prir/openMP$ cc p1.c
/tmp/ccPYU7Hm.o: In function `main':
p1.c:(.text+0x9): undefined reference to `omp_get_thread_num'
p1.c:(.text+0x29): undefined reference to `omp_get_num_threads'
collect2: ld returned 1 exit status
jstar@gaus:~/prir/openMP$ cc -fopenmp p1.c
jstar@gaus:~/prir/openMP$ ./a.out
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 2
Hello World from thread = 3
Hello World from thread = 1
jstar@gaus:~/prir/openMP$  env OMP_NUM_THREADS=2 ./a.out
Hello World from thread = 0
Number of threads = 2
Hello World from thread = 1
jstar@gaus:~/prir/openMP$
```

Zmienne - zasięg

```
#pragma omp parallel [clause ...]  newline
                    if (scalar_expression)
                    private (list)
                    shared (list)
                    default (shared | none)
                    firstprivate (list)
                    reduction (operator: list)
                    copyin (list)
                    num_threads (integer-expression)
   structured_block
```

Zmienne PRIVATE i THREADPRIVATE

- zmienne PRIVATE
  - nowy obiekt jest tworzony dla każdego wątku
  - wszystkie odwołania do obiektu są zastępowane przez odwołania do nowego obiektu
  - nowy obiekt jest niezainicjowany
- Zmienne THREADPRIVATE
  - wyliczone zmienne globalne są lokalne dla wątków i zachowują wartość w czasie, gdy wątki są nieaktywne

```c
#include <omp.h>

int  a, b, i, tid;
float x;

#pragma omp threadprivate(a, x)

main ()  {

/* Explicitly turn off dynamic threads */
  omp_set_dynamic(0);

  printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid)
  {
  tid = omp_get_thread_num();
  a = tid;
  b = tid;
  x = 1.1 * tid +1.0;
  printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }  /* end of parallel section */

  printf("************************************\n");
  printf("Master thread doing serial work here\n");
  printf("************************************\n");

  printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
  {
  tid = omp_get_thread_num();
  printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }  /* end of parallel section */

}
```

```
1st Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 2:   a,b,x= 2 2 3.200000
Thread 3:   a,b,x= 3 3 4.300000
Thread 1:   a,b,x= 1 1 2.100000
************************************
Master thread doing serial work here
************************************
2nd Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 3:   a,b,x= 3 0 4.300000
Thread 1:   a,b,x= 1 0 2.100000
Thread 2:   a,b,x= 2 0 3.200000
```

Zmienne SHARED

- Są współdzielone przez wątki
  - W pamięci istnieje tylko jedna kopia
  - Wszystkie wątki mają do niej dostęp
  - Programista ma zapewnić poprawny dostęp wątków do zmiennej

Deklaracja DEFAULT

… default (shared | none)

- shared – domyślnie zmienne są współdzielone
- none – zasady dostępu muszą być jawnie określone

Inne deklaracje

- FIRSTPRIVATE(lista) – jak PRIVATE, lecz zmienne lokalne z listy we wszystkich wątkach mają wartość początkową jak przed wejściem do bloku równoległego.
- LASTPRIVATE(lista) – jak PRIVATE, lecz po zakończeniu bloku zmienne z listy będą miały wartość jak pod koniec ostatniej instrukcji w wersji sekwencyjnej (np. ostatniej iteracji pętli).
- COPYIN – kopiowanie wartości z master_thread do zmiennych THREADLOCAL
- COPYPRIVATE – kopiowanie z wątku do wątku (związane z SINGLE)
- REDUCTION(operator : lista) – obliczenie łącznych wartości operacji: +, *, -, --, ++, &,&&, ^,…

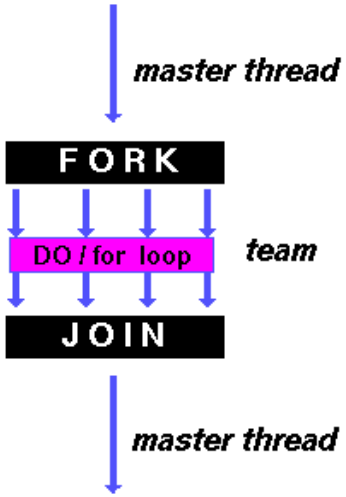
Przykład

```
#include <omp.h>

main ()  {

int   i, n, chunk;
float a[100], b[100], result;

/* Some initializations */
n = 100;
chunk = 10;
result = 0.0;
for (i=0; i < n; i++)
  {
  a[i] = i * 1.0;
  b[i] = i * 2.0;
  }

#pragma omp parallel for      \
  default(shared) private(i)  \
  schedule(static,chunk)      \
  reduction(+:result)

  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);

printf("Final result= %f\n",result);

}
```

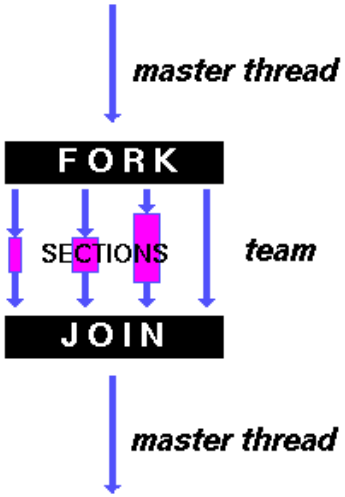
O tym za moment

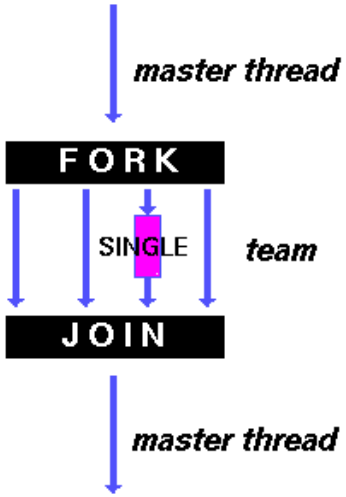
Podział pracy

for (SPMD)



sections (MP)

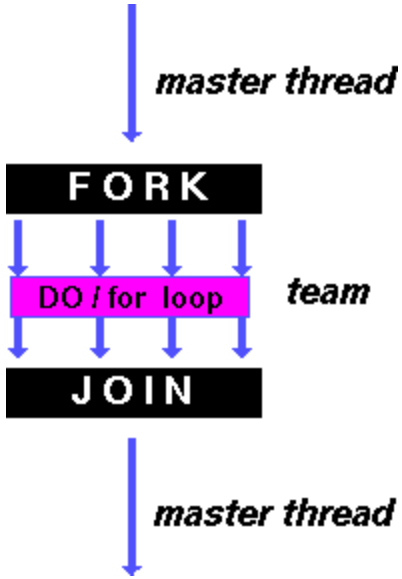




for

```
#pragma omp for [clause ...]  newline
            schedule (type [,chunk])
            ordered
            private (list)
            firstprivate (list)
            lastprivate (list)
            shared (list)
            reduction (operator: list)
            collapse (n)
            nowait
```

pętla_for



Dyrektywy dla for

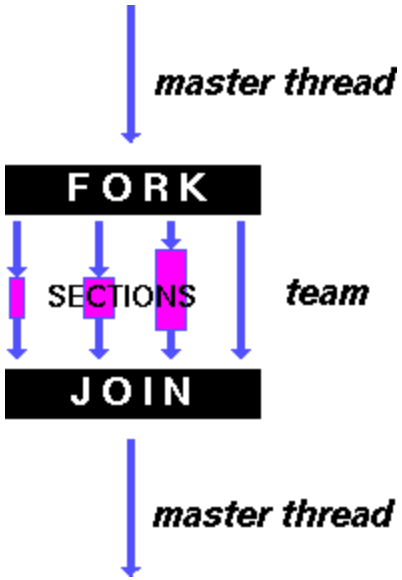
- schedule: jak podzielić iteracje między watki:
  - static – na kawałki i statycznie do watków . Można określić (chunk = wielkość porcji)
  - dynamic na kawałki i dynamicznie do wątków Domyślny chunk=1
  - guided jeśli chunk=1, to liczba iteracji = liczba_niedowiązanych_iteracji / liczba_wątków malejąco do 1
    jeśli chunk=k, to k jest dolnym ograniczeniem na wielkość porcji
  - runtime – decyzja o podziale dopiero po uruchomieniu
  - auto = rób jak chcesz
- nowait: wątki nie muszą się zsynchronizować na koniec pętli
- ordered: kolejność iteracji ma być taka sama, jak w programie sekwencyjnym.
- collapsed: jak wiele pętli w zagłębionej konstrukcji pętli, ma być połączone w jedną większą przestrzeń iteracji i podzielone na kawałki. Kolejność iteracji pozostanie, jak w programie sekwencyjnym.

Przykład

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()
{

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {

  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }  /* end of parallel section */

}
```

Bloki wariantowe (sections)

```
#pragma omp sections [clause ...]  newline
                    private (list)
                    firstprivate (list)
                    lastprivate (list)
                    reduction (operator: list)
                    nowait
 {

 #pragma omp section   newline

    structured_block

 #pragma omp section   newline

    structured_block

 }
```



Przykład

```
#include <omp.h>
#define N     1000

main ()
{

int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = i * 1.5;
  b[i] = i + 22.35;

#pragma omp parallel shared(a,b,c,d) private(i)
  {

  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */

  }  /* end of parallel section */

}
```

Przejście sekwencyjne

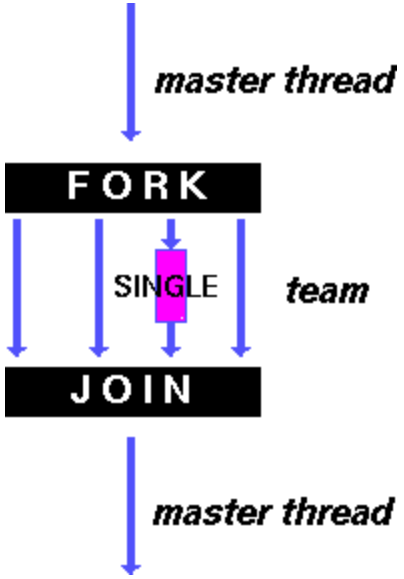
pragma omp single [clause ...] newline

private (list)

firstprivate (list)

nowait

structured_block



Zadanie (task)

- Wprowadzone w OpenMP 3.0 dla ułatwienia zrównoleglania pętli typu while i rekurencji
- Wspierają implementację zagnieżdżonej równoległości
- Dwie dyrektywy:
  - task
  - taskwait

Koncepcja OR ma wady

- Wejście/wyjście z obszar równoległy wymaga często nakładów większych, niż same wykonanie zadania.
- Liczba wątków obsługujących OR jest ustalana przy jego tworzeniu i nie może się zmienić w trakcie jego wykonywania (nawet, gdyby były takie możliwości).
- Użytkownik musi zadecydowac o liczbie watków na każdym poziomie zrównoleglania, bo watek nie może obsługiwać kilku OR.
- Zbyt mało lub zbyt dużo wątków = zła skalowalność lub marnowanie zasobów.

Sortowanie szybkie

```
void quick_sort (int p, int r, float *data)
{
    if (p < r) {
            int q = partition (p, r, data);
            #pragma omp parallel sections firstprivate(data, p, q, r)
            {
                #pragma omp section
                quick_sort (p, q-1, data);
                #pragma omp section
                quick_sort (q+1, r, data);
            }
        }
    }
}
```

```
void par_quick_sort (int n, float *data)
{
    quick_sort (0, n, data);
}
```

Sortowanie szybkie

```
void quick_sort (int p, int r, float *data)
{
    if (p < r) {
            int q = partition (p, r, data);
            #pragma omp task
            quick_sort (p, q-1, data);
            #pragma omp task
            quick_sort (q+1, r, data);
        }
    }
}
```

```
void par_quick_sort (int n, float *data)
{
    #pragma omp parallel
    {
        #pragma omp single nowait
        quick_sort (0, n, data);
    }
}
```

Porównanie efektywności

Experimental Result

First, we set environment variable OMP_THREAD_LIMIT so that the program will use a max of 8 threads.

The experiment was performed on a machine with 8 processors. The number of threads for the parallel region example is set by setting the OMP_NUM_THREADS environment variable.

We obtain these performance numbers:

OMP_NUM_THREADS	task	parreg
2	2.6s	1.8s
4	1.7s	2.1s
8	1.2s	2.6s

The program written with tasks achieves a good scalable speedup, while the program written with nested parallel regions does not.

http://wikis.sun.com/display/openmp/Tasks+vs+Nested+Parallel+Regions

Synchronizacja

Stary problem:

THREAD 1:

```
increment(x)
{
    x = x + 1;
}
```

THREAD 1:

```
10   LOAD A, (x address)
20   ADD A, 1
30   STORE A, (x address)
```

THREAD 2:

```
increment(x)
{
    x = x + 1;
}
```

THREAD 2:

```
10   LOAD A, (x address)
20   ADD A, 1
30   STORE A, (x address)
```

Dyrektywy do synchronizacji

- master – tylko wątek główny
- critical – tylko jeden w danym czasie
- barrier – wszystkie muszą dojść
- taskwait – zaczekaj na wszystkie zadania
- atomic – nie oddawaj procesora
- flush – zapisz wszystko do wspólnej pamięci
- ordered – jakby w sekwencyjnym

master

```
#pragma omp master
        structured_block
```

- Fragment kodu wykonywany tylko przez główny wątek zespołu; inne wątki pomijają ten region
- Nie wprowadza żadnej bariery
- Nie można w/wyskakiwać do/z bloku.

critical

#pragma omp critical [name]
structured_block

- Fragment kodu wykonywany tylko przez jeden wątek naraz; inne wątki czekają
- Nazwa pozwala wprowadzić kilka regionów krytycznych.
  - Nazwy są globalne
  - Wszystkie sekcje nienazwane tworzą ten sam region
- Nie można w/wyskakiwać do/z bloku.

Przykład

```
#include <stdio.h>
#include <omp.h>
main() {
    double      x;
    int         i;
    x = 0;

#pragma omp parallel for shared(x)
    for (i = 0; i < 1000; i++) {
        x =  x + 1;
    }                       /* end of parallel section */

    printf("s=%g\n", x);
    return 0;
}
```

```
jstar@gaus:~/prir/openMP$ vi px.c
jstar@gaus:~/prir/openMP$ cc -fopenmp px.c
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=1 ./a.out
s=1000
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=2 ./a.out
s=876
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=2 ./a.out
s=857
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=8 ./a.out
s=934
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=8 ./a.out
s=980
jstar@gaus:~/prir/openMP$
```

Przykład

```
#include <stdio.h>
#include <omp.h>
main() {
    double      x;
    int         i;
    x = 0;

#pragma omp parallel for shared(x)
    for (i = 0; i < 1000; i++) {
        #pragma omp critical
        x = x +  1;
    }                       /* end of parallel section */

    printf("s=%g\n", x);
    return 0;
}
```

```
jstar@gaus:~/prir/openMP$ cc -fopenmp px.c
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=1 ./a.out
s=1000
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=2 ./a.out
s=1000
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=7 ./a.out
s=1000
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=8 ./a.out
s=1000
jstar@gaus:~/prir/openMP$
```

atomic

#pragma omp atomic
statement_expression

- Wyrażenie jest wykonywane ,,atomowo”, bez oddawania sterowania.
- Wyrażenie nie może być dowolne – różne ograniczenia w różnych wersjach OpenMP.

- Można powiedzieć, że to takie „mini-critical”

Przykład

```
#include <stdio.h>
#include <omp.h>
main() {
    double      x;
    int         i;
    x = 0;

#pragma omp parallel for share
    for (i = 0; i < 1000; i++) {
        #pragma omp atomic
        x = x +  1;
    }
    printf("s=%g\n", x);
    return 0;
}
```

x = ~~x + 1~~ += 1;

```
jstar@gaus:~/prir/openMP$ vi px.c
jstar@gaus:~/prir/openMP$ cc -fopenmp px.c
px.c: In function âmainâ:
px.c:15: error: invalid operator for ‘’#pragma omp atomic” before ‘’=‘’ token
jstar@gaus:~/prir/openMP$ vi px.c
jstar@gaus:~/prir/openMP$ cc -fopenmp px.c
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=8 ./a.out
s=1000
jstar@gaus:~/prir/openMP$ env OMP_NUM_THREADS=5 ./a.out
s=1000
jstar@gaus:~/prir/openMP$
```

flush

#pragma omp flush [(list)]

- Ustala punkt, w którym (wszystkie) lokalne zmienne wątków muszą zostać zapisane do wspólnej pamięci. (Np. w systemach z cache procesora powinna w takim miejscu nastąpić synchronizacja.)
- Dyrektywa mocno dyskutowana.
- Lista pozwala określić, co chcemy zsynchronizować (jeśli nie wszystko).
- Lista: uwaga na wskaźniki – to one są zapisywane!

Przykład

```
#pragma omp parallel sections
   num_threads(2)
{
   #pragma omp section
   {
      printf_s("Thread %d: ",
            omp_get_thread_num( ));
      read(&data);
      #pragma omp flush(data)
      flag = 1;
      #pragma omp flush(flag)
      // Do more work.
   }
   ….
```

```
      ….
      #pragma omp section
      {
         while (!flag) {
            #pragma omp flush(flag)
         }
         #pragma omp flush(data)

         printf_s("Thread %d: ",
               omp_get_thread_num( ));
         process(&data);
         printf_s("data = %d\n", data);
      }
   }
}
```

flush niejawny

- Analog flusha jest domyślnie wykonywany przy:
  - barrier
  - parallel – wejście do i wyjście z
  - critical – wejście do i wyjście z
  - ordered – wejście do i wyjście z
  - for – wyjście z
  - sections – wyjście z
  - single – wyjście z

ordered

```
#pragma omp for ordered [clauses...]
   (loop region)
#pragma omp ordered  newline
   structured_block
   (endo of loop region)
```

- Pętla jest wykonywana tak, jak pętla wykonywana sekwencyjnie
- Wątki muszą czekać na dokończenie poprzednich iteracji.

threadprivate

#pragma omp threadprivate (list)

- Używana do zmiennych globalnych.
- Musi występować po deklaracji takich zmiennych.
- Powoduje, że wyliczone zmienne są kopiowane dla poszczególnych wątków
- Zmienne zachowują wartość w ramach poszczególnych wątków przez szereg regionów równoległych.

(Przykład na slajdzie nr 14)

Funkcje OpenMP

- Funkcje OpenMP służą do:
  - ustalania liczby wątków/procesorów oraz określania potrzebnej liczby wątków,
  - zabaw z zamkami (semaforami)
  - określania czasu
  - modyfikacji środowiska wykonawczego (zagnieżdżona równoległość, dynamiczne określenie liczby wątków).

Liczba wątków

```
#include <omp.h>
void omp_set_num_threads(int num_threads)
int omp_get_num_procs(void)
int omp_get_num_threads(void)
```

- Działanie set zależy od tego, czy mechanizm dynamicznego przydzielania liczby wątków jest włączony – jeśli tak, to określamy maksimum, jeśli nie, to dokładną liczbę.
- Set można wywołać tylko z kodu sekwencyjnego.
- Set jest ważniejsze, niż liczba ustalona przez środowisko.

Liczba wątków

```
#include <omp.h>
int omp_get_max_threads(void)
int omp_get_thread_num (void)
int omp_get_thread_limit (void)
```

- get_max zwraca to, co ustawiono przez zmienna środowiskową, lub funkcję set_num (poprz. slajd).
- get_max można wołać z dowolnego miejsca.
- get_num zwraca 0 jeśli została wywołana z kodu sekwencyjnego, lub zagnieżdżonego równoległego.
- get_limit to nowość w OpenMP 3

Środowisko

```
#include <omp.h>
int omp_get_in_parallel(void)
int omp_get_set_dynamic (void)
int omp_get_get_dynamic (void)
void omp_set_nested(int)
int omp_get_nested(void)
```

- in_parallel zwraca zero, jeśli wywołana z kodu sekwencyjnego, nie-zero, jeśli z równoległego
- get_dynamic zwraca zero, jeśli nie jest włączone dynamiczne sterowanie liczbą wątków.
- set_dynamic w/wyłącza dynamiczne sterowanie liczbą wątków.

Zamki: tworzenie/niszczenie

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock)
void omp_init_nest_lock(omp_nest_lock_t *lock)
void omp_destroy_lock(omp_lock_t *lock)
void omp_destroy_nest__lock(omp_nest_lock_t *lock)
```

- Nowoutworzony zamek jest otwarty.
- Błędem jest niszczenie zamka nie utworzonego.
- Zagnieżdżane zamki to nowość w OpenMP 3.0.

Zamki: zamykanie/otwieranie

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock)
void omp_set_nest__lock(omp_nest_lock_t *lock)
void omp_unset_lock(omp_lock_t *lock)
void omp_unset_nest__lock(omp_nest_lock_t *lock)
int omp_test_lock(omp_lock_t *lock)
int omp_test_nest__lock(omp_nest_lock_t *lock)
```

- Błędem jest operowanie na zamku nie utworzonym.

Przykład

```
#include <stdio.h>
#include <omp.h>
omp_lock_t  my_lock;
int main() {
   double x=0;
   omp_init_lock(&my_lock);
   #pragma omp parallel num_threads(4)
   {
      int tid = omp_get_thread_num( );
      int i, j;

      for (i = 0; i < 1000; ++i) {
         //omp_set_lock(&my_lock);
         x = x + 1;
         //omp_unset_lock(&my_lock);
      }
   }
   printf( "x=%g\n", x );

   omp_destroy_lock(&my_lock);
}
```

Usuwamy //

```
jstar@gaus:~/prir/openMP$ vi py.c
jstar@gaus:~/prir/openMP$ cc -fopenmp py.c
jstar@gaus:~/prir/openMP$ ./a.out
x=3004
jstar@gaus:~/prir/openMP$ vi py.c
jstar@gaus:~/prir/openMP$ cc -fopenmp py.c
jstar@gaus:~/prir/openMP$ ./a.out
x=3000
jstar@gaus:~/prir/openMP$ ./a.out
x=1815
jstar@gaus:~/prir/openMP$ ./a.out
x=2036
jstar@gaus:~/prir/openMP$ vi py.c
jstar@gaus:~/prir/openMP$ cc -fopenmp py.c
jstar@gaus:~/prir/openMP$ ./a.out
x=4000
jstar@gaus:~/prir/openMP$ ./a.out
x=4000
jstar@gaus:~/prir/openMP$ ./a.out
x=4000
jstar@gaus:~/prir/openMP$
```

Pomiar czasu

```
#include <omp.h>
double omp_get_wtime(void)
double omp_get_wtick(void)
```

- get_wtime – zwraca czas jako double czas w sekundach. Trzeba wywołać dwa razy i odjąć. Wymyślona jako „zegar dla wątku”.
- get_wtick – zwraca ilość sekund na jeden „tyk” zegara jako double.

Zmienne środowiskowe

OMP_SCHEDULE

setenv OMP_SCHEDULE "guided, 4"

setenv OMP_SCHEDULE "dynamic“

OMP_NUM_THREADS

setenv OMP_NUM_THREADS 8

OMP_DYNAMIC

setenv OMP_DYNAMIC TRUE

OMP_NESTED

setenv OMP_NESTED TRUE

Zmienne środowiskowe, 3.0

OMP_STACKSIZE

New with OpenMP 3.0. Controls the size of the stack for created (non-Master) threads.

setenv OMP_STACKSIZE 2000500B

setenv OMP_STACKSIZE "3000 k "

setenv OMP_STACKSIZE 10M

setenv OMP_STACKSIZE " 10 M "

setenv OMP_STACKSIZE "20 m "

setenv OMP_STACKSIZE " 1G"

setenv OMP_STACKSIZE 20000

OMP_WAIT_POLICY

New with OpenMP 3.0. Provides a hint to an OpenMP implementation about the desired behavior of waiting threads

setenv OMP_WAIT_POLICY ACTIVE

setenv OMP_WAIT_POLICY PASSIVE

OMP_MAX_ACTIVE_LEVELS

New with OpenMP 3.0. Controls the maximum number of nested active parallel regions

setenv OMP_MAX_ACTIVE_LEVELS 2

OMP_THREAD_LIMIT

New with OpenMP 3.0. Sets the number of OpenMP threads to use for the whole OpenMP program.

setenv OMP_THREAD_LIMIT 8

Przykład

- Program obliczający iloczyn skalarny dwu wektorów 100-elementowych.
- Zapisuje do pliku numery iteracji wykonywanych przez procesory wchodzące do sekcji krytycznej.
- Aby wystąpiło zrównoleglenie należy ustawić odpowiednią wartość zmiennej środowiskowej OMP_NUM_THREADS.

Program OpenMP w języku C

test-openmp.c

Thread safety

- Przy pisaniu programów wielowątkowych wykorzystujących funkcje biblioteczne należy pamiętać o sprawdzeniu, czy są one wielowątkowo bezpieczne (*thread safe*).
- Chodzi o poprawność działania procedury niezależnie od kontekstu (wątku) jej wywołania.
- Dotyczy to szczególnie procedur z pamięcią np. generatorów liczb pseudolosowych.